SISTEMAS OPERATIVOS

Pedro de Miguel Anasagasti
Fernando Pérez Costoya

Departamento de Arquitectura y Tecnología de Sistemas Informáticos
Escuela Técnica Superior de Informática
Universidad Politécnica de Madrid
18-05-2016

Este libro se deriva del libro “Sistemas Operativos. Una visión aplicada” editado en el 2007 y cuyos autores son D. Jesús Carretero Pérez, D. Félix Garcia Carballeira, D. Pedro de Miguel Anasagasti y D. Fernando Pérez Costoya.

El presente libro tiene un enfoque mucho menos generalista y está especialmente dirigido a los alumnos de Informática de la “Escuela Técnica Superior de Informática” de la “Universidad Politécnica de Madrid”.

CONTENIDO

1

CONCEPTOS ARQUITECTÓNICOS DEL COMPUTADOR

En este capítulo se presentan los conceptos de arquitectura de computadores más relevantes desde el punto de vista de los sistemas operativos. El capítulo no pretende convertirse en un tratado de arquitectura, puesto que su objetivo es el de recordar y destacar los aspectos arquitectónicos que afectan de forma directa al sistema operativo.

Para alcanzar este objetivo el capítulo se estructura en los siguientes grandes temas:

- *Funcionamiento básico de los computadores y estructura de los mismos.*
- *Modelo de programación con énfasis en su secuencia de ejecución.*
- *Concepto de interrupción y sus tipos.*
- *Diversas acepciones de reloj.*
- *Aspectos más relevantes de la jerarquía de memoria y, en especial, de la memoria virtual.*
- *Concurrencia de la E/S con el procesador.*
- *Mecanismos de protección.*
- *Multiprocesador y multicomputador.*
- *Prestaciones del sistema.*

1.1. ESTRUCTURA Y FUNCIONAMIENTO DEL COMPUTADOR

El computador es una máquina destinada a procesar datos. En una visión esquemática, como la que muestra la figura 1.1, este procesamiento involucra dos flujos de información: el de datos y el de instrucciones. Se parte del flujo de datos que han de ser procesados. Este flujo de datos es tratado mediante un flujo de instrucciones máquina, generado por la ejecución de un programa, produciendo el flujo de datos resultado.

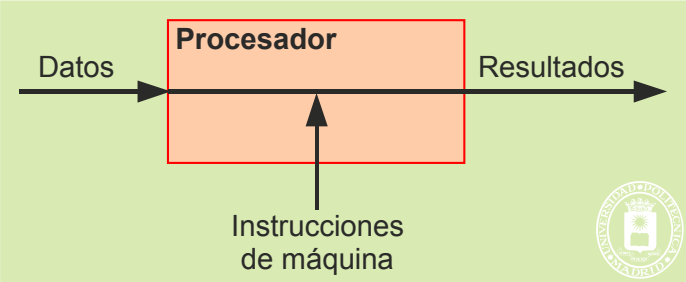


Figura 1.1 *Esquema de funcionamiento del computador.*

Para llevar a cabo la función de procesamiento, un computador con arquitectura von Neumann está compuesto por los cuatro componentes básicos representados en la figura 1.2.

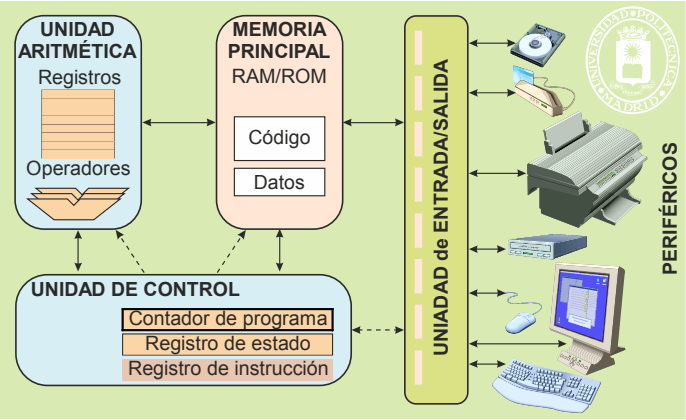


Figura 1.2 *Componentes básicos del computador con arquitectura von Neumann: Memoria principal, unidad aritmética, unidad de control y unidades de entrada/salida.*

Se denomina **procesador** o unidad central de proceso (**UCP**) al conjunto de la unidad aritmética y de la unidad de control. Actualmente, en un único circuito integrado se puede incluir varios procesadores (llamados núcleos o *cores*), además de la unidad de gestión de memoria que se describe en la sección "4.2.7 Concepto de memoria virtual y memoria real".

Desde el punto de vista de los sistemas operativos, nos interesa más profundizar en el funcionamiento interno del computador que en los componentes físicos que lo constituyen.

Memoria principal

La **memoria principal** se construye con memoria RAM (*Random Access Memory*) y memoria ROM (*Read Only Memory*). En ella han de residir los datos a procesar, el programa máquina a ejecutar y los resultados (aclaración 1.1). La memoria está formada por un conjunto de celdas idénticas. Mediante la información de dirección se selecciona de forma única la celda sobre la que se quiere realizar el acceso, pudiendo ser éste de lectura o de escritura. En los computadores actuales es muy frecuente que el direccionamiento se realice a nivel de byte, es decir, que las direcciones 0, 1, 2,... identifiquen los bytes 0, 1, 2,... de memoria. Sin embargo, como se muestra en la figura 1.3, el acceso se realiza generalmente sobre una **palabra** de varios bytes (típicamente de 4 o de 8 bytes) cuyo primer byte se sitúa en la dirección utilizada (que, por tanto, tiene que ser múltiplo de 4 o de 8).

Aclaración 1.1. Se denomina programa máquina (o código) al conjunto de instrucciones máquina que tiene por objeto que el computador realice una determinada función. Los programas escritos en cualquiera de los lenguajes de programación han de convertirse en programas máquina para poder ser ejecutados por el computador.

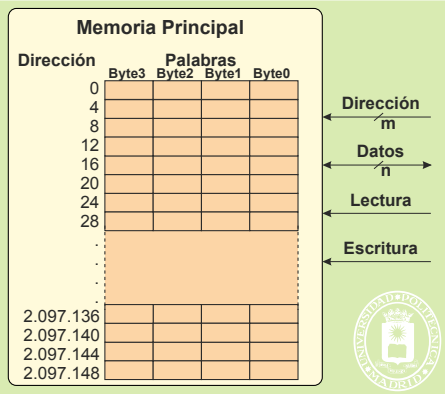


Figura 1.3 *La Unidad de memoria está compuesta por un conjunto de celdas iguales que se seleccionan mediante una dirección. Las señales de control de lectura y escritura determinan la operación a realizar.*

Unidad aritmético-lógica

La **unidad aritmético-lógica** permite realizar una serie de operaciones aritméticas y lógicas sobre uno o dos operandos. Como muestra la figura 1.4, los datos sobre los que opera esta unidad están almacenados en un conjunto de registros o bien provienen directamente de la memoria principal. Por su lado, los resultados también se almacenan en registros o en la memoria principal.

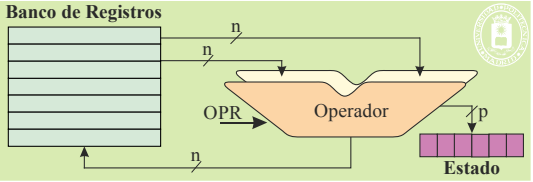


Figura 1.4 *Estructura de la unidad aritmético-lógica. Pueden observarse el banco de registros, los operadores y el registro de estado.*

Otro aspecto muy importante de la unidad aritmética es que, además del resultado de la operación solicitada, genera unos resultados adicionales que se cargan en el registro de estado del computador. Típicamente dichos resultados adicionales son los siguientes:

- Cero: Se pone a “1” si el resultado es cero.
- Signo: Se pone a “1” si el resultado es negativo.
- Acarreo: Se pone a “1” si el resultado tiene acarreo.
- Desbordamiento: Se pone a “1” si el resultado tiene desbordamiento.

La importancia de estos bits de estado es que las instrucciones de salto condicional se realizan sobre ellos, por lo que son los bits que sirven para tomar decisiones en los programas.

Unidad de control

La **unidad de control** es la que se encarga de hacer funcionar al conjunto, para lo cual realiza cíclicamente la siguiente secuencia:

- Lee de memoria la siguiente instrucción máquina que forma el programa.
- Interpreta la instrucción leída: aritmética, lógica, de salto, etc.
- Lee, si los hay, los datos de memoria referenciados por la instrucción.
- Ejecuta la instrucción.
- Almacena, si lo hay, el resultado de la instrucción.

La unidad de control tiene asociados una serie de registros, entre los que cabe destacar: el **contador de programa (PC,** *Program Counter***)**, que indica la dirección de la siguiente instrucción máquina a ejecutar; el **puntero de pila (SP,** *Stack Pointer***)**, que sirve para manejar cómodamente una pila en memoria principal; el **registro de instrucción** (RI), que permite almacenar —una vez leída de la memoria principal— la instrucción máquina a ejecutar, y el **registro de estado (RE)**, que almacena diversa información producida por la ejecución de alguna de las últimas instrucciones del programa (bits de estado aritméticos) e información sobre la forma en que ha de comportarse el computador (bits de interrupción, modo de ejecución, etc.).

Unidad de entrada/salida

Finalmente, la **unidad de entrada/salida** (**E/S**) se encarga de hacer la transferencia de información entre la memoria principal (o los registros generales) y los periféricos. La entrada/salida se puede hacer bajo el gobierno de la unidad de control (E/S programada) o de forma independiente (acceso directo a memoria o DMA), como se verá en la sección “1.6 Entrada/Salida”.

1.2. MODELO DE PROGRAMACIÓN DEL COMPUTADOR

El modelo de programación a bajo nivel de un computador, que también recibe el nombre de arquitectura ISA (*Instruction Set Architecture*) del computador, define los recursos y características que éste ofrece al programador de bajo nivel. Este modelo se caracteriza por los siguientes aspectos, que se muestran gráficamente en la figura 1.5.

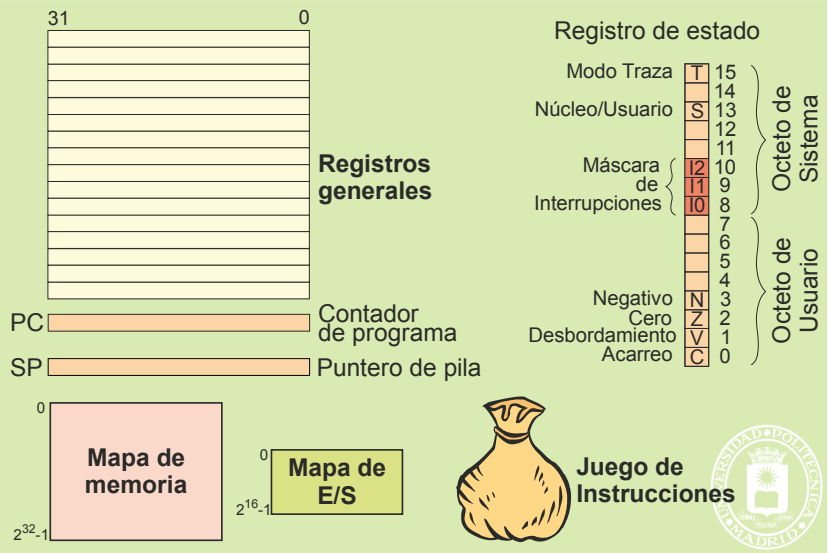


Figura 1.5 *Modelo de programación de un computador.*

- **Elementos de almacenamiento**. Son los elementos de almacenamiento del computador que son visibles a las instrucciones máquina. En esta categoría están incluidos los registros generales, el contador de programa, el o los punteros de pila, el registro de estado, la memoria principal y los registros de los controladores de E/S. La memoria principal se ubica en el mapa de memoria, mientras que los registros de E/S se ubican en el mapa de E/S. Véase aclaración 1.2.
- **Juego de instrucciones,** con sus correspondientes **modos de direccionamiento**. El juego de instrucciones máquina define las operaciones que es capaz de hacer el computador. Los modos de direccionamiento determinan la forma en que se especifica la localización de los operandos, es decir, los elementos de almacenamiento que intervienen en las instrucciones máquina.
- **Secuencia de funcionamiento**. Define el orden en que se van ejecutando las instrucciones máquina.
- **Modos de ejecución**. Un aspecto crucial de los computadores, que está presente en todos ellos menos en los modelos más simples, es que disponen de más de un modo de ejecución, concepto que se analiza en la sección siguiente y que es fundamental para el diseño de los sistemas operativos.

Aclaración 1.2. Por **mapa** se entiende todo el conjunto de posibles direcciones y, por tanto, de posibles palabras de memoria o de posibles registros de E/S que se pueden incluir en el computador. Es muy frecuente que los computadores incluyan el mapa de E/S dentro del mapa de memoria, en vez de incluir un mapa específico de E/S, reservando para ello un rango de direcciones del mapa de memoria (véase la sección "1.6.5 Buses y direccionamiento"). En este caso, se utilizan las mismas instrucciones máquina para acceder a la memoria principal y a los registros de E/S.

1.2.1. Modos de ejecución

La mayoría de los computadores de propósito general actuales presentan dos o más modos de ejecución. En el modo menos permisivo, generalmente llamado modo usuario, el computador ejecuta solamente un subconjunto de las instrucciones máquina, quedando prohibidas las demás, que se consideran "privilegiadas". Además, el acceso a determinados registros, o a partes de esos registros, y a determinadas zonas del mapa de memoria y de E/S también queda prohibido. En el modo más permisivo, denominado modo núcleo o modo privilegiado, el computador ejecuta todas sus instrucciones sin restricción, y permite el acceso a todos los registros y mapas de direcciones.

Se puede decir que el computador presenta más de un modelo de programación. Uno más restrictivo, que permite realizar un conjunto limitado de acciones, y otros más permisivos, que permiten realizar un mayor conjunto de acciones. Uno o varios bits del registro de estado establecen el modo en el que está ejecutando. Modificando estos bits se cambia de modo de ejecución.

Como veremos más adelante, los modos de ejecución se incluyen en los computadores para dar soporte al sistema operativo. Los programas de usuario, por razones de seguridad, no podrán realizar determinadas acciones, al ejecutar en modo usuario. Por su lado, el sistema operativo, que ejecuta en modo privilegiado, podrá ejecutar todo tipo de acciones. Cuando arranca el computador lo hace en modo privilegiado, puesto que lo primero que se debe hacer es cargar el sistema operativo, que debe ejecutar en modo privilegiado. El sistema operativo se encargará de poner en ejecución en modo usuario a los programas que ejecuten los usuarios.

Generalmente, el modo usuario no permite operaciones de E/S, ni modificar una gran parte del registro de estado, ni modificar los registros de soporte de gestión de memoria. La figura 1.6 muestra un ejemplo de dos modelos de programación de un computador. También es frecuente que el procesador incluya dos punteros de pila (SP y SP'), el SP para usarlo en modo usuario y el SP' para usarlo en modo privilegiado.

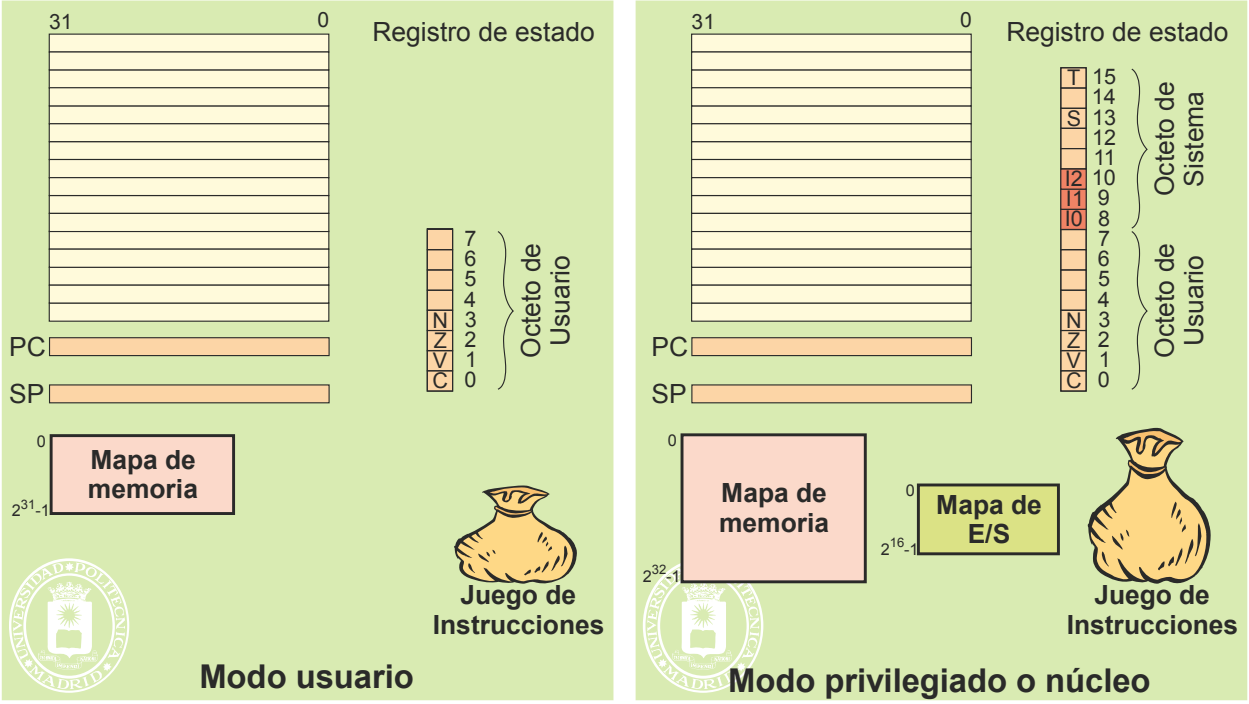


Figura 1.6 *Modelos de programación de usuario y de privilegiado.*

1.2.2. Secuencia de funcionamiento del procesador

La unidad de control del procesador es la que establece el funcionamiento del mismo. Este funcionamiento está basado en una secuencia sencilla, que se repite sin cesar y a muy alta velocidad (miles de millones de veces por segundo). Como muestra la figura 1.7, esta secuencia consiste en tres pasos: a) lectura de memoria principal de la instrucción máquina apuntada por el contador de programa, b) incremento del contador de programa —para que apunte a la siguiente instrucción máquina— y c) ejecución de la instrucción (que puede incluir la lectura de operandos en memoria o el almacenamiento de resultados en memoria). Esta secuencia tiene dos propiedades fundamentales: es lineal, es decir, ejecuta de forma consecutiva las instrucciones que están en direcciones consecutivas, y forma un bucle infinito. Esto significa que la unidad de control del procesador está continua e ininterrumpidamente realizando esta secuencia (advertencia 1.1).

Advertencia 1.1. Algunos procesadores tienen una instrucción de parada (p. ej.: `HALT`) que hace que la unidad de control se detenga hasta que llegue una interrupción. Sin embargo, esta instrucción es muy poco utilizada (salvo en equipos portátiles en los que interesa ahorrar batería), por lo que, a efectos prácticos, podemos considerar que la unidad de control no para nunca de realizar la secuencia de lectura de instrucción, incremento de PC y ejecución de la instrucción.

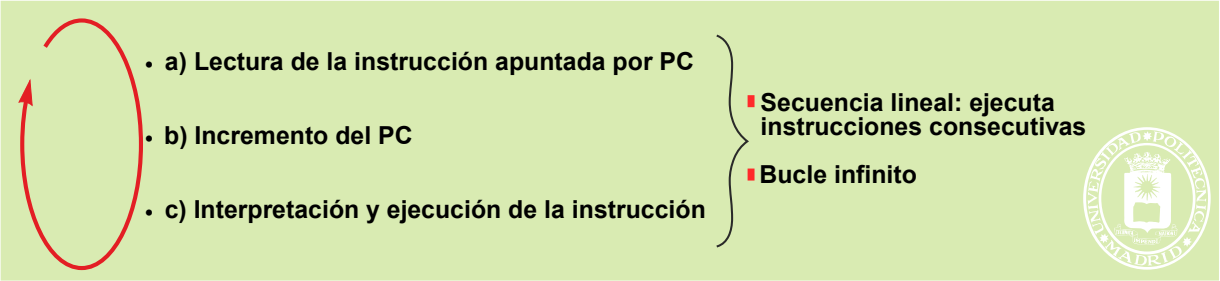


Figura 1.7 *Secuencia de ejecución del procesador.*

Podemos decir, por tanto, que lo único que sabe hacer el procesador es repetir a gran velocidad esta secuencia. Esto quiere decir que, para que realice algo útil, se ha de tener, adecuadamente cargado en memoria, un programa máquina con sus datos, y se ha de conseguir que el contador de programa apunte a la instrucción máquina inicial de dicho programa.

El esquema de ejecución lineal es muy limitado, por lo que se añaden unos mecanismos que permiten alterar esta ejecución lineal. Todos ellos se basan en algo muy simple: modifican el contenido del contador de programa, con lo que se consigue que se salte o bifurque a otra sección del programa o a otro programa (que, lógicamente, también ha de residir en memoria).

Los tres mecanismos básicos de ruptura de secuencia son los siguientes:

- Las instrucciones máquina de salto o bifurcación, que permiten que el programa rompa su secuencia lineal de ejecución, pasando a otra sección de sí mismo.
- Las interrupciones externas o internas, que hacen que la unidad de control modifique el valor del contador de programa, saltando a otro programa (que deberá ser el sistema operativo).
- Una instrucción máquina de llamada al sistema (p. ej.: `TRAP`, `INT` o `SC`), que produce un efecto similar a la interrupción, haciendo que se salte a otro programa (que deberá ser el sistema operativo).

Si desde el punto de vista de la programación el interés se centra en las instrucciones de salto, y, en especial en las de salto a procedimiento y retorno de procedimiento, desde el punto de vista de los sistemas operativos son mucho más importantes las interrupciones y las instrucciones máquina de llamada al sistema. Por tanto, centraremos nuestro interés en resaltar los aspectos fundamentales de estas dos últimas.

1.2.3. Registros de control y estado

Como se ha indicado anteriormente, la unidad de control tiene asociada una serie de registros que denominamos de control y estado. Estos registros dependen de la arquitectura del procesador. Muchos de ellos se refieren a aspectos que se analizarán a lo largo del texto, por lo que no se intentará explicar aquí su función. Entre los más importantes se pueden encontrar los siguientes:

- Contador de programa PC. Contiene la dirección de la siguiente instrucción máquina.
- Puntero de pila SP. Contiene la dirección de la cima de la pila. En algunos procesadores existen dos punteros de pila: uno para la pila del sistema operativo y otra para la del usuario.
- Registro de instrucción RI. Contienen la instrucción en curso de ejecución.
- Registro de estado, que contiene, entre otros, los bits siguientes:
  - Bits de estado aritméticos como: Signo, Acarreo, Cero y Desbordamiento.
  - Bits de modo de ejecución. Indican el modo en el que ejecuta el procesador.
  - Bits de control de interrupciones. Establecen las interrupciones que se pueden aceptar.
- Registros de gestión de memoria, como pueden ser los registros de protección de memoria o el registro identificador del espacio de direccionamiento (véase la sección “1.7.2 Mecanismos de protección de memoria”).

Algunos de estos registros son **visibles** en el modo de ejecución de usuario, como el PC, el SP y parte del estado, pero otros no lo son, como los de gestión de memoria.

Al contenido de todos los registros del procesador en un instante determinado le denominamos **estado del procesador**, término que utilizaremos profusamente a lo largo del libro. Un subconjunto del estado del procesador lo constituye el **estado visible** del procesador, formado por el conjunto de los registros visibles en modo usuario.

1.3. INTERRUPCIONES

Una interrupción se solicita activando una señal que llega a la unidad de control. El agente generador o solicitante de la interrupción ha de activar la mencionada señal cuando necesite que se le atienda, es decir, que se ejecute un programa que le atienda.

Ante la solicitud de una interrupción, siempre y cuando esté habilitada ese tipo de interrupción, la unidad de control realiza un **ciclo de aceptación de interrupción**. Este ciclo se lleva a cabo en cuanto termina la ejecución de la instrucción máquina que se esté ejecutando, y los pasos que realiza la unidad de control son los siguientes:

- **Salva algunos registros** del procesador, como son el de estado y el contador de programa. Normalmente utiliza para ello la **pila de sistema**, gestionada por el puntero de pila SP'.
- **Eleva el modo de ejecución** del procesador, pasándolo a núcleo.
- **Carga un nuevo valor en el contador de programa**, por lo que se pasa a ejecutar otro programa, que, generalmente, será el sistema operativo.
- En muchos procesadores **inhibe** las interrupciones (véase más adelante “Niveles de Interrupción”).

La figura 1.8 muestra la interrupción vectorizada, solución usualmente utilizada para determinar la dirección de salto. Se puede observar que el agente que interrumpe suministra el llamado **vector de interrupción** que determina la dirección de comienzo del programa que desea que le atienda (programa que se suele denominar rutina de tratamiento de interrupción). La unidad de control, utilizando un direccionamiento indirecto, toma la mencionada dirección de una tabla de interrupciones IDT (*Interrupt Descriptor Table*) y la carga en el contador de programa. El resultado de esta carga es que la siguiente instrucción máquina ejecutada es la primera del mencionado programa de tratamiento de interrupción.

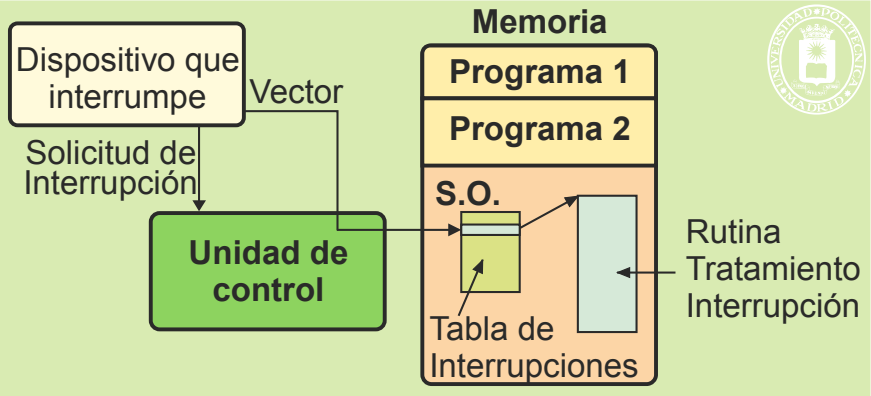


Figura 1.8 *Acceso a la rutina de tratamiento de la interrupción.*

Observe que se han incluido como parte del sistema operativo tanto la tabla IDT como la rutina de tratamiento de la interrupción. Esto debe ser así por seguridad, ya que la rutina de tratamiento de interrupción ejecuta en modo privilegiado. En caso contrario, un programa de usuario ejecutaría sin limitación ninguna, por lo que podría acceder a los datos y programas de otros usuarios. Como se verá más adelante, la seguridad es una de las funciones primordiales del sistema operativo.

Se dice que la interrupción es **síncrona** cuando es consecuencia directa de las instrucciones máquina que se están ejecutando. En el resto de los casos se dice que es **asíncrona**. Las interrupciones se pueden generar por diversas causas, que clasificaremos de la siguiente forma:

- **Excepciones *hardware* síncronas**. Hay determinadas causas que hacen que un programa presente una incidencia en su ejecución, por lo que se generará una interrupción, para que el sistema operativo entre a ejecutar y decida lo que debe hacerse. Dichas causas se pueden estructurar en las tres clases siguientes:
  - Problemas de ejecución:
    - Operación inválida en la unidad aritmética.
    - División por cero.
    - Operando no normalizado.
    - Desbordamiento en el resultado, siendo demasiado grande o demasiado pequeño.
    - Resultado inexacto en la unidad aritmética.
    - Dispositivo no existente (p. ej.: no existe coprocesador).
    - Región de memoria inválida.
    - Región de memoria privilegiada en modo de ejecución usuario.
    - Desbordamiento de la pila.
    - Violación de los límites de memoria asignada.
    - Error de alineación en acceso a memoria.
    - Código de operación máquina inválido.
    - Código de operación máquina privilegiado en modo de ejecución usuario.
  - Depuración:
    - Punto de ruptura.
  - Fallo de página.

  Todas ellas son producidas directa o indirectamente por el programa en ejecución, por lo que decimos que se trata de interrupciones **síncronas** (advertencia 1.2).
- **Excepciones *hardware* asíncronas**. Se trata de interrupciones **asíncronas** producidas por un error en el *hardware*. En muchos textos se denominan simplemente excepciones *hardware*. Ejemplos son los siguientes:
  - Error de paridad en bus.
  - Error de paridad en memoria.
  - Fallo de alimentación.
  - Límite de temperatura excedido.

  En las excepciones *hardware*, tanto síncronas como asíncronas, será el módulo del computador que produce la excepción el que genere el vector de interrupción. Además, dicho módulo suele cargar en un registro o en la pila un código que especifica el tipo de problema encontrado.
- **Interrupciones externas**. Se trata de interrupciones **asíncronas** producidas por elementos externos al procesador como son: a) el reloj, que se analizará en detalle en la sección siguiente, b) los controladores de los dispositivos de E/S, que necesitan interrumpir para indicar que han terminado una operación o conjunto de ellas, o que necesitan atención, y c) otros procesadores.
- Instrucciones máquina de **llamada al sistema** (p. ej.: `TRAP`, `INT`, `SYSENTER` o `SC`). Estas instrucciones permiten que un programa genere una interrupción de tipo síncrono. Como veremos más adelante,

estas instrucciones se incluyen para que los programas de usuario puedan solicitar los servicios del sistema operativo.

Advertencia 1.2. Las **excepciones *hardware* síncronas** se denominan muy frecuentemente **excepciones *software***, pero en este texto reservamos dicho nombre para el concepto de excepción soportado por el sistema operativo (véase sección "3.5 Señales y excepciones", página 83).

Como complemento al mecanismo de aceptación de interrupción, los procesadores incluyen una instrucción máquina para retornar desde la rutina de tratamiento de interrupción (p. ej.: `RETI`). El efecto de esta instrucción es restituir los registros de estado y PC, desde el lugar en que fueron salvados al aceptarse la interrupción (p. ej.: desde la pila del sistema).

Niveles de interrupción

Como muestra la figura 1.9, los procesadores suelen incluir varias líneas de solicitud de interrupción, cada una de las cuales puede tener asignada una determinada prioridad. En caso de activarse al tiempo varias de estas líneas, se tratará la de mayor prioridad, quedando las demás a la espera de ser atendidas. Las más prioritarias suelen ser las excepciones *hardware* asíncronas, seguidas por las excepciones *hardware* síncronas (o de programa), las interrupciones externas y las de llamada al sistema o TRAP.

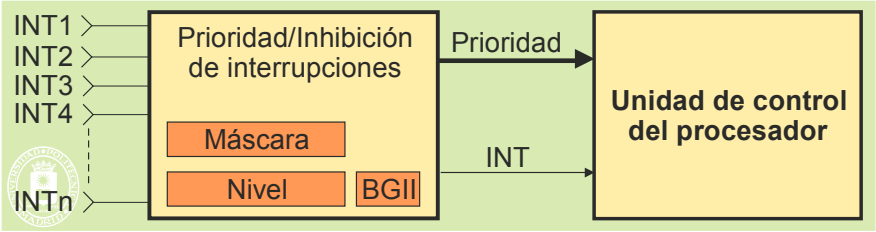


Figura 1.9 *Mecanismo de inhibición/prioridad de interrupción.*

Además, el procesador suele incluir un mecanismo de **inhibición** selectiva que permite detener todas o determinadas líneas de interrupción. Este mecanismo, que es muy específico de cada máquina, puede estar basado en los dos registros de máscara y de nivel, y en el biestable general de inhibición de interrupción (BGII). El comportamiento de estos mecanismos es el siguiente. Mientras esté activo BGII no se admitirá ninguna interrupción. El registro de **nivel** inhibe todas las interrupciones con prioridad menor o igual que el valor que contenga. Finalmente, el registro de **máscara** tiene un bit por línea de interrupción, por lo que permite inhibir de forma selectiva cualquiera de las líneas de interrupción.

Las interrupciones de las líneas inhibidas no son atendidas hasta que, al modificarse los valores de máscara, nivel o BGII, dejen de estar inhibidas. Dependiendo del módulo que las genera y del *hardware* de interrupción, se pueden encolar o se pueden llegar a perder algunas de las interrupciones inhibidas. Estos valores de máscara, nivel y BGII deben incluirse en la parte del registro de estado que solamente es modificable en modo privilegiado, por lo que su modificación queda restringida al sistema operativo. La unidad de control, al aceptar una interrupción, suele modificar determinados valores de los registros de inhibición para facilitar el tratamiento de las mismas. Las alternativas más empleadas son dos: inhibir todas las interrupciones o inhibir las interrupciones que tengan prioridad igual o menor que la aceptada.

Tratamiento de interrupciones

En el ciclo de aceptación de una interrupción se salvan algunos registros, pero un correcto tratamiento de las interrupciones exige preservar los valores del resto de los registros del programa interrumpido. De esta forma, se podrán restituir posteriormente dichos valores, para continuar con la ejecución del programa como si no hubiera pasado nada. Téngase en cuenta que el nuevo programa, que pone en ejecución la interrupción, cargará nuevos valores en los registros del procesador. También es necesario mantener la información que tenga el programa en memoria, para lo cual basta con no modificar la zona de memoria asignada al mismo.

Además, la rutina de tratamiento de interrupciones deberá, en cuanto ello sea posible, restituir los valores de los registros de inhibición de interrupciones para admitir el mayor número posible de interrupciones.

La instrucción máquina de retorno de interrupción RETI realiza una función inversa a la del ciclo de aceptación de la interrupción. Típicamente, restituye el valor del registro de estado E y el del contador de programa PC. En muchos casos, la instrucción RETI producirá, de forma indirecta, el paso del procesador a modo usuario. En efecto, supóngase que está ejecutando el programa A en modo usuario y que llega una interrupción. El ciclo de aceptación salva el registro de estado (que tendrá un valor Ea) y el contador de programa (que tendrá un valor PCa). Como el bit que especifica el modo de funcionamiento del procesador se encuentra en el registro de estado, Ea tiene activo el modo usuario. Seguidamente, el ciclo de aceptación cambia el registro de estado activando el modo privilegiado. Cuando más tarde se restituya al registro de estado el valor salvado Ea, que tiene activo el modo usuario, se pone el procesador en este modo. Por otro lado, al restituir el valor de contador de programa, se consigue que el procesador siga ejecutando a continuación del punto en el que el programa A fue interrumpido.

En las sección "3.10 Tratamiento de interrupciones" se detalla el tratamiento de las interrupciones realizado por el sistema operativo, dando soluciones al **anidamiento** de las mismas, fenómeno que se produce cuando se acepta una interrupción sin haberse completado el tratamiento de la anterior.

1.4. EL RELOJ

El término reloj se aplica a los computadores con tres acepciones diferentes, si bien relacionadas, como se muestra en la figura 1.10. Estas tres acepciones son las siguientes:

- Señal que gobierna el ritmo de ejecución de las instrucciones máquina (CLK).
- Generador de interrupciones periódicas o temporizador.
- Contador de fecha y hora, o reloj de tiempo real RTC (*Real Time Clock*).

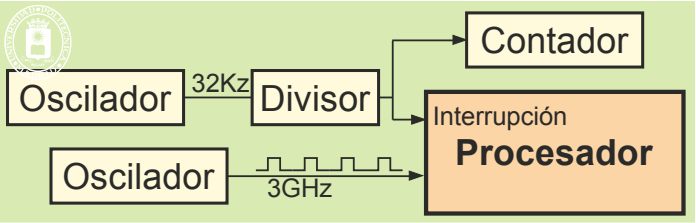


Figura 1.10 *Relojes del computador.*

El oscilador que gobierna las fases de ejecución de las instrucciones máquina se denomina reloj. Cuando se dice que un microprocesador es de 5 GHz, se está especificando que el oscilador que gobierna su ritmo de funcionamiento interno produce una onda cuadrada con una frecuencia de 5 GHz.

La señal producida por el oscilador anterior, o por otro oscilador, se divide mediante un divisor de frecuencia para generar una interrupción externa cada cierto intervalo de tiempo. Estas interrupciones, que se están produciendo constantemente, se denominan **interrupciones de reloj** o **tics**, dando lugar al segundo concepto de reloj. El objetivo de estas interrupciones es, como veremos más adelante, hacer que el sistema operativo entre a ejecutar de forma periódica. De esta manera, podrá evitar que un programa monopolice el uso del computador y podrá hacer que entren a ejecutarse programas en determinados instantes de tiempo. Estas interrupciones se producen con un periodo de entre 1 a 100 ms; por ejemplo, en la arquitectura PC clásica para MS-DOS el temporizador 8254 produce un tic cada 54,926138 ms.

La tercera acepción de reloj, denominada RTC, se aplica a un contador que permite conocer **la fecha y la hora**. Tomando como referencia un determinado instante (p. ej.: 0 horas del 1 de enero de 1990 (advertencia 1.3)) se puede calcular la hora y fecha en que estamos. Observe que este concepto de reloj es similar al del reloj electrónico de pulsera. En los computadores actuales esta cuenta se hace mediante un circuito dedicado que, además, está permanentemente alimentado, de forma que aunque se apague el computador se siga manteniendo el reloj en hora. En sistemas más antiguos el sistema operativo se encargaba de hacer esta cuenta, por lo que había que introducir la fecha y la hora cada vez que se arrancaba el computador. Por ejemplo, la arquitectura clásica PC incluye un RTC alimentado por una pequeña pila y gobernado por un cristal de 32.768 kHz, que almacena la hora con resolución de segundo.

Advertencia 1.3. En el caso de UNIX se toma como referencia las 0 horas del 1 de enero de 1970. Si se utiliza una palabra de 32 bits y se almacena la hora con resolución de segundo, el mayor número que se puede almacenar es el 2.147.483.647, que se corresponde a las 3h 14m y 7s de enero de 2038. Esto significa que, a partir de ese instante, el contador tomaría el valor 0 y la fecha volvería a ser el 1 de enero de 1970.

En algunos sistemas de tiempo real, o cuando se trata de monitorizar el sistema, es necesario tener una medida muy precisa del tiempo. En la arquitectura PC con el RTC se tiene una resolución de 1 segundo y de decenas de ms con los tics. Estos valores no son a veces suficientemente pequeños. Por ello, en el procesador Pentium se incluyó un registro de 64 bits que cuenta ciclos de la señal CLK y que se denomina TSC (*Time-Stamp Counter*). Este registro puede ser leído por las aplicaciones y suministra una gran resolución, pero hay que tener en cuenta que mide ciclos del oscilador y no tiempo. Un incremento de 4.000 en el TSC supone 2 µs en un procesador de 2 GHz, mientras que supone 1µs en uno de 4 GHz.

1.5. JERARQUÍA DE MEMORIA

Dado que la memoria de alta velocidad tiene un precio elevado y un tamaño reducido, la memoria del computador se organiza en forma de una jerarquía, como la mostrada en la figura 1.11. En esta jerarquía se utilizan memorias permanentes de alta capacidad y baja velocidad, como son los discos, para almacenamiento permanente de la información, mientras que se emplean memorias de semiconductores de un tamaño relativamente reducido, pero de alta velocidad, para almacenar la información que se está utilizando en un momento determinado.

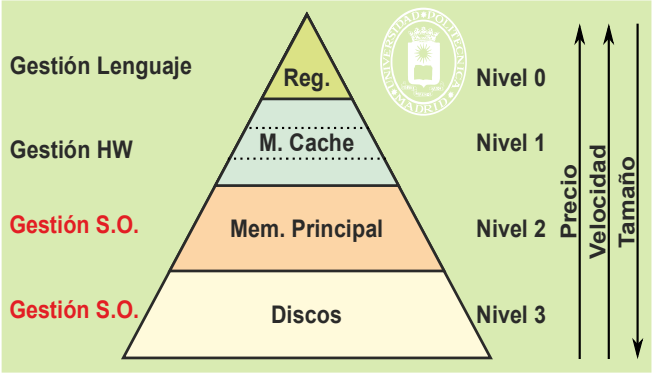


Figura 1.11 *Jerarquía de memoria*

El funcionamiento de la jerarquía de memoria exige hacer copias de información de los niveles más lentos a los niveles más rápidos, en los cuales es utilizada (p. ej.: cuando se desea ejecutar un programa hay que leer de disco el fichero ejecutable y almacenarlo en memoria principal, de donde se irá leyendo y ejecutando instrucción a instrucción por la unidad de control). Inversamente, cuando se modifica o crea la información en un nivel rápido, y se desea su permanencia, hay que enviarla al nivel inferior, por ejemplo, al nivel de disco o cinta.

Para entender bien el objetivo y funcionamiento de la jerarquía de memoria, es muy importante tener siempre presente tanto el orden de magnitud de los tiempos de acceso de cada tecnología de memoria como los tamaños típicos empleados en cada nivel de la jerarquía. La tabla 1.1 presenta algunos valores típicos.

Tabla 1.1 *Valores típicos de la jerarquía de memoria.*

Nivel de memoria	Capacidad	Tiempo de acceso	Tipo de acceso
Registros	64 a 1024 bytes	0,25 a 0,5 ns	Palabra
cache de memoria principal	8 KiB a 8 MiB	0,5 a 20 ns	Palabra
Memoria principal	128 MiB a 64 GiB	60 a 200 ns	Palabra
Disco electrónico SSD	128 GiB a 1 TiB	50 µs (lectura)	Sector
Disco magnéticos	256 GiB a 4 TiB	5 a 30 ms	Sector

La gestión de la jerarquía de memoria es compleja, puesto que ha de tener en cuenta las copias de información que están en cada nivel y ha de realizar las transferencias de información a niveles más rápidos, así como las actualizaciones hacia los niveles permanentes. Una parte muy importante de esta gestión corre a cargo del sistema operativo, aunque, para hacerla correctamente, requiere de la ayuda del *hardware*. Por ello, se revisan en esta sección los conceptos más importantes de la jerarquía de memoria, para analizar más adelante su aplicación a la memoria virtual, de especial interés para nosotros, dado que su gestión la realiza el sistema operativo.

1.5.1. Memoria cache y memoria virtual

La explotación correcta de la jerarquía de memoria exige tener, en cada momento, la información adecuada en el nivel adecuado. Para ello, la información ha de moverse de nivel, esto es, ha de migrar de un nivel a otro. Esta migración puede ser **bajo demanda explícita** o puede ser **automática**. La primera alternativa exige que el programa solicite explícitamente el movimiento de la información, como ocurre, por ejemplo, con un programa editor, que va solicitando la parte del fichero que está editando el usuario en cada momento. La segunda alternativa consiste en hacer la migración transparente al programa, es decir, sin que éste tenga que ser consciente de que se produce. **La migración automática se utiliza en las memorias cache y en la memoria virtual**.

Memoria cache

El término cache deriva del verbo francés *cacher*, que significa ocultar, esconder. Con este término se quiere reflejar que la memoria cache no es visible al programa máquina, puesto que no está ubicada en el mapa de memoria. Se trata de una memoria de apoyo a la memoria principal que sirve para acelerar los accesos. La memoria cache alberga información recientemente utilizada, con la esperanza de que vuelva a ser empleada en un futuro próximo. Los aciertos sobre cache permiten atender al procesador más rápidamente que accediendo a la memoria principal.

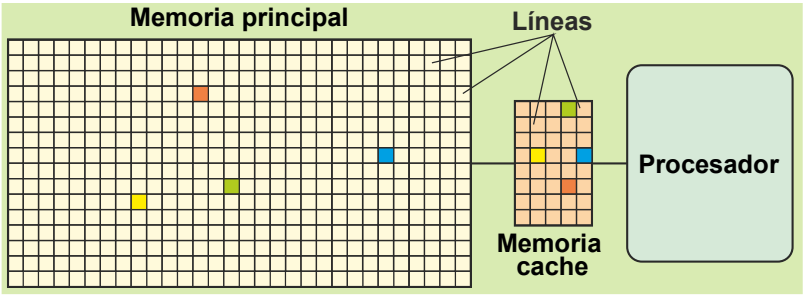


Figura 1.12 *La memoria cache se interpone entre el procesador y la memoria principal para acelerar los accesos a esta última. La memoria cache almacena la información utilizada más recientemente con la esperanza de volver a utilizarla en un futuro próximo.*

El bloque de información que se migra entre la memoria principal y la cache se denomina **línea** y está formado por varias palabras (valores típicos de línea son de 32 a 128 bytes). Toda la gestión de la cache necesaria para migrar líneas y para direccionar la información dentro de la cache se realiza por ***hardware***, debido a la gran velocidad a la que debe funcionar. El tiempo de tratamiento de un fallo tiene que ser del orden del tiempo de acceso a la memoria lenta, es decir, de los 60 a 200 ns que se tarda en acceder a la memoria principal, puesto que el procesador se queda esperando a poder realizar el acceso solicitado.

En la actualidad, debido a la gran diferencia de velocidad entre los procesadores y las memorias principales, se utilizan tres niveles de cache, que se incluyen en el mismo chip que los procesadores.

Aunque la memoria cache es una memoria oculta, no nos podemos olvidar de su existencia, puesto que repercute fuertemente en las prestaciones de los sistemas. Plantear adecuadamente un problema para que genere pocos fallos de cache puede disminuir espectacularmente su tiempo de ejecución.

La memoria virtual versus memoria real

Una máquina con **memoria real** es una máquina convencional que **solamente utiliza memoria principal para soportar el mapa de memoria**. Por el contrario, una máquina con **memoria virtual soporta su mapa de memoria mediante dos niveles de la jerarquía de memoria: la memoria principal y una memoria de respaldo,** que suele ser una parte del disco que llamamos zona de intercambio o *swap*.

La memoria virtual es un mecanismo de migración automática, por lo que exige una gestión automática de la parte de la jerarquía de memoria formada por la memoria principal y el disco. Esta gestión la realiza el sistema operativo con ayuda de una unidad *hardware* de gestión de memoria, llamada **MMU** (*Memory Management Unit*) (ver sección "4.2.9 Unidad de gestión de memoria (MMU)"). Como muestra la figura 1.13, esta gestión incluye toda la memoria principal y la parte del disco que sirve de respaldo a la memoria virtual.

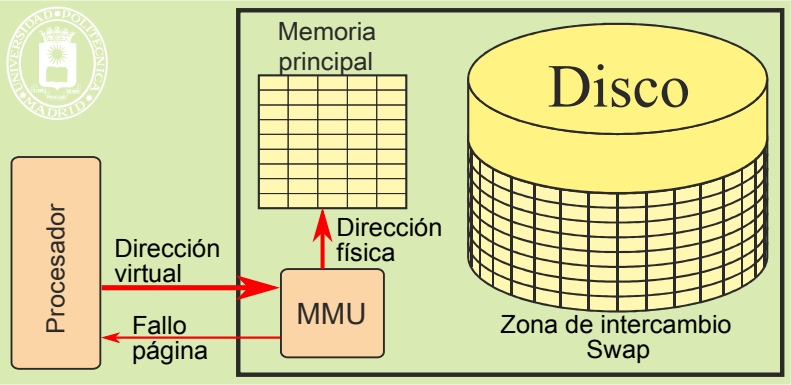


Figura 1.13 *Fundamento de la memoria virtual.*

Tanto el mapa de memoria (que denominaremos espacio virtual) como la memoria principal y el *swap* se dividen en páginas. Se denominan **páginas virtuales** a las páginas del mapa de memoria, **páginas de intercambio** a las páginas de *swap* y **marcos de página** a los espacios en los que se divide la memoria principal. Normalmente, cada marco de página puede albergar una página virtual cualquiera, sin ninguna restricción de direccionamiento.

Existe una estructura de información llamada **tabla de páginas** que contiene la ubicación de cada página virtual, es decir, el marco de página o la página de intercambio donde se encuentra. La MMU utiliza la tabla de páginas para traducir la dirección generada por el procesador a la dirección de memoria principal correspondiente. Si la dirección corresponde a una página que no está en memoria principal, la MMU genera una excepción de **fallo de página**, para que el sistema operativo se encargue de traer la página del disco a un marco de página. Será, por tanto, responsabilidad del sistema operativo la creación y mantenimiento de la tabla de páginas para que refleje en cada instante la ubicación real de las páginas.

La tabla de páginas puede tener un tamaño bastante grande y muy variable según las aplicaciones que se ejecuten, por lo que reside en memoria principal. Para acelerar el acceso a la información de la tabla de páginas la MMU utiliza la TLB (*Translation Look-aside buffer*). La TLB es una memoria asociativa muy rápida que almacena las parejas página-marco de las páginas a las que se ha accedido recientemente, de forma que en la mayoría de los casos la MMU no accederá a la memoria principal, al encontrar la información de traducción en la TLB. Cuando la MMU no encuentra la información de traducción en la TLB (lo que se denomina fallo de la TLB) debe acceder a memoria principal y sustituir una de las parejas página-marco de la misma por la nueva información. De esta forma, se va renovando la información de la TLB con los valores de interés en cada momento.

Operación en modo real

Como se ha indicado, el sistema operativo es el encargado de crear y mantener las tablas de páginas para que la MMU pueda hacer su trabajo de traducción. Ahora bien, cuando arranca el computador no existen tablas de páginas, por lo que la MMU no puede realizar su función. Para evitar este problema la MMU incorpora un modo de funcionamiento denominado real, en el cual se limita a presentar la dirección recibida del procesador a la memoria principal. En modo real el computador funciona, por tanto, como una máquina carente de memoria virtual.

1.6. ENTRADA/SALIDA

Los mecanismos de entrada/salida (E/S) del computador tienen por objeto el intercambio de información entre los periféricos y la memoria o los registros del procesador. En esta sección se presentan los dos aspectos de la E/S que revisten mayor relevancia de cara al sistema operativo: la concurrencia de la E/S con el procesador y el direccionamiento.

1.6.1. Características de la entrada/salida

En el intercambio de información con los periféricos se suelen transferir grandes porciones de información, encapsuladas en lo que se denomina **operación de entrada/salida** (p.e. imprimir un documento). Como muestra la figura 1.14, las operaciones de entrada/salida se descomponen en transferencias de bloques. A su vez cada transferencia de bloque requiere tantas transferencias elementales como palabras tenga el bloque. Las operaciones de E/S se realizan por *software*, siendo el sistema operativo el responsable de realizar las mismas dado que es el único que debe ejecutar en modo núcleo y, por tanto, el único que puede dialogar con los periféricos. Actualmente, las transferencias de bloque se realizan por *hardware* mediante la técnica del DMA (véase la sección "1.6.4 E/S y concurrencia")

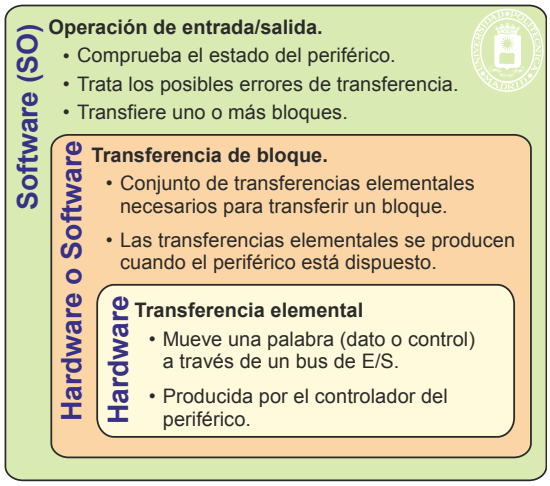


Figura 1.14 *Una operación de E/S se suele descomponer en una serie de transferencias de bloques (e.g. sectores del disco). A su vez cada bloque requiere una transferencia elemental por cada palabra de información transferida.*

Otras características de la E/S son las siguientes:

- Muchos periféricos tienen un tamaño de información privilegiado, que denominaremos bloque. Por ejemplo, el disco magnético funciona en bloques denominados sectores, que tienen un tamaño típico de 512 B.
- Los periféricos tienen unas velocidades de transmisión de información muy variable, que puede ser de unos pocos B/s (bytes/segundo), lo que ocurre en un teclado, hasta varios cientos de MiB/s (megabytes/segundo), lo que ocurre en un disco a un adaptador de red. En términos generales los periféricos son mucho más lentos que los procesadores.
- El ancho de palabra de los periféricos suele ser de un byte, frente a los 32 o 64 bits de los procesadores.
- Los periféricos suelen necesitar un **control permanente**. Por ejemplo, hay que saber si la impresora está encendida o apagada, si tiene papel, si el lector de CD-ROM tiene un disco o no, si el módem tiene línea, etc.
- Los periféricos tienen su ritmo propio de funcionamiento, por ejemplo, producen o aceptan datos e información de control a su propia velocidad, no a la que el computador podría hacerlo, por lo que los programas que tratan con ellos han de adaptarse a dicho ritmo. Decimos que es necesario **sincronizar el programa con el periférico**, de forma que el programa envíe o lea la información de control y los datos del periférico cuando éste esté disponible.

1.6.2. Periféricos

Los periféricos son componentes que sirven para introducir o extraer información del procesador. Por su función, los periféricos se pueden clasificar en las siguientes cuatro grandes categorías, pudiendo algún periférico pertenecer a más de una de ellas:

- Periféricos de **captura** de información. Tales como teclados, ratones, cámaras de vídeo, escáneres o convertidores analógico/digital, que sirven para introducir información en el computador.
- Periféricos de **presentación** de información. Tales como pantallas, impresoras, trazadoras, altavoces o convertidores digital/analógico, que sirven para mostrar información del computador.
- Periféricos de **almacenamiento** de información. Tales como discos, memorias USB, DVD o cintas, que permiten grabar y recuperar la información.

- Periféricos de **transmisión** de información. Tales como adaptadores Ethernet, adaptadores WIFI o módems ADSL, que permiten transmitir información entre computadores.

La figura 1.15 muestra el esquema general de un periférico, compuesto por el dispositivo y su controlador. Este último tiene una serie de registros incluidos en el mapa de E/S del computador, por lo que se puede acceder a ellos mediante las correspondientes instrucciones máquina. En máquinas con protección, solamente se puede acceder a estos registros en modo privilegiado.

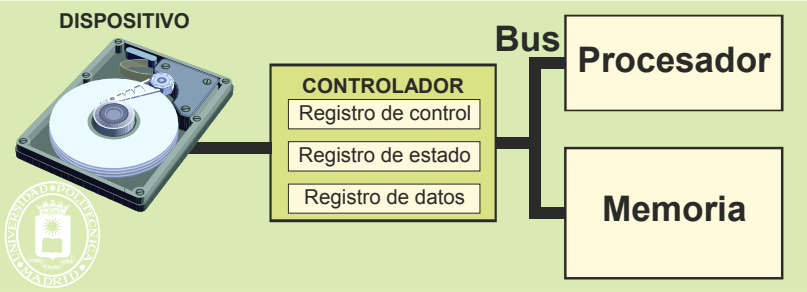


Figura 1.15 *Modelo de periférico.*

El registro de datos sirve para el intercambio de datos. En él el controlador irá cargando los datos leídos y de él irá extrayendo los datos para su escritura en el periférico.

Un bit del registro de estado sirve para indicar que el controlador puede transferir una palabra. En las operaciones de lectura esto significa que ha cargado en el registro de datos un nuevo valor, mientras que en las de escritura significa que necesita un nuevo dato. Otros bits de este registro sirven para que el controlador indique los problemas que ha encontrado en la ejecución de la última operación de entrada/salida y el modo en el que está operando.

El registro de control sirve para indicar al controlador las operaciones que ha de realizar. Los distintos bits de este registro indican distintas acciones que ha de realizar el periférico.

Conexión de los controladores de periférico

Los computadores disponen de un bus con varias ranuras, en el que se pueden enchufar los controladores de periféricos, tal y como se puede observar en la figura 1.16. Dicho bus incluye conexiones para datos, direcciones, señales de control y alimentación eléctrica del controlador. Las conexiones de direcciones se corresponden con el mapa de E/S del procesador (ya sea éste un mapa independiente de E/S o una parte del mapa de memoria utilizado para E/S).

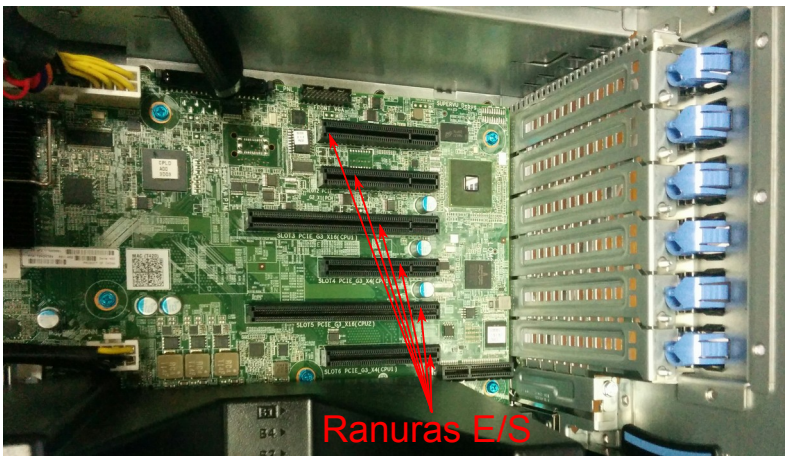


Figura 1.16 *Ranuras de E/S en un PC.*

Cada registro del controlador ha de estar asociado a una dirección de dicho mapa, por lo que el controlador tiene un decodificador que activa el registro cuando su dirección es enviada por el bus. Dicho decodificador ha de conocer, por tanto, la dirección asignada al registro.

Para permitir flexibilidad a la hora de conectar controladores de periférico, los registros de éstos no tienen direcciones fijas sino configurables. Inicialmente dicha configuración se hacía mediante pequeños puentes o jumperes, de forma que la dirección quedaba cableada en el controlador. Lo mismo ocurría con determinadas señales de control como las que determinan el nivel de interrupción y el canal DMA utilizado por el controlador.

Sin embargo, en la actualidad, los controladores de periférico utilizan la tecnología ***plug and play***. Para ello, incluyen una memoria con información del controlador, como tipo de dispositivo y fabricante, y una memoria flash en la que se puede grabar la mencionada información de configuración. De esta forma, el sistema operativo puede conocer el tipo de dispositivo y configurar directamente el controlador sin que el usuario tenga que preocuparse de asignar direcciones o canales libres.

Dispositivos de bloques y caracteres

Los sistemas operativos tipo UNIX organizan los periféricos en las tres grandes categorías de bloques, caracteres y red, que son tratados de forma distinta.

Los dispositivos de bloques se caracterizan por estar divididos en unidades de almacenamiento (bloques) numeradas y direccionables. El acceso se realiza como mínimo a una unidad de almacenamiento. El ejemplo más representativo es el disco magnético que está dividido en sectores y que permite lecturas o escrituras a uno o varios sectores consecutivos. Existen otros dispositivos de bloques como las cintas magnéticas, los DVD y los CD. Todos ellos se caracterizan por tener un tiempo de acceso importante comparado con el tiempo de transferencia de una palabra, por lo que interesa amortizar este tiempo de acceso transfiriendo bastantes palabras.

Otros dispositivos como el teclado o ratón entran en la categoría de caracteres. Son una fuente o sumidero de bytes no direccionables. Muchos de estos dispositivos se caracterizan por ser lentos.

La categoría de red está formada por los controladores de red como son los adaptadores Ethernet, adaptadores Wifi y módem ADSL, que se caracterizan por llegar a tener una alta velocidad de transferencia de datos.

1.6.3. Periféricos más importantes

Dada su importancia, se describen seguidamente el disco magnético, el terminal y el controlador de red.

El disco

El disco es el periférico más importante, puesto que sirve de espacio de intercambio a la memoria virtual y sirve de almacenamiento permanente para los programas y los datos, encargándose el sistema operativo de la gestión de este tipo de dispositivo. En la actualidad existen dos tipos de discos, los **discos magnéticos o duros** y los **discos de estado sólido** o **SSD** (*Solid-State Drive*).

El disco magnético se compone de uno o varios platos recubiertos de material magnético, en los que se graba la información en pistas circulares, de un brazo que soporta los transductores de lectura y escritura, que se puede posicionar encima de la pista deseada, y de una electrónica que permite posicionar el brazo y leer y escribir en el soporte magnético.

Para entender la forma en que el sistema operativo trata los discos magnéticos es necesario conocer las características de los mismos, entre las que destacaremos tres: organización de la información, tiempo de acceso y velocidad de transferencia.

La **organización de la información** del disco se realiza en contenedores de tamaño fijo denominados **sectores** (el tamaño típico del sector es de 512 bytes). Como muestra la figura 1.17, el disco se divide en pistas circulares que, a su vez, se dividen en sectores. Las operaciones de lectura y escritura se realizan a nivel de sector, es decir, no se puede escribir o leer una palabra o byte individual: hay que escribir o leer de golpe uno o varios sectores contiguos.

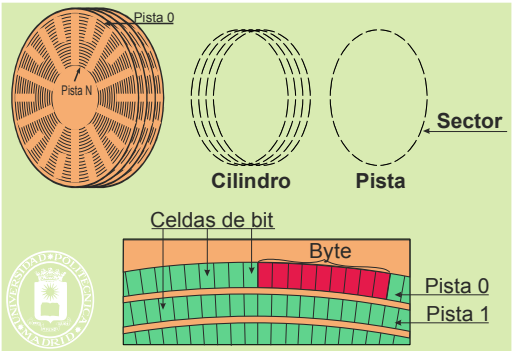


Figura 1.17 *Organización del disco.*

El **tiempo de acceso** de estos dispositivos viene dado por el tiempo que tardan en posicionar su brazo en la pista deseada, esto es, por el **tiempo de búsqueda**, más el tiempo que tarda la información del sector deseado en pasar delante de la cabeza por efecto de la rotación del disco, esto es, más la **latencia**, lo que suma en total del orden de los 15 ms por término medio. Observe que estos tiempos dependen de la posición de partida y de la posición deseada. No se tarda lo mismo en mover el brazo de la pista 1 a la 2, que de la 1 a la 1385. Por ello, los fabricantes suelen dar los valores medios y los peores.

La técnica de cabezas fijas, que fue popular hace años, consiste en montar un transductor por pista. Estas unidades presentan solamente latencia sin tiempo de búsqueda. Por tanto, suponiendo que gira a 10.000 rpm, tendrá un tiempo medio de acceso de 3 ms (½ revolución).

La **velocidad de transferencia** mide en B/s y sus múltiplos el número de bytes transferidos por unidad de tiempo, una vez alcanzado el sector deseado. Esta velocidad llega a superar los 100MiB/s.

Se dice que un fichero almacenado en el disco está **fragmentado** cuando se encuentra dividido en varios trozos repartidos a lo largo del disco. Si los ficheros de un disco están fragmentados, se dice que el disco está fragmentado. Esta situación se produce, como se verá más adelante, porque el sistema operativo, a medida que un fichero va creciendo, le va asignado espacio libre allí donde lo encuentra. Debido a los tiempos de acceso del disco magnético, la lectura de un fichero fragmentado requiere mucho más tiempo que si está sin fragmentar. De ahí que sea interesante realizar operaciones de desfragmentación de los discos magnéticos para mejorar sus prestaciones.

En la actualidad cada vez son más populares los discos de estado sólido basados en memorias Flash. Se utilizan especialmente en equipos portátiles dado que tienen un tiempo de espera y un consumo mucho más reducido que los magnéticos. Sus características más destacadas son las siguientes:

- Latencia en lectura: 50 µs.
- Velocidad de transferencia: 100-700 MiB/s.
- Escriben en bloques grandes (¼- 4 MiB).
- Número limitado de escrituras.
- No requieren desfragmentación (no mejora las prestaciones y realiza gran número operaciones de escritura).

El terminal

El terminal es la combinación de un monitor, que permite visualizar información, y de un teclado, que permite introducir información. Normalmente se añade un ratón o manejador de puntero. Es de destacar que el teclado y el monitor no están conectados entre sí. El que aparezca en pantalla lo que se teclea se debe a que el *software* hace lo que se denomina **eco**, es decir, reenvía a la pantalla las teclas pulsadas en el teclado.

El controlador de red

El controlador de red permite que unos computadores dialoguen con otros. Existen distintas tecnologías de red, cada una de las cuales exige su propio controlador. Las grandes redes o WAN (*Wide Area Network*) como Internet está formadas por numerosas redes físicas de distintas tecnologías que están interconectadas entre sí.

Las características más importantes de un controlador de red son las siguientes:

- Dentro de una red física cada controlador de red tiene una dirección que ha de ser única (p. ej. la dirección Ethernet). Esta dirección se denomina dirección física y es distinta de la dirección que puede tener el computador a nivel de la red WAN (p. ej. distinta de la dirección IP utilizada en Internet).
- La información a enviar se divide en pequeños trozos que se encapsulan en lo que se llaman **tramas**. Es frecuente que el tamaño máximo de estas tramas sea del orden del KiB. Las tramas son las unidades de información que circulan por la red.
- Cada trama incluye la dirección física origen y la dirección física destino, además de otras informaciones de control y de la información útil a enviar.
- Existe un protocolo que define la forma en que dialogan los controladores de red. Este protocolo define tramas de envío y de respuesta o confirmación, necesarios para que ambas partes estén siempre de acuerdo en la transmisión.

Cuando la información viaja por una WAN se ha de adaptar a las tramas y a las direcciones de cada red física que forman su recorrido. Por esta razón, la información a transmitir se organiza en **paquetes**. Cada paquete incluye la dirección WAN origen y destino, información de control y la información útil a enviar. Los paquetes se encapsulan en las tramas de las redes físicas, como se indica en la figura 1.18.

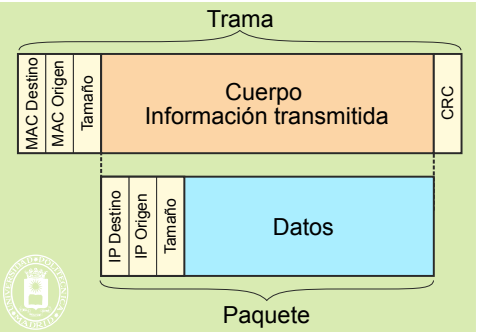


Figura 1.18 *Los paquetes se incorporan en una trama para su transmisión por la red de datos.*

Los factores más importantes que caracterizan las prestaciones de la red son la latencia y el ancho de banda. La **latencia** mide el tiempo que se tarda en enviar una unidad de información desde su origen a su destino. La unidad de información será la trama para el caso de una red física o será el paquete para el caso de la red WAN. A su vez, el **ancho de banda** expresa la cantidad de información que se puede enviar por unidad de tiempo. Este ancho de banda se suele expresar en bits por segundo o en número de tramas o de paquetes por segundo. Dado que las comunicaciones exigen un diálogo con envío de paquetes de confirmación o respuesta, en muchos casos el factor que limita la capacidad real de transmisión de una red es la latencia y no su ancho de banda.

1.6.4. E/S y concurrencia

Los periféricos son sensiblemente más lentos que el procesador, por ejemplo, durante el tiempo que se tarda en acceder a una información almacenada en un disco, un procesador moderno es capaz de ejecutar cientos o miles de millones de instrucciones máquina. Es, por tanto, muy conveniente que el procesador, mientras se está esperando a que se complete una operación de E/S, esté ejecutando un programa útil y no un bucle de espera.

Los computadores presentan tres modos básicos de realizar operaciones de E/S: E/S programada, E/S por interrupciones y E/S por DMA (*Direct Memory Access*). La E/S programada exige que el procesador esté ejecutando un programa de E/S, por lo que no existe ninguna concurrencia entre el procesador y la E/S. Sin embargo, en los otros dos modos de E/S el procesador no tiene que estar atendiendo directamente la E/S, por lo que puede estar ejecutando otro programa. Se dice, entonces, que existe **concurrencia** entre la E/S y el procesador. Esta concurrencia permite optimizar el uso del procesador, pero exige que los controladores de los periféricos sean más inteligentes, lo que supone que sean más complejos y más caros.

En términos generales, una operación de E/S se compone de tres fases: envío de la orden al periférico, lectura o escritura de los datos y fin de la operación.

La fase de envío de la orden consiste en escribir la orden en los registros del controlador del periférico, operación que puede hacerse mediante unas cuantas instrucciones máquina. Dado que el controlador es un dispositivo electrónico, estas escrituras se hacen a la velocidad del bus de E/S, sin esperas intermedias.

En la fase de transferencia de los datos interviene el periférico, en general, mucho más lento que el procesador. Imaginemos una lectura a disco. Para realizar esta operación con E/S programada debemos ejecutar un bucle que lea el registro de estado del controlador, observe si está activo el bit de dato disponible y, en caso positivo, que lo lea. El bucle podría tener la estructura que se muestra en el ejemplo del programa 1.1.

Programa 1.1 Bucle de E/S programada.

```
Ejemplo de bucle simplificado de lectura en pseudocódigo
   n = 0
   while n < m
      read registro  control
      if (registro_control = dato_disponible)
         read registro_datos
         store en memoria principal
         n = n + 1
      endif
   endwhile

;Ejemplo de bucle simplificado de lectura en ensamblador
P_ESTADO     ASSIGN      H'0045        ;Dirección del puerto de estado del periférico
P_DATO       ASSIGN      H'0046        ;Dirección del puerto de datos del periférico
MASCARA      ASSIGN      H'00000002    ;Definición de la máscara que especifica el bit de dato disponible
SECTOR       ASSIGN      H'0100        ;Definición del tamaño del sector en bytes
buffer:   DATA      [100]        ;Dirección del buffer donde se almacena el
                                 ;sector leído

          SUB .R1,.R1            ;Se pone a "0" el registro R1
BUCLE:    IN  .R2,/P_ESTADO      ;Se lee el puerto de estado del periférico al
                                 ;registro R2
          AND.R2,#MASCARA        ;Será <> 0 si el bit de dato disponible está a "1"
          BZ     /BUCLE          ;Si cero se repite el bucle
          IN  .R2,/P_DATO        ;Se lee el puerto de datos del periférico a R2
          ST     .R2,/buffer[.1] ;Se almacena el dato en la posición de memoria
                                 ;obtenida de sumar buffer al registro R1
          ADD.R1,#4              ;Se incrementa R1 en 4 (palabras de 4 bytes)
          CMP.R1,#SECTOR         ;Se compara el registro R1 con el tamaño del sector
          BLT /BUCLE             ;Si menor (R1<SECTOR)se repite el bucle
```

Observe que, hasta que no se disponga del primer dato, el bucle puede ejecutarse millones de veces, y que, entre dato y dato, se repetirá varias decenas de veces. Al llegar a completar el número *m* de datos a leer, se termina la operación de E/S.

Se denomina **espera activa** cuando un programa queda en un bucle hasta que ocurra un evento. La espera activa consume tiempo del procesador, por lo que es muy poco recomendable cuando el tiempo de espera es grande en comparación con el tiempo de ejecución de una instrucción.

En caso de utilizar E/S con interrupciones, el procesador, tras enviar la orden al controlador del periférico, puede dedicarse a ejecutar otro programa. Cuando el controlador disponga de un dato generará una interrupción externa. La rutina de interrupción deberá hacer la lectura del dato y su almacenamiento en memoria principal, lo cual conlleva un cierto tiempo del procesador. En este caso se dice que se hace **espera pasiva**, puesto que el programa que espera el evento no está ejecutándose. La interrupción externa se encarga de «despertar» al programa cuando ocurre el evento.

Finalmente, en caso de utilizar E/S por DMA, el controlador del dispositivo se encarga directamente de ir transfiriendo los datos entre el periférico y la memoria, sin interrumpir al procesador. Una vez terminada la transferencia de todos los datos, el controlador genera una interrupción externa de forma que se sepa que ha terminado. Un

controlador que trabaje por DMA dialoga directamente con la memoria principal del computador. La fase de envío de una orden a este tipo de controlador exige incluir la dirección de memoria donde está el *buffer* de la transferencia, el número de palabras a transmitir y la dirección de la zona del periférico afectada.

Existe un tipo evolucionado de DMA llamado **canal**, que es capaz de leer las órdenes directamente de una cola de trabajos, construida en memoria principal, y de almacenar los resultados de las órdenes en una cola de resultados, construida también en memoria principal, y todo ello sin la intervención del procesador. El canal solamente interrumpe al procesador cuando terminado un trabajo encuentra vacía la cola de trabajos.

La tabla 1.2 presenta la ocupación del procesador en la realización de las distintas actividades de una operación de E/S según el modelo de E/S utilizado.
Puede observarse que la solución que presenta la máxima concurrencia, y que descarga al máximo al procesador, es la de E/S por canal. Por otro lado, es la que exige una mayor inteligencia por parte del controlador.

Tabla 1.2 *Ocupación del procesador en operaciones de entrada/salida.*

	Enviar orden	Esperar dato	Transferir dato	Fin operación
E/S programada	Procesador	Procesador	Procesador	Procesador
E/S por interrupciones	Procesador	Controlador	Procesador	Procesador
E/S por DMA	Procesador	Controlador	Controlador	Procesador
E/S por canal	Controlador	Controlador	Controlador	Controlador

Un aspecto fundamental de esta concurrencia es su explotación. En efecto, de nada sirve descargar al procesador del trabajo de E/S si durante ese tiempo no tiene nada útil que hacer. Será una función importante del sistema operativo explotar esta concurrencia entre la E/S y el procesador, haciendo que este último realice trabajo útil el mayor tiempo posible.

1.6.5. Buses y direccionamiento

Un computador moderno incluye diversos buses para interconectar el procesador, la memoria principal y los controladores de los periféricos. Estos buses pueden utilizar espacios de direcciones propios, lo que obliga a disponer de puentes que traduzcan las direcciones de unos espacios a otros. Por otro lado, cada vez son más utilizados los buses serie, en los que la información de dirección circula por el bus en la cabecera de las tramas.

Por ejemplo, el procesador PXA-255 de Intel (comercializado en el año 2003 para diseño de PDA (*Personal Data Assistant*)), cuenta con un único espacio de direcciones que se utiliza para la memoria principal, para dos buses PCMCIA, para los periféricos integrados y para los periféricos externos. Este espacio se reparte según lo indicado en la tabla 1.3. Por tanto, se trata de una máquina con mapa de memoria y E/S común.

Tabla 1.3 *Reparto del mapa de direcciones del PXA-255.*

Dirección	Tamaño	Uso inicial
0xA000 0000	(64 MiB)	SDRAM banco 3
0xA400 0000	(64 MiB)	SDRAM banco 2
0xA800 0000	(64 MiB)	SDRAM banco 1
0xAC00 0000	(64 MiB)	SDRAM banco 0
0x4800 0000	(64 MiB)	Registros del controlador SDRAM
0x4400 0000	(64 MiB)	Registros del controlador LCD
0x4000 0000	(64 MiB)	Registros de los controladores de los periféricos internos
0x3000 0000	(256 MiB)	PCMCIA/CF - bus 1
0x2000 0000	(256 MiB)	PCMCIA/CF - bus 0
0x0000 0000	(384 MiB)	Periféricos externos del 0 al 5, dedicando 64 MiB a cada uno

Se puede observar en dicha tabla que los buses PCMCIA/CF están proyectados en las direcciones 0x2000 0000 y 0x3000 0000 respectivamente. Esto significa que la dirección 0 del PCMCIA/CF - bus 0 se proyecta en la dirección 0x2000 0000 del procesador, por lo que los periféricos conectados al bus ven unas direcciones distintas de las que ve el procesador.

Por otro lado, el bus I^2C, incluido en este procesador, es un bus serie de dos hilos a 400 kb/s que permite conectar una gran variedad de pequeños periféricos. La información circula en tramas que incluyen una dirección de 7 bits, lo que permite diferenciar hasta 128 periféricos distintos.

1.7. PROTECCIÓN

Como veremos más adelante, una de las funciones del sistema operativo es la protección de unos usuarios contra otros: ni por malicia ni por descuido un usuario deberá acceder a la información de otro, a menos que el propietario de la misma lo permita. Para ello es necesario imponer ciertas restricciones a los programas en ejecución y es necesario comprobar que se cumplen. Dado que, mientras se está ejecutando un programa de usuario, no se está ejecutan-

do el sistema operativo, la vigilancia de los programas se ha de basar en mecanismos *hardware* (aclaración 1.3). En esta sección se analizarán estos mecanismos, para estudiar en capítulos posteriores cómo resuelve el sistema operativo los conflictos detectados. Se analizará en primer lugar el mecanismo de protección que ofrece el procesador, para pasar seguidamente a los mecanismos de protección de memoria y a la protección de E/S.

Aclaración 1.3. El sistema operativo no puede vigilar los programas de los usuarios, puesto que está «dormido» mientras éstos ejecutan. Por tanto, **sin mecanismos *hardware* específicos no puede haber protección**.

1.7.1. Mecanismo de protección del procesador

Como ya se ha explicado, el mecanismo de protección que ofrece el procesador consiste en tener varios modos de ejecución. En modo privilegiado se pueden ejecutar todas las instrucciones máquina y se puede acceder a todos los registros y a la totalidad de los mapas de memoria y de E/S. Sin embargo, en modo usuario:

- Se prohíben ciertas instrucciones de máquina.
- Se limitan los registros que se pueden acceder y modificar.
- Se limita el mapa de memoria a una parte del mismo.
- Se prohíbe el mapa de entrada salida, por lo que los programas de usuario **no pueden acceder directamente a los periféricos**. Deben pedírselo al sistema operativo.

Cuando el *hardware* detecta que un programa en modo usuario intenta ejecutar una instrucción no permitida, o acceder directamente a un periférico o una dirección fuera del mapa de memoria, genera una excepción de error. Dicha excepción activa al sistema operativo, que decide la acción a tomar.

Los programas de usuario deberán ejecutar en el modo más restrictivo para evitar que puedan interferir unos con otros. Es absolutamente imprescindible evitar que un programa de usuario pueda poner el procesador en modo privilegiado, puesto que se perdería la protección. Por ello, no existe ninguna instrucción máquina que cambie directamente los bits de modo de ejecución de usuario a privilegiado. Sin embargo, existe la instrucción máquina inversa, que cambia de modo privilegiado a modo usuario. Esta instrucción, que suele ser el RETI (retorno de interrupción), es utilizada por el sistema operativo antes de dejar que ejecute un programa de usuario.

El único mecanismo que tiene el procesador para pasar de modo usuario a modo privilegiado es la interrupción. Por tanto, si queremos garantizar la protección es imprescindible cumplir que:

- Todas las direcciones de la tabla de interrupciones apunten a rutinas del sistema operativo, puesto que es el único que debe ejecutar en modo privilegiado.
- Los programas de usuario no puedan modificar dicha tabla.

Para que un programa de usuario —que ejecuta en modo usuario— pueda activar al sistema operativo —que ejecuta en modo privilegiado— existe la instrucción máquina de llamada al sistema. Dicha instrucción produce un ciclo de aceptación de interrupción, por lo que indirectamente pone al procesador en modo privilegiado y salta al sistema operativo. De forma genérica llamaremos a esa instrucción TRAP.

1.7.2. Mecanismos de protección de memoria

Los mecanismos de protección de memoria deben evitar que un programa en ejecución direccione posiciones de memoria que no le hayan sido asignadas por el sistema operativo.

La solución más empleada, en las máquinas de **memoria real**, consiste en incluir una pareja de registros valla (límite y base), como los mostrados en la figura 1.19. En esta solución se le asigna al programa una zona de memoria contigua. Todos los direccionamientos se calculan sumando el contenido del registro base, de forma que para el programa es como si estuviese cargado a partir de la posición «0» de memoria. Además, en cada acceso un circuito comparador compara la dirección generada con el valor del registro límite. En caso de desbordamiento el comparador genera una excepción *hardware* síncrona de **violación de memoria**, para que el sistema operativo trate esta situación, lo que, en la práctica, supone que parará el programa produciendo un volcado de memoria (*core dump*), que consiste en almacenar en disco el contenido de la memoria. Posteriormente, se puede determinar la razón del fallo analizando el volcado de memoria mediante una adecuada herramienta de depuración.

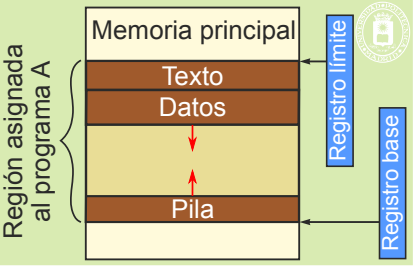


Figura 1.19 *Uso de registros valla.*

En los sistemas con memoria virtual suelen existir dos mecanismos de protección de memoria. El primero se basa en que en modo privilegiado se puede direccionar todo el mapa de memoria virtual, mientras que en modo usuario solamente se puede direccionar una parte del mapa (p. ej.: las direcciones que empiezan por «0»), produciéndose una excepción *hardware* síncrona de violación de memoria en caso de que se genere una dirección no permitida (p. ej.: que empiece por «1»).

El segundo consiste en dotar, a cada programa en ejecución, de su propia tabla de páginas, que se selecciona mediante el registro RIED. De esta forma, se consigue que cada programa en ejecución disponga de su propio espacio virtual, por lo que no puede acceder a los espacios de memoria de los otros programas que estén ejecutando. Como muestra la figura 1.20, la tabla de páginas especifica los marcos y páginas de intercambio que tiene asignados cada programa.

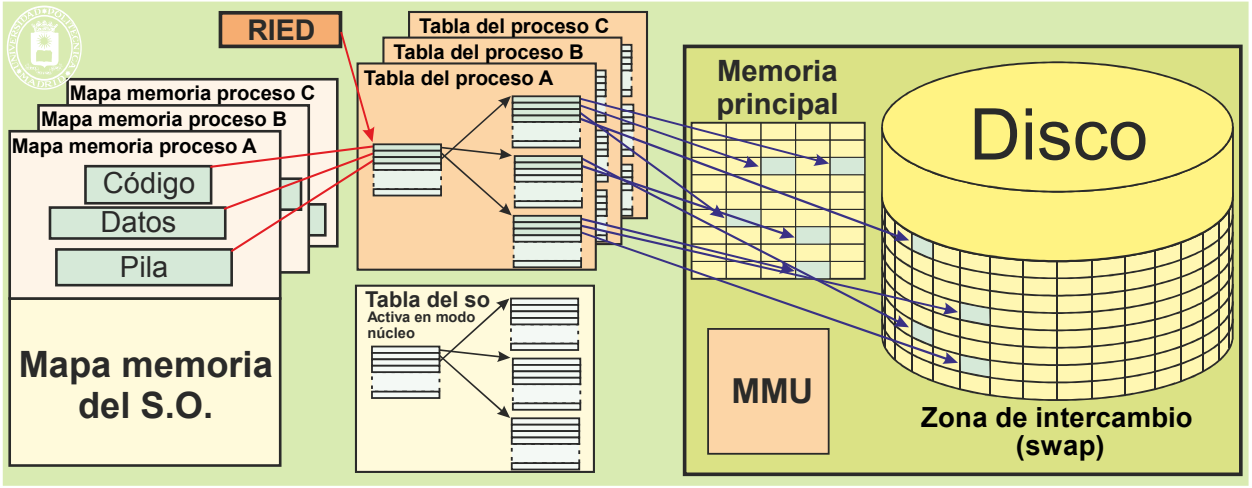


Figura 1.20 *La tabla de páginas como mecanismo de protección de memoria.*

La tabla de páginas incorpora información de protección tanto a nivel de región como de página. Típicamente se pueden especificar permisos de lectura, escritura y ejecución.

La MMU, al mismo tiempo que realiza la traducción de cada dirección, comprueba que:

- La dirección es correcta, es decir, que está dentro de las regiones asignadas al programa.
- El acceso solicitado está permitido (lectura, escritura o ejecución).

1.7.3. Protección de entrada/salida

La protección de la entrada/salida es fundamental para evitar que unos usuarios puedan acceder indiscriminadamente a los periféricos. Uno de los objetivos prioritarios es la protección de los periféricos de almacenamiento. Un controlador de disco no entiende de ficheros ni de derechos de acceso, sólo de sectores. Por tanto, un programa que pueda acceder directamente a dicho controlador puede leer y escribir en cualquier parte del dispositivo. Prohibiendo el acceso de los programas de usuario al mapa de E/S se evita dicho problema. Cuando un programa necesite acceder a un periférico deberá solicitárselo al sistema operativo, que previamente se asegurará de que la operación está permitida.

Dependiendo de que la arquitectura del procesador sea de mapa de E/S propio o de mapa de E/S común con el mapa de memoria, la protección se realiza por mecanismos distintos.

Cuando existe un mapa de E/S propio, el procesador cuenta con instrucciones de máquina específicas para acceder a ese mapa. Estas son las instrucciones de E/S. La protección viene garantizada porque dichas instrucciones solamente son ejecutables en modo privilegiado. Si un programa de usuario las intenta ejecutar el procesador produce una excepción *hardware* síncrona de código de instrucción inválido.

Cuando el mapa de E/S es común con el mapa de memoria, el acceso se realiza mediante las instrucciones máquina de LOAD y STORE, que están permitidas en modo usuario. La protección se basa, en este caso, en proteger ese espacio del mapa de memoria, de forma que los usuarios no lo puedan acceder.

1.8. MULTIPROCESADOR Y MULTICOMPUTADOR

Dada la insaciable apetencia por máquinas de mayor potencia de cómputo, cada vez es más corriente encontrarse con computadores que incluyen más de un procesador. Buena prueba de ello son los procesadores multinúcleo o *multicore*, puesto que cada núcleo o *core* es realmente un procesador. Esta situación tiene una repercusión inmediata en el sistema operativo, que ha de ser capaz de explotar adecuadamente todos estos procesadores y que ha de tener en cuenta los distintos tiempos de acceso a las diversas memorias que componen el sistema. Las dos arquitecturas para estos computadores son la de multiprocesador y la de multicomputador, que se analizan seguidamente.

Multiprocesador

Como muestra la figura 1.21, un multiprocesador es una máquina formada por un conjunto de procesadores que comparten el acceso a una memoria principal común y que están gobernados por un único sistema operativo. Cada procesador ejecuta su propio programa, debiendo todos ellos compartir la memoria principal común.

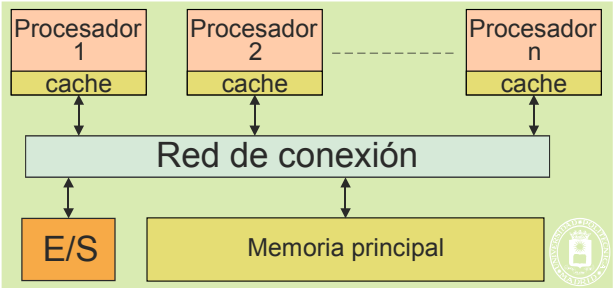


Figura 1.21 *Estructura de un multiprocesador.*

Las características más importantes de estas máquinas son las siguientes:

- El acceso a datos comunes por parte de varios programas es muy sencillo, puesto que utilizan la misma memoria principal.
- Cada procesador suele tener su propia cache y TLB, lo que exige un diseño que garantice la coherencia de las distintas copias de información que residan en las distintas caches privadas de los procesadores y en sus TLB.
- Suele existir un único reloj RTC, de forma que todos los procesadores comparten la misma hora.
- Suele existir una interrupción especial para que cada procesador pueda ser interrumpido por los demás. Esta interrupción se suele llamar IPI (*InterProcessor Interrupt*).
- En muchos casos existe un APIC (*Advanced Programmable Interrupt Controller*) que recibe todas las interrupciones externas y las distribuye entre los procesadores, de forma inteligente y programable. El APIC incluye mecanismos de inhibición que son comunes a todos los procesadores.
- Es frecuente que los periféricos sean compartidos y que sus interrupciones se distribuyan entre los procesadores.
- Los procesadores suelen contar con una instrucción máquina de TestAndSet, que ejecuta de forma atómica, lo que permite instrumentar mecanismos de exclusión mutua para ordenar el acceso a las zonas de memoria compartida (véase el capítulo "4 Gestión de memoria").
- Su mayor inconveniente consiste en el limitado número de procesadores que se pueden incluir sin incurrir en el problema de saturar el ancho de banda de la memoria común.

Existen tres tipos de multiprocesador en relación al esquema de memoria utilizado:

- Esquema UMA (*Uniform Memory Access*), también llamado **multiprocesador simétrico**. Todos los procesadores comparten uniformemente toda la memoria principal, pudiendo acceder con igual velocidad a cada una de sus celdas.
- Esquema NUMA (*Non Uniform Memory Access*). Todos los procesadores pueden acceder a toda la memoria principal, pero en cada uno de ellos existen unas zonas que acceden a mayor velocidad.
- Esquema COMA (*cache Only Memory Architecture*). En este caso solamente existen memorias cache que son de gran tamaño.

La solución UMA es la más sencilla de utilizar, pero admite el menor número de procesadores sin saturar la memoria principal (un límite típico es el de 16 procesadores).

Un problema de las arquitecturas NUMA y COMA es que el tiempo que se tarda en acceder a una información residente en memoria depende de su posición. Surge, por tanto, la necesidad de optimizar el sistema, manteniendo la información cerca del procesador que la está utilizando.

Multicomputador

El multicomputador es una máquina compuesta por varios nodos, estando cada nodo formado por uno o varios procesadores, su memoria principal y, en su caso, elementos de E/S. La figura 1.22 muestra el esquema global de estas máquinas.

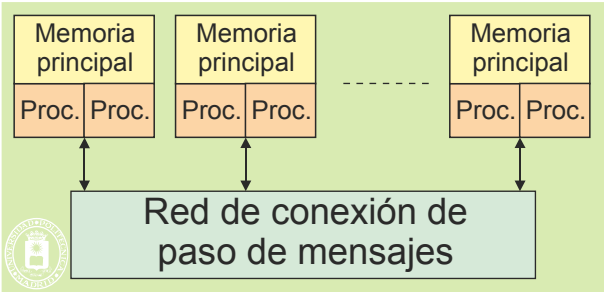


Figura 1.22 *Estructura de un multiprocesador.*

Sus características más importantes son las siguientes:

- Al contrario que en los multiprocesadores, en estas máquinas los programas de dos o más nodos no pueden compartir datos en memoria principal.
- No existe la limitación anterior en cuanto al número de procesadores que se pueden incluir. Buena prueba de ello es que existen máquinas con más de un millón de procesadores.
- La programación se basa en el paradigma de paso de mensajes, lo que las hace más difíciles de utilizar.
- Cada nodo tiene su propia copia del sistema operativo.
- La comunicación entre los procesadores se realiza mediante una red de interconexión y el correspondiente *software* de comunicaciones.

Finalmente, destacaremos que los computadores actuales más potentes están basados en una arquitectura de tipo multicomputador en la que cada nodo es a su vez un multiprocesador UMA de unos pocos procesadores.

1.9. PRESTACIONES

En esa sección se recogen algunas consideraciones relativas a las prestaciones de los procesadores, los discos y la memoria virtual.

Procesadores

Los procesadores actuales son segmentados y superescalares. Esto significa que ejecutan al tiempo varias instrucciones máquina, de forma que en cada ciclo de reloj ejecutan más de una instrucción. Un procesador de 4 GHz puede ejecutar 8 GIPS (8 mil millones de instrucciones máquina por segundo). Sin embargo, las memorias principales tienen una latencia de decenas de ns. Ello obliga a disponer de varios niveles de memoria cache para poder alimentar de información al procesador sin que se produzcan grandes parones. Un fallo de acceso a la memoria cache se resuelve generalmente con el acceso a la cache de nivel superior, por lo que el tiempo de penalización es reducido. Esto significa que toda la gestión de las memorias cache ha de hacerse por *hardware* a muy alta velocidad.

Discos

Los discos magnéticos son dispositivos comparativamente mucho más lentos que el procesador, puesto que el tiempo de acceso es del orden de los 10 ms, tiempo durante el cual el procesador puede ejecutar cientos de millones de instrucciones. Esto significa dos cosas: que los fallos de página pueden ser atendidos por *software* (el coste que ello implica es pequeño) y que el procesador no se puede quedar esperando a que se resuelva el fallo de página, ha de pasar a ejecutar otros programas.

Los discos de estado sólido (SSD) tienen un tiempo de acceso 100 veces menor que los magnéticos y no sufren de fragmentación.

Memoria virtual

El tamaño del espacio virtual suele ser muy grande. En la actualidad se emplean direcciones de 32, 48 o hasta 64 bits, lo que significa espacios virtuales de 2^{32}, 2^{48} y 2^{64} bytes. A esto hay que añadir que cada programa en ejecución tiene su propio espacio virtual. Sin embargo, el soporte físico disponible no cubre toda la memoria virtual de todos los programas, puesto que se compone de la suma de la memoria principal (p. ej.: 16 GiB) más el espacio de intercambio (p. ej.: 32 GiB). En general, esto no es ningún problema, puesto que es suficiente para que los programas dispongan del espacio necesario. Recuérdese que los espacios virtuales no asignados por el sistema operativo a un programa no tienen soporte físico.

Sin embargo, para que un computador con memoria virtual pueda competir con la memoria real, la traducción ha de tardar una fracción del tiempo de acceso a memoria. En caso contrario sería mucho más rápido y, por ende más económico, el sistema real. La traducción se ha de realizar en todos y cada uno de los accesos que realiza el procesador a su espacio de memoria. Dado que una parte importante de estos accesos se realizan con acierto sobre la cache de primer nivel, la TLB ha de construirse de forma que no entorpezca estos accesos, es decir, que la combinación de la TLB más la cache de primer nivel ha de ser capaz de producir un resultado en un ciclo de reloj. Para una máquina de 4 GHz esto significa que se dispone de 0,25 ns para dicha combinación.

Para hacernos una idea del valor que debe tener la tasa de aciertos de la memoria virtual basta con calcular el número de instrucciones máquina que ejecuta el computador durante el tiempo que se tarda en atender un fallo. Por término medio, durante todo ese tiempo no deberá producirse otro fallo, puesto que el dispositivo de intercambio está ocupado. Supóngase una máquina de 5 GIPS y un tiempo de acceso del disco de 10 ms. Si no hay que salvar la página a sustituir el tiempo tardado en atender el fallo será de 10 ms, mientras que será de 20 ms si hay que salvar dicha página. Esto significa que el procesador ha ejecutado 50 o 100 millones de instrucciones y, por tanto, que la tasa de fallos debe ser de uno por cada 50 o 100 millones de accesos.

Redes

El dato de red más frecuente que emplean los suministradores es el del ancho de banda expresado en bits por segundo (b/s). Valores típicos son los de 10 Mb/s, 100 Mb/s, 1 Gb/s y 10 Gb/s de las distintas versiones de la red Ethernet, los 2 Gb/s de la red Myrinet, los 3,75Gb/s de InfiniBand o los 1 a 16 Mb/s de ADSL. Este valor indica la capacidad pico de transmisión y es, en muchos casos, muy poco significativo debido a la latencia.

Los valores de latencia varían sustancialmente en función de la carga que tenga la red de comunicación. Es, por tanto, recomendable realizar pruebas reales con la red en situación de carga normal para poder determinar si la capacidad real de transmisión viene dada por el ancho de banda disponible o por la latencia.

Para optimizar la utilización de las redes es interesante que tanto el sistema operativo como las aplicaciones agrupen lo máximo posible la información, utilizando el menor número de paquetes y tramas. De esta forma se aumenta la proporción de información útil con respecto al total de información enviado y se disminuye la sobrecarga de gestión, es decir, el tiempo de procesamiento dedicado al tratamiento de paquetes y tramas.

1.10. LECTURAS RECOMENDADAS

Para completar el contenido de este tema puede consultarse cualquiera de los siguientes libros:
[Carpinelli, 2001], [Hamacher, 2002], [Hennessy, 2002], [deMiguel, 2004], [Patterson, 2004], [Stallings, 2003] y [Tanenbaum, [2005].

1.11. EJERCICIOS

1. ¿Qué contiene una entrada de la tabla de interrupciones IDT?
   a) El nombre de la rutina de tratamiento.
   b) La dirección de la rutina de tratamiento.
   c) El número de la interrupción.
   d) El nombre de la tarea del sistema operativo que trata la interrupción.

2. ¿Cuál de las siguientes instrucciones máquina no debería ejecutarse en modo privilegiado? Razone su respuesta.
   a) Inhibir interrupciones.
   b) Instrucción TRAP.
   c) Modificar el reloj del sistema.
   d) Cambiar el mapa de memoria.

3. Considere un sistema con un espacio lógico de memoria de 128 KiB páginas con 8 KiB cada una, una memoria física de 64 MiB y direccionamiento a nivel de byte. ¿Cuántos bits hay en la dirección?

4. Sea un sistema de memoria virtual paginada con direcciones de 32 bits que proporcionan un espacio virtual de 220 páginas y con una memoria física de 32 MiB. ¿Cuánto ocupará la tabla de marcos de página si cada entrada de la misma ocupa 32 bits?

5. Sea un computador con memoria virtual y un tiempo de acceso a memoria de 70 ns. El tiempo necesario para tratar un fallo de página es de 9 ms. Si la tasa de aciertos a memoria principal es del 98%, ¿cuál será el tiempo medio de acceso a una palabra en este computador?

2

INTRODUCCIÓN A LOS SISTEMAS OPERATIVOS

Un sistema operativo es un programa que tiene encomendadas una serie de funciones cuyo objetivo es simplificar el manejo y optimizar la utilización del computador, haciéndole seguro y eficiente. En este capítulo, prescindiendo de muchos detalles en aras a la claridad, se introducen varios conceptos sobre los sistemas operativos. Muchos de los temas tratados se plantearán con más detalle en los capítulos posteriores, dado el carácter introductorio empleado aquí. De forma más concreta, el objetivo del capítulo es presentar una visión globalizadora de los siguientes aspectos:

- *Definición y funciones del sistema operativo.*
- *Arranque del computador y del sistema operativo.*
- *Activación del sistema operativo.*
- *Tipos de sistemas operativos.*
- *Gestión de procesos.*
- *Gestión de memoria.*
- *Gestión de la E/S.*
- *Gestión de ficheros y directorios.*
- *Comunicación y sincronización entre procesos.*
- *Seguridad y protección.*
- *Activación del sistema operativo y llamadas al sistema.*
- *Interfaz de usuario del sistema operativo e interfaz del programador.*
- *Diseño de los sistemas operativos.*
- *Historia y evolución de los sistemas operativos.*

2.1. ¿QUÉ ES UN SISTEMA OPERATIVO?

En esta sección se plantea los conceptos de máquina desnuda, ejecutable y usuario, para pasar acto seguido a introducir el concepto de sistema operativo, así como sus principales funciones.

Máquina desnuda

El término de máquina desnuda se aplica a un computador carente de sistema operativo. El término es interesante porque resalta el hecho de que un computador en sí mismo no hace nada. Como se vio en el capítulo anterior, un computador solamente es capaz de repetir a alta velocidad la secuencia de: lectura de instrucción máquina, incremento del contador de programa o PC y ejecución de la instrucción leída. Para que realice una función determinada se han de cumplir, como muestra la figura 2.1, las dos condiciones siguientes:

- Ha de existir un programa máquina específico para realizar dicha función. Además dicho programa así como sus datos han de estar ubicados en el mapa de memoria.
- Ha de conseguirse que el registro PC contenga la dirección de comienzo del programa.

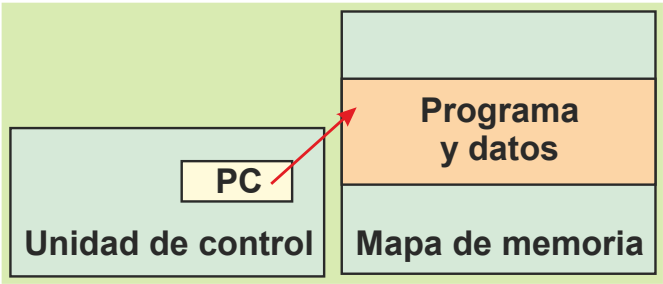


Figura 2.1 *Para ejecutar un programa ha de estar ubicado con sus datos en el mapa de memoria del computador y el contador de programa ha de apuntar al comienzo del mismo.*

La misión del sistema operativo, como se verá en la siguiente sección, es completar (vestir) la máquina mediante una serie de programas que permitan su cómodo manejo y utilización.

Programa ejecutable

Un programa ejecutable, también llamado simplemente ejecutable, es un programa en lenguaje máquina que puede ser cargado en memoria para su ejecución. Cada ejecutable está normalmente almacenado en un fichero que contiene la información necesaria para ponerlo en ejecución. La estructura típica de un ejecutable comprende la siguiente información:

- Cabecera que contiene, entre otras informaciones, las siguientes:
  - Estado inicial de los registros del procesador.
  - Tamaño del código y de los datos.
  - Palabra «mágica» que identifica al fichero como un ejecutable.
- Código en lenguaje máquina.
- Datos con valor inicial.
- Tabla de símbolos.

Usuario

En primera instancia podemos decir que un usuario es una persona autorizada a utilizar un sistema informático.

2.1.1. Sistema operativo

Un sistema operativo (SO) es un programa que tiene encomendadas una serie de funciones cuyo objetivo es simplificar el manejo y la utilización del computador, haciéndolo seguro y eficiente. Históricamente se han ido completando las misiones encomendadas al sistema operativo, por lo que los productos comerciales actuales incluyen una gran cantidad de funciones, como son interfaces gráficas, protocolos de comunicación, bases de datos, etc.

Las funciones clásicas del sistema operativo se pueden agrupar en las tres categorías siguientes:

- Gestión de los recursos del computador.
- Ejecución de servicios para los programas en ejecución.
- Ejecución de los mandatos de los usuarios.

El objetivo del computador es ejecutar programas, por lo que el objetivo del sistema operativo es facilitar la ejecución de dichos programas. La ejecución de programas en una máquina con sistema operativo da lugar al concepto de **proceso**, más adelante se presenta el concepto de proceso.

Como muestra la figura 2.2, el sistema operativo está formado conceptualmente por tres capas principales. La capa más cercana al *hardware* se denomina **núcleo** (***kernel***), y es la que gestiona los recursos *hardware* del sistema y la que suministra la funcionalidad básica del sistema operativo. Esta capa ha de ejecutar en modo privilegiado, mientras que las otras pueden ejecutar en modos menos permisivos.

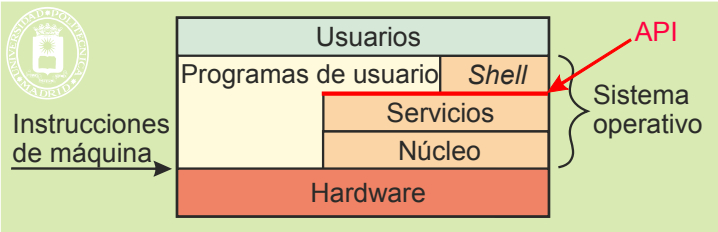


Figura 2.2 *Niveles del sistema operativo.*

La capa de **servicios,** o de **llamadas al sistema,** ofrece a los procesos unos servicios a través de una interfaz de programación o **API** (*Application Programming Interface*). Desde el punto de vista de los procesos esta capa extiende la funcionalidad del computador, por lo que se suele decir que el sistema operativo ofrece una **máquina extendida** a los procesos. De esta forma se facilita la elaboración de los programas, puesto que se apoyan en las funciones que les suministra el sistema operativo.

La capa de **intérprete de mandatos** o *shell* suministra una interfaz a través de la cual el usuario puede dialogar de forma interactiva con el computador. El *shell* recibe los mandatos u órdenes del usuario, los interpreta y, si puede, los ejecuta. Dado que el *shell* suele ejecutar en modo usuario, algunos autores consideran que no forma parte del sistema operativo. En opinión de los autores de este texto, el *shell* es uno más de los elementos del sistema operativo.

Seguidamente se analizarán cada una de estas tres facetas del sistema operativo.

El sistema operativo como gestor de recursos

Recurso es todo medio o bien que sirve para conseguir lo que se pretende. En un computador la memoria y el procesador son recursos físicos y un temporizador o un puerto de comunicaciones son ejemplos de recursos lógicos. Los recursos son limitados y son reutilizados una vez que el proceso que los disfruta ya no los necesite.

En un computador actual suelen coexistir varios procesos, del mismo o de varios usuarios, ejecutando simultáneamente. Estos procesos compiten por los recursos, siendo el sistema operativo el encargado de arbitrar su asignación y uso. El sistema operativo debe asegurar que no se produzcan violaciones de seguridad, evitando que los procesos accedan a recursos a los que no tienen derecho. Además, ha de suministrar información sobre el uso que se hace de los recursos.

Asignación de recursos

Para asignar los recursos a los procesos, el sistema operativo ha de mantener unas estructuras que le permitan saber qué recursos están libres y cuáles están asignados a cada proceso. La asignación de recursos se realiza según la disponibilidad de los mismos y la prioridad de los procesos, debiéndose resolver los conflictos causados por las peticiones coincidentes.

Especial mención reviste la recuperación de los recursos cuando los procesos ya no los necesitan. Una mala recuperación de recursos puede hacer que el sistema operativo considere, por ejemplo, que ya no le queda memoria disponible cuando, en realidad, sí la tiene. La recuperación se puede hacer o bien porque el proceso que tiene asignado el recurso le comunica al sistema operativo que ya no lo necesita, o bien porque el proceso haya terminado.

Las **políticas** de gestión de recursos determinan los criterios seguidos para asignar los recursos. Estas políticas dependen del objetivo a alcanzar por el sistema. No tienen las mismas necesidades los sistemas personales, los sistemas departamentales, los sistemas de tiempo real, etc.

Muchos sistemas operativos permiten establecer cuotas o límites en los recursos asignados a cada proceso o usuario. Por ejemplo, se puede limitar la cantidad de disco, memoria o tiempo de procesador asignados.

Protección

El sistema operativo ha de garantizar la protección entre los usuarios del sistema: ha de asegurar la confidencialidad de la información y que unos trabajos no interfieran con otros. Para conseguir este objetivo ha de impedir que unos procesos puedan acceder a los recursos asignados a otros procesos.

Contabilidad

La contabilidad permite medir la cantidad de recursos que, a lo largo de su ejecución, utiliza cada proceso. De esta forma se puede conocer la carga de utilización o trabajo que tiene cada recurso y se pueden imputar a cada usuario los recursos que ha utilizado. Cuando la contabilidad se emplea meramente para conocer la carga de los componentes del sistema se suele denominar monitorización. La monitorización se utiliza, especialmente, para determinar los puntos sobrecargados del computador y, así, poder corregirlos.

El sistema operativo como máquina extendida

El sistema operativo ofrece a los programas un conjunto de servicios, o llamadas al sistema, proporcionándoles una visión de máquina extendida. El modelo de programación que ofrece el *hardware* se complementa con estos servi-

cios *software*, que permiten ejecutar de forma cómoda y protegida ciertas operaciones complejas. La alternativa consistiría en complicar los programas de usuario y en no tener protección frente a otros usuarios.

Las funciones de máquina extendida se pueden agrupar en las cuatro clases siguientes: ejecución de programas, operaciones de E/S, operaciones sobre ficheros y detección y tratamiento de errores. Gran parte de este texto se dedicará a explicar los servicios ofrecidos por los sistemas operativos, por lo que aquí nos limitaremos a hacer unos simples comentarios sobre cada una de estas cuatro clases.

Ejecución de programas

El sistema operativo incluye servicios para lanzar la ejecución de un programa, creando un proceso, así como para parar o abortar la ejecución de un proceso. También existen servicios para conocer y modificar las condiciones de ejecución de los procesos.

Bajo la petición de un usuario, el sistema operativo leerá un ejecutable, para lo cual deberá conocer su estructura, lo cargará en memoria y lo pondrá en ejecución. Observe que varios procesos pueden estar ejecutando el mismo programa, por ejemplo, varios usuarios pueden haber pedido al sistema operativo la ejecución del mismo programa editor.

Órdenes de E/S

Los servicios de E/S ofrecen una gran comodidad y protección al proveer a los programas de operaciones de lectura, escritura y modificación del estado de los periféricos, puesto que la programación de las operaciones de E/S es muy compleja y dependiente del *hardware* específico de cada periférico. Los servicios del sistema operativo ofrecen un alto nivel de abstracción, de forma que el programador de aplicaciones no tenga que preocuparse de esos detalles.

Operaciones sobre ficheros

Los ficheros ofrecen un nivel de abstracción mayor que el de las órdenes de E/S, permitiendo operaciones tales como creación, borrado, renombrado, apertura, escritura y lectura de ficheros. Observe que muchos de los servicios son parecidos a las operaciones de E/S y terminan concretándose en este tipo de operación.

Servicios de memoria

El sistema operativo incluye servicios para que el proceso pueda solicitar y devolver zonas de memoria para albergar datos. También suele incluir servicios para que dos o más procesos puedan compartir una zona de memoria.

Comunicación y sincronización entre procesos

Los servicios de comunicación entre procesos son muy importantes, puesto que constituyen la base sobre la que se ha construido Internet. Servicios como el *pipe* o compartir memoria permiten la comunicación entre procesos de un mismo computador, pero servicios como el *socket* permiten la comunicación entre procesos que ejecutan en máquinas remotas pero conectadas en red.

Los servicios de sincronización como el semáforo o los *mutex*, permiten sincronizar la ejecución de los procesos, es decir, conseguir que ejecuten de forma ordenada (como en un debate en el que nadie le quite la palabra a otro).

Detección y tratamiento de errores

Además de analizar detalladamente todas las órdenes que recibe, para comprobar que se pueden realizar, el sistema operativo se encarga de tratar todas las condiciones de error que detecte el *hardware*. Como más relevantes, destacaremos las siguientes: errores en las operaciones de E/S, errores de paridad en los accesos a memoria o en los buses y errores de ejecución en los programas, tales como los desbordamientos, las violaciones de memoria, los códigos de instrucción prohibidos, etc.

El sistema operativo como interfaz de usuario

El módulo del sistema operativo que permite que los usuarios dialoguen de forma interactiva con él es el intérprete de mandatos o *shell*. El *shell* se comporta como un bucle infinito que está repitiendo constantemente la siguiente secuencia:

- Espera una orden del usuario. En el caso de interfaz textual, el *shell* está pendiente de lo que escribe el usuario en la línea de mandatos. En las interfaces gráficas está pendiente de los eventos del apuntador (ratón) que manipula el usuario, además, de los del teclado.
- Analiza la orden y, en caso de ser correcta, la ejecuta, para lo cual emplea los servicios del sistema operativo.
- Concluida la orden muestra un aviso o *prompt* y vuelve a la espera.

El diálogo mediante interfaz textual exige que el usuario memorice la sintaxis de los mandatos, con el agravante de que son distintos para cada sistema operativo (p. ej.: para mostrar el contenido de un fichero en Windows se emplea el mandato `type`, pero en UNIX se usa el mandato `cat`). Por esta razón, cada vez son más populares los intérpretes de mandatos con interfaz gráfica, como el que se encuentra en las distintas versiones de Windows o el KDE o Gnome de Linux.

Sin embargo, la interfaz textual es más potente que la gráfica y permite automatizar operaciones (véase ficheros de mandatos a continuación), lo cual es muy interesante para administrar los sistemas.

Ficheros de mandatos

Casi todos los intérpretes de mandatos pueden ejecutar ficheros de mandatos, llamados *shell scripts*. Estos ficheros incluyen varios mandatos totalmente equivalentes a los mandatos que se introducen en el terminal. Además, para realizar funciones complejas, pueden incluir mandatos especiales de control del flujo de ejecución como pueden ser los bucles, las secuencias condicionales o las llamadas a función, así como etiquetas para identificar líneas de mandatos.

Para ejecutar un fichero de mandatos basta con invocarlo de igual forma que un mandato estándar del intérprete de mandatos.

2.1.2. Concepto de usuario y de grupo de usuarios

Como ya se ha visto, un usuario es una persona autorizada a utilizar un sistema informático. El usuario se autentica mediante su nombre de cuenta y su contraseña o *password*. Sin embargo, el sistema operativo no asocia el concepto de usuario con el de persona física sino con un nombre de cuenta. Una persona puede tener más de una cuenta y una cuenta puede ser utilizada por más de una persona. Es más, el usuario puede ser un computador remoto. Internamente, el sistema operativo suele asignar a cada usuario (cuenta) un identificador «uid» (*user identifier*) y un perfil.

El sistema de protección de los sistemas operativos está basado en la entidad usuario. Cada usuario tiene asociados en su perfil unos permisos, que definen las operaciones que le son permitidas.

Existe un usuario privilegiado, denominado **superusuario** o **administrador**, que no tiene ninguna restricción, es decir, que puede hacer todas las operaciones sin ninguna traba. La figura del superusuario es necesaria para poder administrar el sistema (recomendación 2.1).

Recomendación 2.1. La figura del superusuario entraña no pocos riesgos, por su capacidad de acción. Es, por tanto, muy importante que la persona o personas que estén autorizadas a utilizar una cuenta de superusuario sean de toda confianza y que las contraseñas utilizadas sean difíciles de adivinar. Además, como una buena norma de administración de sistemas, siempre se deberá utilizar la cuenta con los menores permisos posibles que permiten realizar la función deseada. De esta forma se minimiza la posibilidad de cometer errores irreparables.

Los usuarios se organizan en grupos (p. ej.: en una universidad se puede crear un grupo para los alumnos de cada curso y otro para los profesores). Todo usuario debe pertenecer a un grupo. Los grupos también se emplean en la protección del sistema, puesto que los derechos de un usuario son los suyos propios más los del grupo al que pertenezca. Internamente se suele asignar un identificador «**gid**» (*group identifier*) a cada grupo.

Aclaración 2.1 No se debe confundir superusuario con modo de ejecución núcleo. **EL SUPERUSUARIO EJECUTA EN MODO USUARIO**, como todos los demás usuarios.

2.1.3. Concepto de proceso y multitarea

En esta sección analizaremos primero el concepto de proceso de una forma simplificada para pasar, seguidamente, al concepto de multitarea o multiproceso.

Concepto simple de proceso

Un proceso se puede definir de forma sencilla como un programa puesto en ejecución por el sistema operativo. Como muestra la figura 2.3, el sistema operativo parte de un fichero ejecutable, guardado en una unidad de almacenamiento secundario. Con ello forma la imagen de memoria del proceso, es decir, ubica en el mapa de memoria el programa y sus datos. Adicionalmente, el sistema operativo establece una estructura de datos con información relevante al proceso.

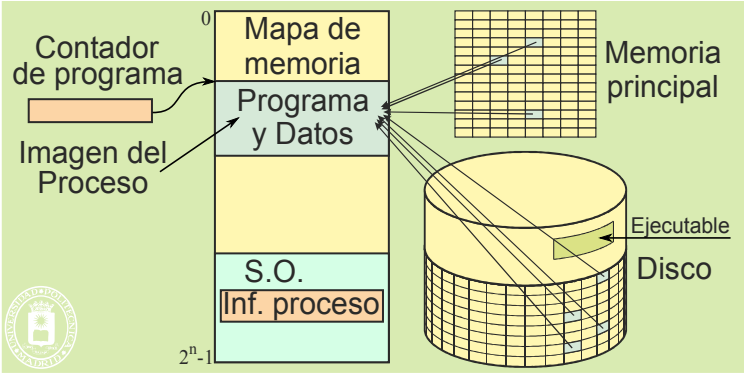


Figura 2.3 *Concepto de proceso.*

La mencionada figura también muestra que, en un sistema con memoria virtual, el soporte físico de la imagen de memoria es una mezcla de trozos de memoria principal y de disco.

El sistema operativo asigna a cada proceso un identificador único, denominado «pid» (***p**rocess **id**entifier*).

Más adelante se tratará con más detalle el concepto de proceso.

Concepto de multitarea

En la ejecución de los procesos alternan **rachas de procesamiento** con **rachas de espera**, que muy frecuentemente consisten en esperar a que se complete una operación de entrada/salida como puede ser que el usuario pulse una tecla o accione el ratón, o que termine una operación sobre el disco. La figura 2.4 muestra gráficamente esta alternancia.

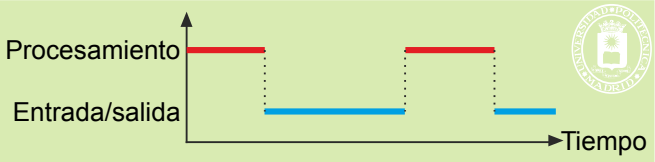


Figura 2.4 *En la ejecución de un proceso se alternan fases de procesamiento con fases de espera, por ejemplo a que termine una operación de entrada/salida.*

Dado que las operaciones de entrada/salida se pueden realizar por un *hardware* más económico que el procesador, hoy en día, estas operaciones se hacen por un *hardware* especializado, dejando al procesador libre. Esto significa que el procesador se puede dedicar a ejecutar otro proceso, aprovechando las rachas de espera del primero.

Para que esto se pueda producir, el sistema operativo tiene que ser capaz de tener activos de forma simultánea varios procesos. Además, todos los procesos activos tienen que tener su imagen de memoria en el mapa de memoria, lo que exige que se tengan suficientes recursos de memoria para soportar a todos ellos.

Se dice que un proceso es **intensivo en procesamiento** cuando tiene largas rachas de procesamiento con pocas rachas de espera. Por el contrario, un proceso es intensivo en entrada/salida cuando tiene poco procesamiento frente a las esperas por entrada/salida. En general, los programas interactivos, que están a la espera de que un usuario introduzca información, suelen ser intensivos en entrada/salida.

2.2. ARRANQUE Y PARADA DEL SISTEMA

El arranque de un computador actual tiene dos fases: la fase de arranque *hardware* y la fase de arranque del sistema operativo. La figura 2.5 resume las actividades más importantes que se realizan en el arranque del computador.

Iniciador ROM

Como se ha indicado con anterioridad, el computador solamente es capaz de realizar actividades útiles si cuenta con el correspondiente programa cargado en memoria principal. Ahora bien, la memoria principal de los computadores es de tipo RAM, que es volátil, lo que significa que, cuando se enciende la máquina, no contiene ninguna información válida. Por tanto, al arrancar el computador no es capaz de realizar nada.

Para resolver esta situación, los computadores antiguos tenían una serie de conmutadores que permitían introducir, una a una, palabras en la memoria principal y en los registros. El usuario debía introducir a mano, y en binario, un primer programa que permitiese cargar otros programas almacenados en algún soporte, como la cinta de papel. En la actualidad, la solución empleada es mucho más cómoda, puesto que se basa en un programa permanente grabado en una memoria ROM no volátil que ocupa, como muestra la figura 2.5, una parte del mapa de memoria. Esta memoria suele estar situada en la parte baja o alta del mapa de memoria, aunque en la arquitectura PC se encuentra al final del primer megabyte del mapa de memoria. En esta memoria ROM hay un programa de arranque, que está siempre disponible, puesto que la ROM no pierde su contenido. Llamaremos **iniciador ROM** a este programa.

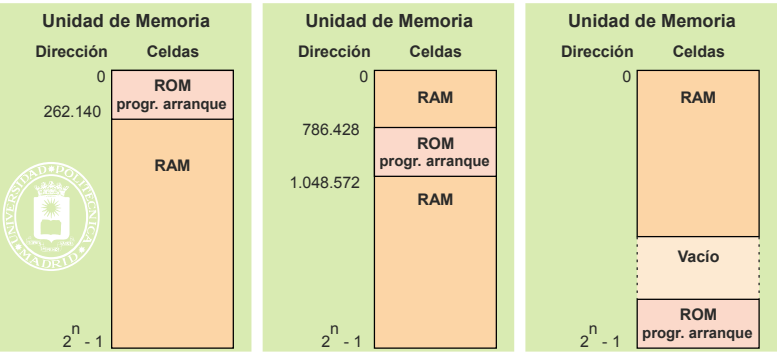


Figura 2.5 *Una parte del mapa memoria del computador está construida con memoria ROM.*

En el caso de un computador de tipo PC, la memoria ROM contiene, además del programa iniciador, *software* de E/S denominado **BIOS** (*Basic Input-Output System*). La BIOS la proporciona el fabricante del computador y suele contener procedimientos para leer y escribir de disco, leer caracteres del teclado, escribir en la pantalla, etc.

Cargador del sistema operativo o *boot*

El sistema operativo se encuentra almacenado en una unidad de almacenamiento como el disco, tal y como muestra la figura 2.6. Hay una parte del mismo en la que estamos especialmente interesados ahora: se trata del programa cargador del sistema operativo o *boot* del sistema operativo (aclaración 2.2). Este programa está almacenado en una zona predefinida del mencionado dispositivo (p. ej.: los cuatro primeros sectores del disco) y tiene un tamaño prefijado. Además, incluye una contraseña (o palabra mágica) que lo identifica como *boot*.

Aclaración 2.2. La operación combinada de leer un programa ubicado en un periférico y de almacenarlo en memoria principal se denomina **carga**. El programa que realiza esta operación se llama **cargador**.

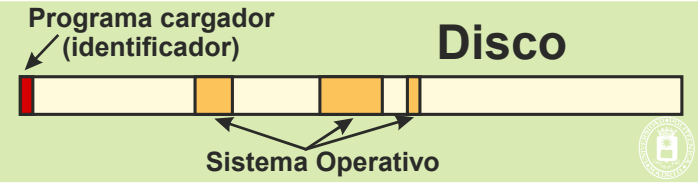


Figura 2.6 *El sistema operativo se encuentra almacenado en una unidad de almacenamiento como el disco duro.*

2.2.1. Arranque *hardware*

Cuando se arranca el computador, o cuando se pulsa el botón de RESET, se genera una señal eléctrica que carga unos valores predefinidos en los registros. En especial, esta señal carga en el contador de programa la dirección de comienzo del iniciador ROM. De esta forma, se cumplen todas las condiciones para que el computador ejecute un programa y realice funciones útiles. Adicionalmente, el RESET pone el procesador en modo privilegiado, por lo que puede utilizar el juego completo de instrucciones máquina, en modo real, es decir, sin memoria virtual (la MMU queda desactivada), y con las interrupciones inhibidas.

El iniciador ROM realiza las tres funciones siguientes.

- Primero hace una comprobación del sistema, que sirve para detectar sus características (p. ej.: la cantidad de memoria principal disponible o los periféricos instalados) y comprobar si el conjunto funciona correctamente.
- Una vez pasada esta comprobación, entra en la fase de lectura y almacenamiento en memoria del programa cargador del sistema operativo o *boot*. El programa iniciador ROM y el sistema operativo tienen un convenio sobre la ubicación, dirección de arranque, contraseña y tamaño del cargador del sistema operativo. De esta forma, el iniciador ROM puede leer y verificar que la información contenida en la zona prefijada contiene efectivamente el programa cargador de un sistema operativo. Observe que el iniciador ROM es independiente del sistema operativo, siempre que éste cumpla con el convenio anterior, por lo que la máquina podrá soportar diversos sistemas operativos.
- Finalmente, da control al programa *boot*, bifurcando a la dirección de memoria en la que lo ha almacenado. Observe que se sigue ejecutando en modo privilegiado y modo real.

2.2.2. Arranque del sistema operativo

El programa *boot* tiene por misión traer a memoria principal y ejecutar el programa de arranque del sistema operativo, que incluye las siguientes operaciones:

- Comprobación del sistema. Se completan las pruebas del *hardware* realizadas por el iniciador ROM y se comprueba que el sistema de ficheros tiene un estado coherente. Esta operación exige revisar todos los directorios, lo que supone un largo tiempo de procesamiento. Por esta razón, los sistemas operativos para máquinas personales solamente realizan esta revisión si se apagó mal el sistema.
- Se carga en memoria principal aquella parte del sistema operativo que ha de estar siempre en memoria, parte que se denomina sistema operativo residente.
- Se establecen las estructuras de información propias del sistema operativo, tales como la tabla de interrupciones IDT, la tabla de procesos, las tablas de memoria y las de E/S. El contenido de estas tablas se describirá a lo largo del libro.
- En su caso, creadas las tablas de páginas necesarias, se activa la MMU, pasando a modo virtual.
- Se habilitan las interrupciones.
- Se crea un proceso de inicio o *login* por cada terminal definido en el sistema, así como una serie de procesos auxiliares y de demonios como el de impresión o el de comunicaciones (véase la sección “3.7.2 Demonio”, página 87). Dichos procesos ya ejecutan en modo usuario.

En el caso de los sistemas operativos tipo UNIX el arranque del mismo solamente crea un proceso denominado INIT, estando este proceso encargado de crear los procesos de inicio y los procesos auxiliares.

Los procesos de inicio muestran en su terminal el mensaje de bienvenida y se quedan a la espera de que un usuario arranque una sesión, para lo cual ha de teclear el nombre de su cuenta y su contraseña o *password*. El proceso de inicio **autentica** al usuario, comprobando que los datos introducidos son correctos y lanza un proceso *shell*.

Suele existir algún mecanismo para que el usuario pueda establecer sus preferencias, por ejemplo, el *shell* puede primero ejecutar uno o varios ficheros de mandatos, como es el «`autoexec.bat`» en MS-DOS o los «`.profile`» y «`.bashrc`» en UNIX. A continuación, el *shell* se queda esperando órdenes de los usuarios, ya sean textuales o acciones sobre un menú o un icono. Es frecuente que el *shell* genere uno o varios procesos para llevar a cabo cada operación solicitada por el usuario.

En algunos de los sistemas operativos monousuario utilizados en los computadores personales no hay fase de *login*, creándose directamente el proceso *shell* para atender al usuario.

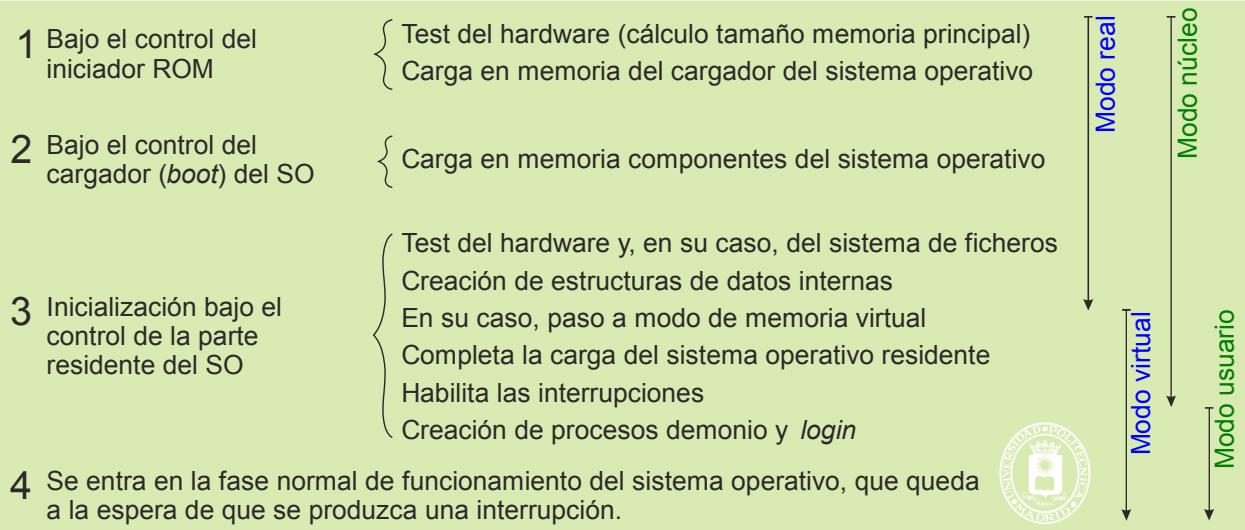


Figura 2.7 *Operaciones realizadas en el arranque del computador y situación del procesador.*

2.2.3. Parada del computador

Para garantizar una alta velocidad, el sistema operativo mantiene en memoria principal una gran cantidad de información crítica, entre la que cabe destacar parte de los ficheros que se están utilizando. Para que toda esa información no se pierda o corrompa al apagar el computador, el sistema operativo debe proceder a un apagado ordenado, que consiste en copiar a disco toda la información crítica que mantiene en memoria y en terminar todos los procesos activos. Este apagado ordenado lleva un cierto tiempo, pero es imprescindible para que no se produzcan pérdidas de información.

Cuando se produce un apagón brusco, por ejemplo, porque se interrumpe la alimentación o porque se pulsa el RESET del computador, se corre el riesgo de perder información importante y de que el sistema de ficheros quede inconsistente. Por esta razón, después de un apagado brusco se debe comprobar siempre la consistencia del sistema de ficheros y se deben reparar los posibles problemas detectados.

En los sistemas multiusuario el apagado del computador debe anunciarse con suficiente antelación para que los usuarios cierren sus aplicaciones y salven en disco su información.

En los computadores personales se puede **hibernar** el sistema. Esta operación se diferencia del apagado en que los procesos no se cierran, simplemente se hace una copia a disco de toda la memoria principal. Cuando se vuelve a arrancar, se recupera esta copia, quedando el computador en el estado que tenía antes de hibernar, con todos los procesos activos. La hibernación y rearranque posterior suelen ser bastante más rápidas que el apagado y arranque normal.

Los computadores portátiles suelen incorporar una función de **apagado en espera** (*standby*). En este modo se para todo el computador, pero se mantiene alimentada la memoria principal, de forma que conserve toda su información. Se trata de un apagado parecido a la hibernación pero que es mucho más rápido, puesto que no se copia la información al disco, pero se necesita una batería para mantener la memoria alimentada. Del apagado en espera se sale mediante una interrupción, como puede ser de un temporizador o batería baja. Este apagado no se puede utilizar, por ejemplo, en los aviones, puesto que el computador puede despertar, aunque esté cerrado, violando las normas de aviación civil.

2.3. ACTIVACIÓN DEL SISTEMA OPERATIVO

Una vez finalizada la fase de arranque, el sistema operativo cede la iniciativa a los procesos y a los periféricos, que, mediante interrupciones, solicitarán su atención y servicios. Desde ese momento, el sistema operativo está «dormido» y solamente se pone en ejecución, se «despierta», mediante una interrupción.

El sistema operativo es, por tanto, un servidor que está a la espera de que se le encargue trabajo mediante interrupciones. Este trabajo puede provenir de las siguientes fuentes:

- Llamadas al sistema emitidas por los programas mediante la instrucción máquina TRAP o de llamada al sistema. Esta instrucción genera un ciclo de interrupción por lo que pone al procesador en modo privilegiado. Observe que los servicios del sistema operativo no se pueden solicitar mediante una instrucción máquina normal de llamada a procedimiento o función. Estas instrucciones no cambian el modo de ejecución, por lo que no sirven para activar el sistema operativo.
- Interrupciones externas. Se considerarán tres tipos de interrupciones: de E/S, de reloj y de otro procesador. Las de E/S están producidas por los periféricos para indicar, por ejemplo, que tienen o necesitan un dato, que la operación ha terminado o que tienen algún problema.
- Excepciones *hardware* síncronas (véase sección "1.3 Interrupciones", página 14).
- Excepciones *hardware* asíncronas (véase sección "1.3 Interrupciones", página 14).

Las llamadas al sistema y las excepciones *hardware* síncronas son interrupciones síncronas, puesto que las produce directa o indirectamente el programa, mientras que el resto son asíncronas. En todos los casos, el efecto de la interrupción es que se entra a ejecutar el sistema operativo en modo privilegiado.

La secuencia simplificada de activación del sistema operativo es la mostrada en la figura 2.8.

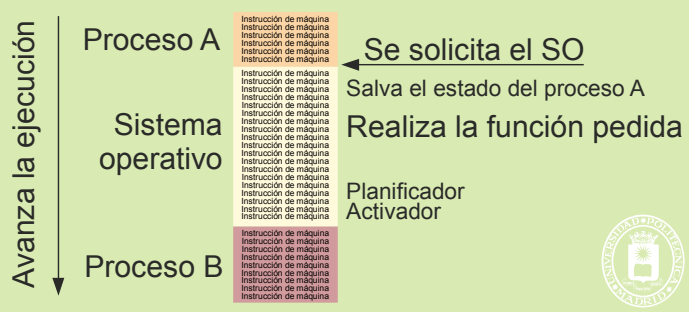


Figura 2.8 *Modelo simplificado de la activación del sistema operativo.*

Dicha secuencia consta de los siguientes pasos:

- Está ejecutando un proceso A y, en un instante determinado, se genera una interrupción para solicitar la atención del sistema operativo. Dicha interrupción pone el computador en modo privilegiado.
- El sistema operativo entra en ejecución y **salva el estado** del proceso A.
- Seguidamente, el sistema operativo realiza la tarea solicitada.
- Una vez finalizada la tarea, entra en acción el **planificador**, módulo del sistema operativo que selecciona un proceso B para ejecutar (que puede volver a ser el A).
- La actuación del sistema operativo finaliza con el **activador**, módulo que se encarga de restituir los registros con los valores previamente almacenados del proceso B y de poner el computador en modo usuario. El instante en el que el activador restituye el contador de programa marca la transición del sistema operativo a la ejecución del proceso B.

Más adelante se completará este modelo para tener en cuenta aspectos de diseño del sistema operativo.

2.3.1. Servicios del sistema operativo y funciones de llamada

Los servicios del sistema operativo forman un puente entre el espacio de usuario y el espacio de núcleo. También son el puente entre las aplicaciones de usuario y el *hardware*, puesto que es la única forma que tienen éstas de acceder al *hardware*. Cada servicio tiene su propio número identificador interno.

En esta sección se tratará primero la ejecución de los servicios, para pasar seguidamente a analizar las funciones de biblioteca.

Ejecución de los servicios

Cuando el sistema operativo recibe la solicitud de un servicio pueden ocurrir dos cosas: que el servicio se pueda realizar de un tirón o que requiera acceder a un recurso lento, como puede ser un disco, por lo que requiere una o varias esperas. Lógicamente, el sistema operativo no hará una espera activa y pondrá en ejecución otros procesos durante las esperas, lo que significa que el servicio no se realiza de un tirón, requiriendo varias fases de ejecución.

La figura 2.9 muestra la ejecución del servicio en el caso de requerir una sola fase y en el caso de requerir dos. En este último se puede observar que, una vez completada la primera fase del servicio, se pasa a ejecutar otro proceso, que puede ser interrumpido o puede, a su vez solicitar un servicio. Cuando llega el evento por el que está esperando el servicio A (interrupción A), se ejecuta la segunda fase del servicio A.

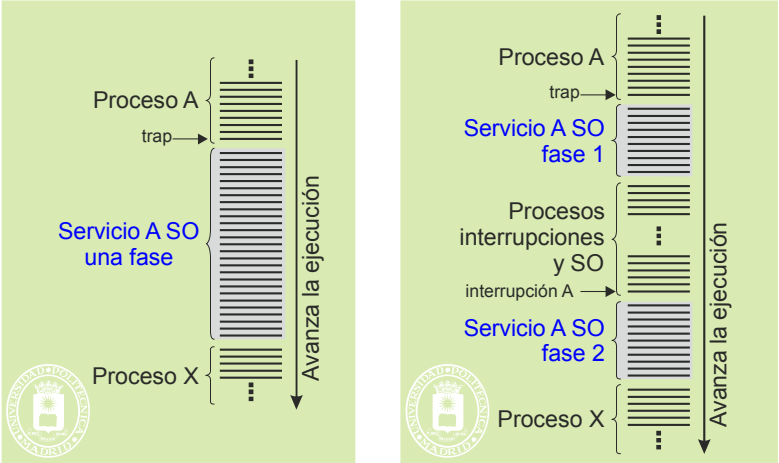


Figura 2.9 *Ejecución del servicio del sistema operativo en una y dos fases.*

Se dice que el servicio es **síncrono** cuando el proceso que lo solicita queda bloqueado hasta que se completa el servicio (véase figura 2.10). Por el contrario, se dice que el servicio es **asíncrono** si el proceso que lo solicita puede seguir ejecutando aunque éste no se haya completado. La utilización de servicios asíncronos por parte del programador es bastante más compleja que la de los síncronos, puesto que el trozo de programa que sigue a la solicitud del servicio conoce el resultado del mismo.

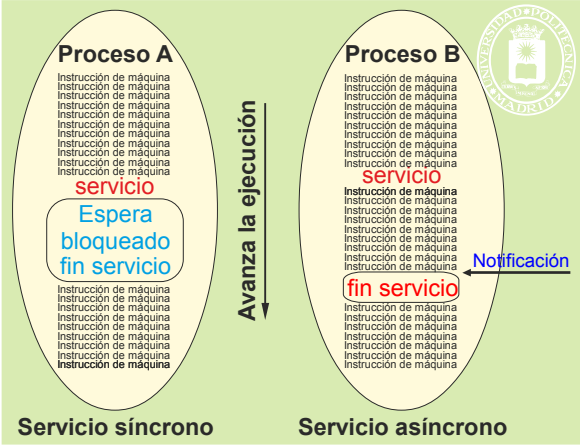


Figura 2.10 *Los servicios del sistema operativo pueden ser síncronos o asíncronos. En el caso de servicios síncronos el proceso queda bloqueado hasta que se completa el servicio. Por el contrario, en los asíncronos el proceso puede seguir ejecutando mientras se realiza el servicio.*

Sintaxis de la llamada al servicio. Funciones de biblioteca

Los lenguajes de alto nivel incluyen funciones para realizar las llamadas al sistema operativo y facilitar, así, la tarea del programador. Por ejemplo, para crear en lenguaje C un nuevo proceso de acuerdo al API de UNIX se hace la llamada a función `n = fork()`. No hay que confundir esta llamada con el servicio del sistema operativo. La función `fork()` del lenguaje C no realiza ella misma el servicio `fork`, se lo solicita al sistema operativo. Las bibliotecas de cada lenguaje incluyen estas funciones.

Estas bibliotecas son específicas para cada tipo de API de sistema operativo (Win32, Win64, UNIX, ...)

En el lenguaje C la sintaxis de la llamada a la función de biblioteca es la siguiente:

mivariable = servicio(parámetro 1, parámetro 2, etc.);

En muchos casos los parámetros de la función de biblioteca son referencias a estructuras que recibe y/o modifica el SO.

Estas funciones se encuentran en las bibliotecas del lenguaje y no deben ser confundidas con otras funciones del lenguaje que no llaman al sistema operativo, como pueden ser las de tratamiento de cadenas.

La figura 2.11, muestra la estructura de la función de biblioteca que llama al sistema operativo, compuesta por:

- Una parte inicial que prepara el código y los argumentos del servicio según los espera el sistema operativo.
- La instrucción máquina TRAP de llamada al sistema, que realiza el paso al sistema operativo (el sistema operativo completará el servicio, lo que puede suponer varias fases de ejecución).
- Una parte final (que se ejecuta una vez completado el servicio) y que recupera los parámetros de contestación del sistema operativo, para devolverlos al programa que le llamó.

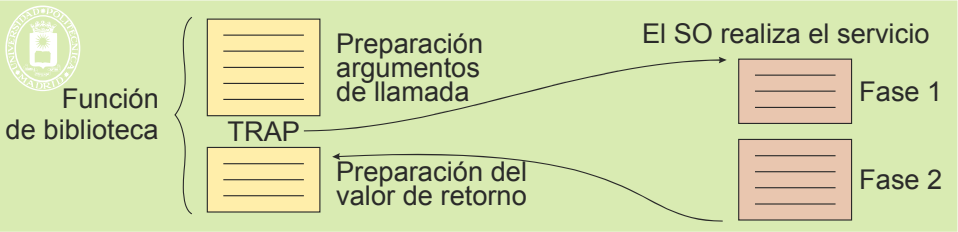


Figura 2.11 *Estructura de una rutina de biblioteca para llamar a un servicio del sistema operativo.*

Para completar la imagen de que se está llamando a una función, el sistema operativo devuelve un valor, imitando a una función normal. Al programador le parece, por tanto, que invoca al sistema operativo como a una función. Sin embargo, esto no es así, puesto que lo que hace es invocar una función que realiza la solicitud al sistema operativo.

En UNIX el servicio devuelve generalmente un entero. El valor -1 significa que el servicio ha fracasado, lo que puede estar causado por las siguientes condiciones:

- Porque se produce un error al intentar el sistema operativo ejecutar el servicio.
- Porque le llega una señal al proceso (`errno = EINTR`), lo que causa que se termine el servicio.

Además, existe una variable global `errno`, que indica el tipo de error que se ha producido. Una breve descripción de cada tipo de error se encuentra en el fichero `errno.h` (la función `perror` sirve para producir un mensaje con la descripción del error contenido en `errno`).

Dado que casi todos los servicios del SO pueden fallar por falta de recursos o por falta de privilegios, en un programa profesional es imprescindible incluir código que trate el caso de fracaso de todos y cada uno de los servicios utilizados.

Veamos, con un ejemplo, cómo se produce una llamada al sistema. Supondremos que en lenguaje C se escribe la siguiente sentencia:

n = read(fd1, buf3, 120);

El bloque de activación de la función de C `read` está formado por las palabras siguientes:

PILA

Puntero a marco anterior	Palabra que almacena el valor del puntero del marco de pila anterior
Dirección retorno de `read`	Palabra donde se guarda la dirección de retorno de la función `read`
Valor `fd1`	Palabra donde se copia el valor de `fd1` (argumento pasado por valor)
Dirección del *buffer* `buf3`	Palabra donde se copia la dirección de `buf3`
120	Palabra donde se copia el valor `120` (argumento pasado por valor)

Suponiendo a) que el registro BP contiene el puntero de marco, b) que la máquina es de 32 bits, c) que la pila crece hacia direcciones menos significativas, d) que el sistema operativo recibe los argumentos en registros e) que el sistema operativo devuelve en R9 el código de terminación y f) que las funciones C devuelven el valor de la misma también en R9, una versión simplificada del cuerpo de la función de C `read`, sería la incluida en el programa 2.1.

Programa 2.1 Ejemplo simplificado de la función de biblioteca `read`.

```
int read() {
    PUSH    .R3                         ;Se salvan los registros que usa la función
    PUSH    .R4
    PUSH    .R5
    PUSH    .R6
                                        ;Almacena en R3 el identificador de READ
    LOAD    .R3, #READ  SYSTEM  CALL
    LOAD    .R4, #8[.BP]                ;Almacena en R4 el Valor fd1
    LOAD    .R5, #12[.BP]               ;Almacena en R5 la dirección del buffer buf3
    LOAD    .R6, #16[.BP]               ;Almacena en R6 el valor 120
    TRAP                                ;Instrucción de llamada al sistema operativo
    CMP   .R9, #0                       ;Se compara R9 con 0
      ;Si R9<0 es que el SO devuelve error. Se cambia de signo el
      ;valor devuelto y se almacena en la variable global errno
    BNC     $FIN                        ;Si r9 >= 0, no hay error y se salta a FIN
    SUB   .R6, .R6                      ;Se pone R6 a cero
    SUB   .R6, .R9                      ;Se hace R6 = - R9
    ST    .R6, /ERRNO                   ;Se almacena el tipo de error en errno
    LOAD    .R9, #-1                    ;Se hace R9 = - 1
FIN: POP  .R6                           ;Se restituyen los registros que usa la función
    POP   .R5
    POP   .R4
```

```
    POP     .R3
    RETURN
}
```

Se puede observar que la función copia los argumentos de la llamada y el identificador de servicio a registros y ejecuta una instrucción TRAP de llamada al sistema operativo. Se podría argumentar que la copia de los argumentos a registros es innecesaria, puesto que el sistema operativo podría tomarlos de la pila del proceso. Esto tiene, sin embargo, un problema: cada lenguaje estructura el bloque de activación de las funciones a su manera, por lo que el sistema operativo no sabe las posiciones de la pila donde se encuentran los argumentos. En algunos sistemas como Windows la rutina de biblioteca (que es específica del lenguaje) pasa en un registro la dirección de pila donde comienzan los argumentos, por lo que éstos no se copian a registros.

El programa llamante, que ha ejecutado la sentencia `n = read(fd1, buf3, 120);`, ha de copiar el valor de retorno desde el registro R9 a su variable `n`.

2.4. TIPOS DE SISTEMAS OPERATIVOS

Existe una gran diversidad de sistemas operativos diseñados para cubrir las necesidades de los distintos dispositivos y de los distintos usos. Dependiendo de sus características, un sistema operativo puede ser:

- Según el número de procesos simultáneos que permita ejecutar: monotarea o monoproceso y multitarea o multiproceso.
- Según la forma de interacción con el usuario: interactivo o por lotes.
- Según el número de usuarios simultáneos: monousuario o personal y multiusuario o de tiempo compartido.
- Según el número de procesadores que pueda atender: monoprocesador y multiprocesador.
- Según el número de *threads* que soporte por proceso: monothread y multithread, (véase sección “3.8 Threads”).
- Según el uso: cliente, servidor, empotrado, de comunicaciones o de tiempo real.
- Según la movilidad: fijos y móviles.

Estas clasificaciones no son excluyentes, así un sistema operativo servidor será generalmente también multiprocesador, multithread y multiusuario.

Un sistema operativo **monotarea**, también llamado monoproceso, solamente permite que exista un proceso en cada instante. Si se quieren ejecutar varias procesos, o tareas, hay que lanzar la ejecución de la primera y esperar a que termine antes de poder lanzar la siguiente. El ejemplo típico de sistema operativo monoproceso es el MS-DOS, utilizado en los primeros computadores PC. La ventaja de estos sistemas operativos es que son muy sencillos.

Por el contrario, un sistema operativo **multitarea,** o multiproceso (recordatorio 2.1), permite que coexistan varios procesos activos a la vez. El sistema operativo se encarga de ir repartiendo el tiempo del procesador entre estos procesos, para que todos ellos vayan avanzando en su ejecución.

Recordatorio 2.1. No confundir el término multiproceso con multiprocesador. El término multiproceso se refiere a los sistemas que permiten que existan varios procesos activos al mismo tiempo, mientras que el término multiprocesador se refiere a un computador con varios procesadores. El multiprocesador exige un sistema operativo capaz de gestionar simultáneamente todos sus procesadores, cada uno de los cuales estará ejecutando su propio proceso.

Un sistema **interactivo** permite que el usuario dialogue con los procesos a través, por ejemplo, de un terminal. Por el contrario, en un sistema por **lotes** o ***batch*** se parte de una cola de trabajos que el sistema va ejecutando cuando tiene tiempo y sin ningún diálogo con el usuario.

Un sistema **monousuario,** o personal, está previsto para soportar a un solo usuario interactivo. Estos sistemas pueden ser monoproceso o multiproceso. En este último caso el usuario puede solicitar varias tareas al mismo tiempo, por ejemplo, puede estar editando un fichero y, simultáneamente, puede estar accediendo a una página Web.

El sistema operativo **multiusuario** es un sistema interactivo que da soporte a varios usuarios, que trabajan simultáneamente desde varios terminales locales o remotos. A su vez, cada usuario puede tener activos más de un proceso, por lo que el sistema, obligatoriamente, ha de ser multitarea. Los sistemas multiusuario reciben también el nombre de **tiempo compartido**, puesto que el sistema operativo ha de repartir el tiempo del procesador entre los usuarios, para que las tareas de todos ellos avancen de forma razonable.

La figura 2.12 recoge estas alternativas.

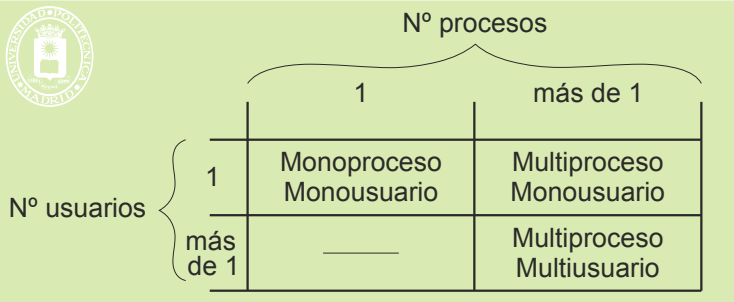


Figura 2.12 *Tipos de sistemas operativos en función del número de procesos y usuarios.*

Un sistema operativo **servidor** está optimizado para que sus usuarios sean sistemas remotos. Por el contrario, un sistema operativo **cliente** es un sistema operativo personal diseñado para poder conectarse a servidores. Los sistemas operativos **personales** interactivos incluyen soporte gráfico, para construir interfaces gráficas (GUI *Graphical User Interface*), con el objetivo de facilitar la interacción con el usuario, así como herramientas que permitan gestionar con facilidad el sistema. Existen versiones de Windows y de Linux con perfil servidor y con perfil personal.

Los sistemas operativos **empotrados** interaccionan con un sistema físico y no con un usuario. Suelen ejecutar en plataformas con poca memoria, poca potencia de proceso y sin disco duro, por lo que suelen ser sencillos, limitándose a las funciones imprescindibles para la aplicación. Se almacenan en memoria ROM y en muchos casos no cuentan con servidor de ficheros ni interfaz de usuario.

Los sistemas operativos de **tiempo real** permiten garantizar que los procesos ejecuten en un tiempo predeterminado, para reaccionar adecuadamente a las necesidades del sistema, como puede ser el guiado de un misil o el control de una central eléctrica. Con gran frecuencia entran también en la categoría de sistemas operativos empotrados. Como ejemplos se puede citar el VxWorks de Wind River o el RTEMS de OAR.

Para atender las peculiaridades de los dispositivos móviles (PDA, teléfono inteligente, *pocket* PC, etc.) existen una serie de sistemas operativos **móviles**. Tienen un corte parecido a los destinados a los computadores personales, pero simplificados, para adecuarse a estos entornos. Además, están previstos para que el dispositivo se encienda y apague con frecuenta y para reducir al máximo el consumo de las baterías. Ejemplos son las familias Android, PALM OS, Windows CE, Windows Mobile y Symbian OS, este último muy utilizado en los teléfonos móviles.

Para las tarjetas inteligentes se construyen sistemas operativos empotrados muy simples pero con funcionalidades criptográficas. Ejemplos son MULTOS, SOLO (de Schlumberger) y Sun's JavaCard.

2.5. COMPONENTES DEL SISTEMA OPERATIVO

El sistema operativo está formado por una serie de componentes especializados en determinadas funciones. Cada sistema operativo estructura estos componentes de forma distinta. En una visión muy general, como se muestra en la figura 2.13, se suele considerar que un sistema operativo está formado por tres capas: el núcleo, los servicios y el intérprete de mandatos o *shell*.

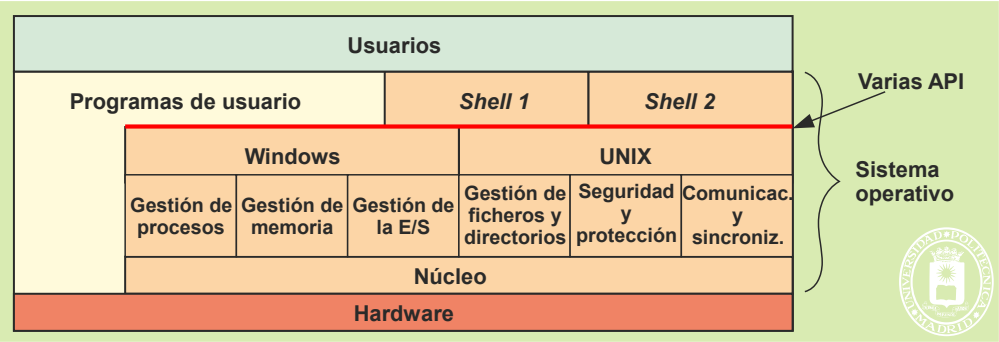


Figura 2.13 *Componentes del sistema operativo.*

El núcleo es la parte del sistema operativo que interacciona directamente con el *hardware* de la máquina. Las funciones del núcleo se centran en la gestión de recursos, como es el procesador, tratamiento de interrupciones y las funciones básicas de manipulación de memoria.

Los servicios se suelen agrupar según su funcionalidad en varios componentes, como los siguientes:

- Gestor de procesos. Encargado de la creación, planificación y destrucción de procesos.
- Gestor de memoria. Componente encargado de saber qué partes de la memoria están libres y cuáles ocupadas, así como de la asignación y liberación de memoria según la necesiten los procesos.
- Gestor de la E/S. Se ocupa de facilitar el manejo de los dispositivos periféricos.
- Gestor de ficheros y directorios. Se encarga del manejo de ficheros y directorios, y de la administración del almacenamiento secundario.

- Gestor de comunicación y sincronización entre procesos. Ofrecer mecanismos para que los procesos puedan comunicarse y sincronizarse.
- Gestor de seguridad frente a ataques del exterior y protección interna. Este componente debe encargarse de realizar la identificación de los usuarios, de definir lo que pueden hacer cada uno de ellos con los recursos del sistema y de controlar el acceso a estos recursos.

Todos estos componentes ofrecen una serie de servicios a través de una interfaz de llamadas al sistema. Aunque no es muy frecuente, la figura 2.13 muestra que un sistema operativo puede incluir más de una interfaz de servicios, definiendo cada interfaz una máquina extendida propia. En la figura se han considerado las interfaces Windows y UNIX, interfaces que serán descritas a lo largo del presente libro. En este caso, los programas podrán elegir sobre qué máquina extendida quieren ejecutar, pero no podrán mezclar servicios de varias máquinas extendidas.

De igual forma, el sistema operativo puede incluir varios intérpretes de mandatos, unos textuales y otros gráficos, pudiendo el usuario elegir los que más le interesen, debiendo utilizar, en cada caso, los mandatos correspondientes.

En las secciones siguientes de este capítulo se van a describir, de forma muy breve, cada uno de los componentes anteriores.

2.5.1. Gestión de procesos

Como se indicó anteriormente, proceso es un programa en ejecución. De una forma más precisa, se puede definir el proceso como la unidad de procesamiento gestionada por el sistema operativo. No hay que confundir el concepto de programa con el concepto de proceso. Un programa no es más que un conjunto de instrucciones máquina, mientras que el proceso surge cuando un programa se pone en ejecución. Esto hace que varios procesos puedan ejecutar el mismo programa a la vez (por ejemplo, que varios usuarios estén ejecutando el mismo navegador). Para que un programa se pueda ejecutar tiene que estar preparado en un fichero ejecutable, que contiene el código y algunos datos iniciales (véase figura 4.27, página 166).

Dado que un computador está destinado a ejecutar programas, podemos decir que la misión más importante del sistema operativo es la generación de procesos y la gestión de los mismos.

Para ejecutar un programa éste ha de residir, junto con sus datos, en el mapa de memoria principal, tal y como muestra la figura 2.14. Se denomina **imagen de memoria** a la información que mantiene el proceso en el mapa de memoria.

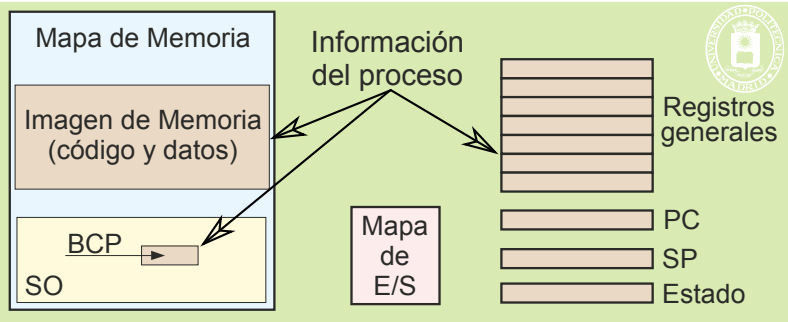


Figura 2.14 *Elementos que constituyen un proceso.*

Durante su ejecución, el proceso va modificando los contenidos de los registros del computador, es decir, va modificando el **estado del procesador**. También va modificando los datos que tiene en memoria, es decir, su imagen de memoria.

El sistema operativo mantiene, por cada proceso, una serie de estructuras de información, que permiten identificar las características de éste, así como los recursos que tiene asignados. Una parte muy importante de esta información está en el **bloque de control del proceso** (**BCP**), que se estudiará con detalle en la sección "3.3.3 Información del bloque de control de proceso (BCP)". El sistema operativo debe encargarse también de ofrecer una serie de servicios para la gestión de procesos y para su planificación, así como para gestionar los posibles interbloqueos que surgen cuando los procesos acceden a los mismos recursos.

Una buena parte de la información del proceso se obtiene del ejecutable, pero otra la produce el sistema operativo. A diferencia del ejecutable, que es permanente, la información del proceso es temporal y desaparece con el mismo. Decimos que el proceso es volátil mientras que el ejecutable perdura hasta que se borre el fichero.

En el capítulo "3 Procesos" se estudiarán en detalle los procesos y en el capítulo "6 Comunicación y sincronización de procesos" se estudiarán los interbloqueos y los mecanismos para manejarlos.

Servicios de procesos

El sistema operativo ofrece una serie de servicios que permiten definir la vida de un proceso, vida que consta de las siguientes fases: creación, ejecución y muerte del proceso, que se analizan seguidamente:

- **Crear un proceso**. El proceso es creado por el sistema operativo cuando así lo solicita otro proceso, que se convierte en el padre del nuevo. Existen dos modalidades básicas para crear un proceso en los sistemas operativos:

- Creación a partir de la imagen del proceso padre. En este caso, el proceso hijo es una copia exacta o clon del proceso padre. Esta variante es la que utiliza el servicio `fork` de UNIX.
- Creación a partir de un fichero ejecutable. Esta modalidad es la que se utiliza en el servicio `CreateProcess` de Windows.

- **Ejecutar un proceso**. Los procesos se pueden ejecutar de tres formas: *batch*, interactiva y segundo plano.
  - Un proceso que ejecuta en modo de **lotes** o ***batch***, no está asociado a ningún terminal. Deberá tomar sus datos de entrada de ficheros y deberá depositar sus resultados en ficheros. Un ejemplo típico de un proceso *batch* es un proceso de nóminas, que parte del fichero de empleados y del fichero de los partes de trabajo para generar un fichero de órdenes bancarias de pago de nóminas.
  - Por el contrario, un proceso que ejecuta en modo **interactivo** está asociado a un terminal, por el que recibe la información del usuario y por el que contesta con los resultados. Un ejemplo típico de un proceso interactivo es un proceso de edición.
  - Los sistemas interactivos permiten lanzar procesos en segundo plano o *background*. Se trata de procesos similares a los de lotes, que no están asociados a ningún terminal.
- **Terminar la ejecución de un proceso**. Un proceso puede finalizar su ejecución por varias causas, entre las que se encuentran las siguientes:
  - El programa ha llegado a su final.
  - Se produce una condición de error en su ejecución, como división por cero o acceso de memoria no permitido.
  - Otro proceso o el usuario decide que ha de terminar y lo mata, por ejemplo, con el servicio `kill` de UNIX.
- **Cambiar el ejecutable de un proceso.** Algunos sistemas operativos incluyen un servicio que cambia, por otro, el ejecutable que está ejecutando un proceso. Observe que esta operación no consiste en crear un nuevo proceso que ejecuta ese nuevo ejecutable, se trata de sustituir el ejecutable que está ejecutando el proceso por un nuevo ejecutable que se trae del disco, manteniendo el mismo proceso. El servicio `exec` de UNIX realiza esta función.

2.5.2. Gestión de memoria

El componente del sistema operativo llamado gestor de memoria se encarga de:

- Asignar memoria a los procesos para crear su imagen de memoria.
- Proporcionar memoria a los procesos cuando la soliciten y liberarla cuando así lo requieran.
- Tratar los errores de acceso a memoria, evitando que unos procesos interfieran en la memoria de otros.
- Permitir que los procesos puedan compartir memoria entre ellos. De esta forma los procesos podrán comunicarse entre ellos.
- Gestionar la jerarquía de memoria y tratar los fallos de página en los sistemas con memoria virtual.

Servicios

Además de las funciones vistas anteriormente, el gestor de memoria ofrece los siguientes servicios:

- **Solicitar memoria**. Este servicio aumenta el espacio de la imagen de memoria del proceso. El sistema operativo satisfará la petición siempre y cuando cuente con los recursos necesarios para ello y no se exceda la cuota en caso de estar establecida. En general, el sistema operativo devuelve un apuntador con la dirección de la nueva memoria. El programa utilizará este nuevo espacio a través del mencionado apuntador, mediante direccionamientos relativos al mismo.
- **Liberar memoria**. Este servicio sirve para devolver trozos de la memoria del proceso. El sistema operativo recupera el recurso liberado y lo añade a sus listas de recursos libres, para su posterior reutilización. Este servicio y el de solicitar memoria son necesarios para los programas que requieren asignación dinámica de memoria, puesto que su imagen de memoria ha de crecer o decrecer de acuerdo a las necesidades de ejecución.
- **Compartir memoria**. Son los servicios de crear y liberar regiones de memoria compartidas, lo que permite que los procesos puedan comunicarse escribiendo y leyendo en ellas.

El gestor de memoria establece la imagen de memoria de los procesos. La figura 2.15 muestra algunas soluciones para sistemas reales y virtuales.

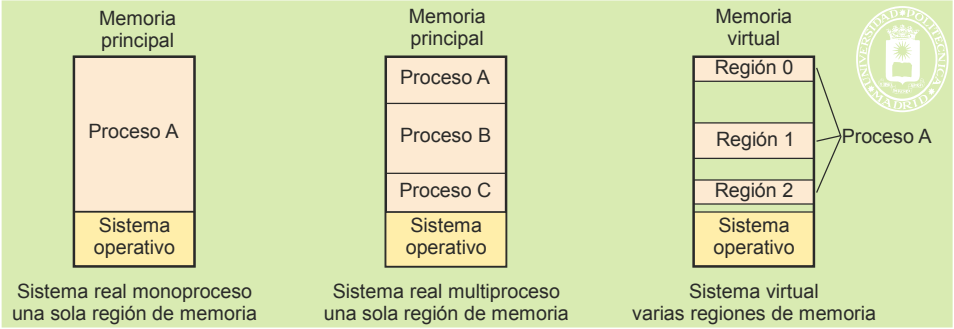


Figura 2.15 *Distintas alternativas de asignación de memoria.*

En el capítulo "4 Gestión de memoria" se estudiarán los conceptos relativos a la gestión de memoria, los servicios ofrecidos por el gestor de memoria y las técnicas de gestión de memoria.

2.5.3. Comunicación y sincronización entre procesos

Los procesos son entes independientes y aislados, puesto que, por razones de seguridad, no deben interferir unos con otros. Sin embargo, cuando se divide un trabajo complejo en varios procesos que cooperan entre sí para realizar dicho trabajo, es necesario que se comuniquen, para transmitirse datos y órdenes, y que se sincronicen en la ejecución de sus acciones. Por tanto, el sistema operativo debe incluir servicios de comunicación y sincronización entre procesos que, sin romper los esquemas de protección, han de permitir la cooperación entre ellos.

El sistema operativo ofrece una serie de mecanismos básicos de comunicación que permiten transferir cadenas de bytes, pero han de ser los procesos que se comunican entre sí los que han de interpretar las cadenas de bytes transferidas. En este sentido, se han de poner de acuerdo en la longitud de la información y en los tipos de datos utilizados. Dependiendo del servicio utilizado, la comunicación se limita a los procesos de una máquina (procesos locales) o puede involucrar a procesos de máquinas distintas (procesos remotos). La figura 2.16 muestra ambas situaciones.

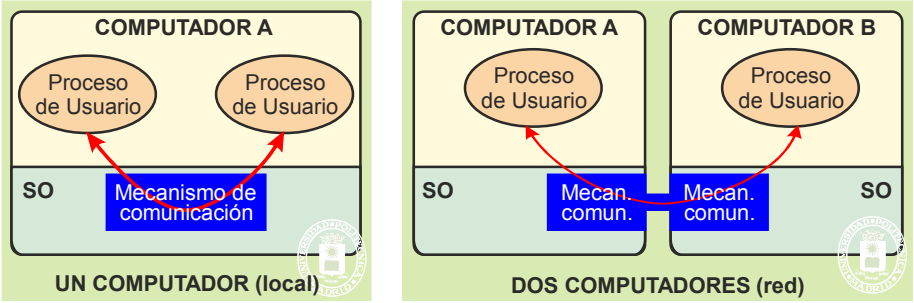


Figura 2.16 *Comunicación entre procesos locales y remotos.*

El sistema operativo ofrece también mecanismos que permiten que los procesos esperen (se bloqueen) y se despierten (continúen su ejecución) dependiendo de determinados eventos.

Servicios de comunicación y sincronización

Existen distintos mecanismos de comunicación (que en muchos casos también sirven para sincronizar), como son las tuberías o *pipes*, la memoria compartida y los *sockets*. Cada uno de estos mecanismos se puede utilizar a través de un conjunto de servicios propios. Estos mecanismos son entidades vivas, cuya vida presenta las siguientes fases: creación del mecanismo, utilización del mecanismo y destrucción del mecanismo. De acuerdo con esto, los servicios básicos de comunicación, que incluyen todos los mecanismos de comunicación, son los siguientes:

- **Crear**. Permite que el proceso solicite la creación del mecanismo. Ejemplo `pipe` de UNIX y `CreatePipe` de Windows.
- **Enviar** o **escribir**. Permite que el proceso emisor envíe información a otro proceso. Ejemplo `write` de UNIX y `WriteFile` de Windows.
- **Recibir** o **leer**. Permite que el proceso receptor reciba información de otro proceso. Ejemplo `read` de UNIX y `ReadFile` de Windows.
- **Destruir**. Permite que el proceso solicite el cierre o destrucción del mecanismo. Ejemplo `close` de UNIX y `CloseHandle` de Windows.

Por otro lado, la comunicación puede ser síncrona o asíncrona. En la comunicación **síncrona** los dos procesos han de ejecutar los servicios de comunicación al mismo tiempo, es decir, el emisor ha de estar ejecutando el servicio de enviar y el receptor ha de estar ejecutando el servicio de recibir. Normalmente, para que esto ocurra uno de ellos ha de esperar a que el otro llegue a la ejecución del correspondiente servicio (véase la figura 2.17).

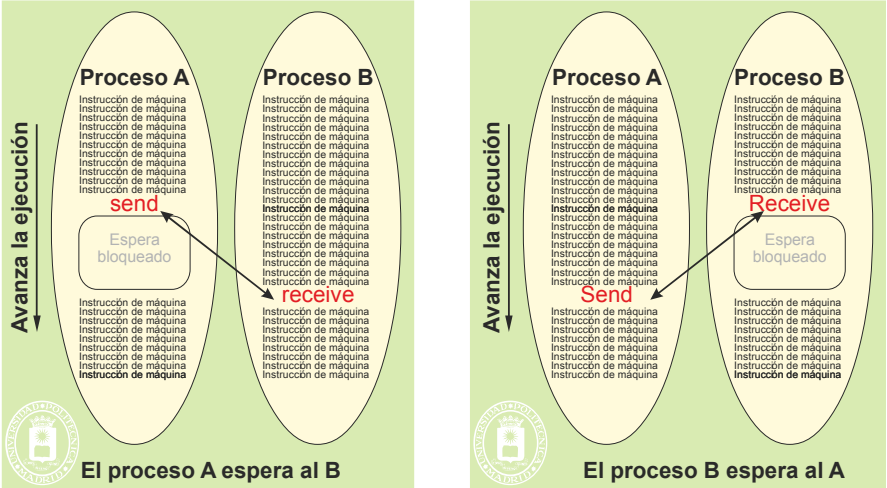


Figura 2.17 *Comunicación síncrona entre procesos.*

En la comunicación **asíncrona** el emisor no tiene que esperar a que el receptor solicite el servicio recibir, por el contrario, hace el envío y sigue con su ejecución. Esto obliga a que el sistema operativo establezca un almacenamiento intermedio para guardar la información enviada hasta que el receptor la solicite.

Los mecanismos de sincronización (como el semáforo y el *mutex*, que se estudian en detalle en el capítulo "6 Comunicación y sincronización de procesos") suelen incluir los siguientes servicios:

- **Crear**. Permite que el proceso solicite la creación del mecanismo. Ejemplo `sem_init` de UNIX y `CreateSemaphore` de Windows.
- **Esperar**. Permite que el proceso se bloquee en espera hasta que ocurra un determinado evento. Ejemplo `sem_wait` de UNIX y `WaitForSingleObject` de Windows.
- **Despertar**. Permite despertar a un proceso bloqueado. Ejemplo `sem_post` de UNIX y `ReleaseSemaphore` de Windows.
- **Destruir**. Permite que el proceso solicite el cierre o la destrucción del mecanismo. Ejemplo `sem_destroy` de UNIX y `CloseHandle` de Windows.

Aunque el sistema operativo es capaz de destruir los mecanismos de comunicación y sincronización cuando terminan los procesos que los utilizan, el programador profesional debe incluir siempre los pertinentes servicios de destrucción.

2.5.4. Gestión de la E/S

El gestor de E/S es el componente del sistema operativo que se encarga de los dispositivos periféricos, controlando su funcionamiento para alcanzar los siguientes objetivos:

- Facilitar el manejo de los dispositivos periféricos. Para ello debe ofrecer una interfaz sencilla, uniforme y fácil de utilizar, y debe gestionar los errores que se pueden producir en el acceso a los dispositivos periféricos.
- Garantizar la protección, impidiendo a los usuarios acceder sin control a los dispositivos periféricos.

Dentro de la gestión de E/S, el sistema operativo debe encargarse de gestionar los distintos dispositivos de E/S: relojes, terminales, dispositivos de almacenamiento secundario y terciario, etc.

Servicios

El sistema operativo ofrece a los usuarios una serie de servicios de E/S independientes de los dispositivos. Esta independencia implica que pueden emplearse los mismos servicios y operaciones de E/S para leer, por ejemplo, datos de un disquete, de un disco duro, de un CD-ROM o de un terminal. Los servicios de E/S están dirigidos básicamente a la lectura y escritura de datos. Según el tipo de periférico, estos servicios pueden estar orientados a caracteres, como ocurre con las impresoras o los terminales, o pueden estar orientados a bloques, como ocurre con las unidades de disco.

2.5.5. Gestión de ficheros y directorios

El servidor de ficheros es la parte de la máquina extendida, ofrecida por el sistema operativo, que cubre el manejo de los periféricos. Los objetivos fundamentales del servidor de ficheros son los siguientes:

- Facilitar el manejo de los dispositivos periféricos. Para ello ofrece una visión lógica simplificada de los mismos en forma de ficheros y de ficheros especiales.
- Proteger a los usuarios, poniendo limitaciones a los ficheros que es capaz de manipular cada usuario.

El servidor de ficheros ofrece al usuario (figura 2.18) una **visión lógica** compuesta por una serie de objetos (ficheros y directorios), identificables cada uno por un nombre lógico distinto. Los servicios son de dos tipos: los servicios dirigidos al manejo de datos (servicios sobre fichero), y los dirigidos al manejo de los nombres (servicios sobre directorio). El servidor de ficheros se suele encargar de ambos tipos de servicios, aunque, a veces, se incluyen servidores separados para datos y para nombres.

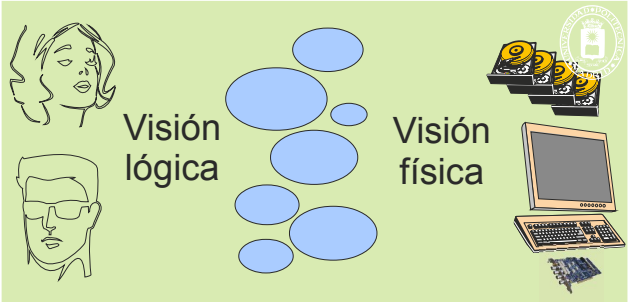


Figura 2.18 *Visión lógica y física del sistema de ficheros.*

La **visión física** incluye los detalles de cómo están proyectados estos objetos en los periféricos correspondientes (p. ej.: en los discos).

Ficheros

Un fichero es una unidad de almacenamiento lógico no volátil que agrupa bajo un mismo nombre un conjunto de informaciones normalmente relacionadas entre sí. Al fichero se le asocian los llamados atributos que utilizan tanto los usuarios como el propio servidor de ficheros. Los atributos más usuales son:

- Tipo de fichero (por ejemplo, fichero de datos, fichero ejecutable, directorio, etc.).
- Propietario del fichero (identificador del usuario que creó el fichero y del grupo de dicho usuario).
- Tamaño real en bytes del fichero. Al fichero se le asigna espacio en unidades de varios KiB llamadas **agrupaciones**. Es muy raro que la última agrupación esté completamente llena, quedando, por término medio, sin usarse media agrupación de cada fichero.
- Instantes (fecha y hora) importantes de la vida del fichero, como son los siguientes: a) instante en que se creó, b) instante de la última modificación y c) instante del último acceso.
- Derechos de acceso al fichero (sólo lectura, lectura-escritura, sólo escritura, ejecución,...).

Las operaciones sobre ficheros que ofrece el servidor de ficheros están referidas a la visión lógica de los mismos. La solución más común es que el fichero se visualice como un vector de bytes o caracteres, tal y como indica la figura 2.19. Algunos sistemas de ficheros ofrecen visiones lógicas más elaboradas, orientadas a registros, que pueden ser de longitud fija o variable. La ventaja de la sencilla visión de vector de caracteres es su flexibilidad, puesto que no presupone ninguna estructura específica interna en el fichero.

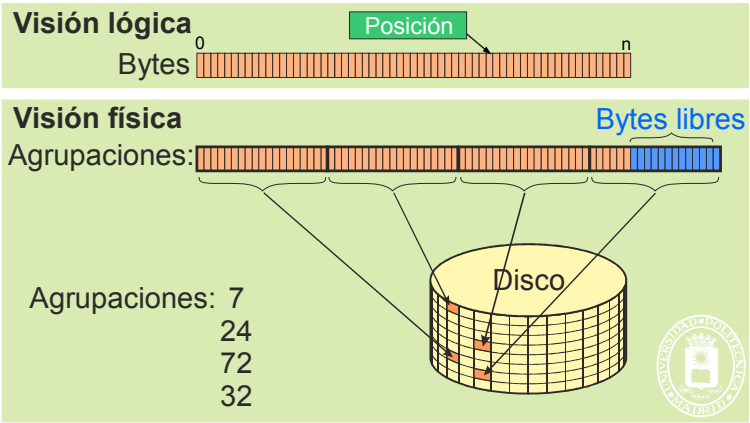


Figura 2.19 *Visión lógica y física de un fichero de disco.*

La visión lógica del fichero incluye normalmente un **puntero de posición**. Este puntero permite hacer operaciones de lectura y escritura consecutivas sin tener que indicar la posición dentro del fichero. Inicialmente el puntero indica la posición 0, pero después de hacer, por ejemplo, una operación de lectura de 7845 bytes señalará a la posición 7845. Otra lectura posterior se referirá a los bytes 7845, 7846, etc.

La visión física está formada por los elementos físicos del periférico que almacenan el fichero. En el caso más usual de tratarse de discos, la visión física consiste en la enumeración ordenada de los bloques de disco en los que reside el fichero. El servidor de ficheros debe mantener esta información en una estructura que se denominará, de forma genérica, **descripción física** del fichero. La descripción física reside en la FAT en MS-DOS, en el registro MFT en Windows y en el nodo-i en UNIX. Finalmente, es de destacar que estas estructuras de información han de residir en el propio periférico (p. ej.: disco), para que éste sea autocontenido y se pueda transportar de un sistema a otro. El servidor de ficheros es capaz de encontrar e interpretar estas estructuras de información, liberando a los programas de usuario de estos detalles.

El servidor de ficheros ofrece una visión lógica similar a la de un fichero, pero sin incluir puntero de posición, para periféricos tales como terminales, controladores de red, impresoras, etc. Se emplea el término de **fichero especial**, para diferenciarlos de los ficheros almacenados en disco o cinta magnética.

Servicios de ficheros

Un fichero es una entidad viva, que va evolucionando de acuerdo a los servicios que se solicitan sobre el mismo. Las fases de esta vida son las siguientes:

- Se **crea** el fichero.
  - Se **abre** el fichero para su uso (se genera un descriptor de fichero).
    - Se opera con el descriptor: **lee** y **escribe** (el fichero puede crecer).
  - Se **cierra** el fichero.
- Se **elimina** el fichero.

Los servicios que ofrece el servidor de ficheros son los siguientes:

- **Abrir un fichero**. Un fichero debe ser abierto antes de ser utilizado. Este servicio comprueba que el fichero existe, que el usuario tiene derechos de acceso y trae a memoria información del mismo para optimizar su acceso, creando en memoria el puntero de posición. Además, devuelve al usuario un identificador, descriptor o manejador de fichero de carácter temporal para su manipulación. Normalmente, todos los sistemas operativos tienen un límite máximo para el número de ficheros que puede tener abierto un usuario. Ejemplos: `open` y `creat` en UNIX y `CreateFile` en Windows.
- **Leer**. La operación de lectura permite traer datos del fichero a memoria. Para ello, se especifica el descriptor de fichero obtenido en la apertura, la posición de memoria para los datos y la cantidad de información a leer. Normalmente, se lee a partir de la posición que indica el puntero de posición del fichero. Ejemplos: `read` en UNIX y `ReadFile` en Windows.
- **Escribir**. Las operaciones de escritura permiten llevar datos situados en memoria al fichero. Para ello, y al igual que en las operaciones de lectura, se debe especificar el descriptor obtenido en la creación o apertura, la posición en memoria de los datos y la cantidad de información a escribir. Normalmente se escribe a partir de la posición que indica el puntero de posición del fichero. Si apunta dentro del fichero, se sobrescribirán los datos, no se añadirán. Si apunta al final del fichero se añaden los datos aumentando el tamaño del fichero. En este caso, el sistema operativo se encarga de hacer crecer el espacio físico del fichero añadiendo agrupaciones libres (si es que las hay). Ejemplos: `write` en UNIX y `WriteFile` en Windows.
- **Posicionar el puntero**. Sirve para especificar la posición del fichero en la que se realizará la siguiente lectura o escritura. Ejemplos: `lseek` en UNIX y `SetFilePointer` en Windows.
- **Cerrar un fichero**. Terminada la utilización del fichero se debe cerrar, con lo que se elimina el descriptor temporal obtenido en la apertura o creación y se liberan los recursos de memoria que ocupa el fichero. Ejemplos: `close` en UNIX y `CloseHandle` en Windows.
- **Crear un fichero**. Este servicio crea un fichero vacío. La creación de un fichero exige una interpretación del nombre, puesto que el servidor de ficheros ha de comprobar que el nombre es correcto y que el usuario tiene permisos para hacer la operación solicitada. La creación de un fichero lo deja abierto para escritura, devolviendo al usuario un identificador, descriptor o manejador de fichero de carácter temporal para su manipulación. Ejemplos: `creat` en UNIX y `CreateFile` en Windows.
- **Borrar un fichero**. El fichero se puede borrar, lo que supone que se borra su nombre del correspondiente directorio y que el sistema de ficheros ha de recuperar los bloques de datos y el espacio de descripción física que tenía asignado. Ejemplos: `unlink` en UNIX y `DeleteFile` en Windows.
- **Acceder a atributos**. Se pueden leer y modificar ciertos atributos del fichero tales como el dueño, la fecha de modificación, etc. Ejemplos: *chown* y *utime* en UNIX.

Se puede observar que el nombre del fichero se utiliza en los servicios de creación y de apertura. Ambos servicios dejan el fichero abierto y devuelven un descriptor de fichero. Los servicios para leer, escribir, posicionar el puntero y cerrar el fichero se basan en este descriptor y no en el nombre. Este descriptor es simplemente una referencia interna que mantiene el sistema operativo para trabajar eficientemente con el fichero abierto. Dado que un proceso puede abrir varios ficheros, el sistema operativo mantiene una **tabla de descriptores de fichero abiertos** por cada proceso. Además, todo proceso dispone, al menos, de tres elementos en dicha tabla, que reciben el nombre de **estándar**, y más concretamente de **entrada estándar**, **salida estándar** y **error estándar**. En un proceso interactivo la salida y error estándar están asignadas a la pantalla del terminal, mientras que la entrada estándar lo está al teclado. El proceso, por tanto, se comunica con el usuario a través de las entradas y salidas estándar. Por el contrario, un proceso que ejecuta en lotes tendrá los descriptores estándar asignados a ficheros.

Se denomina **redirección** a la acción de cambiar la asignación de un descriptor estándar.

Varios procesos pueden tener abierto y, por tanto, pueden utilizar simultáneamente el mismo fichero, por ejemplo, varios usuarios pueden estar leyendo la misma página de ayuda. Esto plantea un problema de coutilización del fichero, que se tratará en detalle en el capítulo “5 E/S y Sistema de ficheros”.

Servicios de directorios

Un directorio es un objeto que relaciona de forma unívoca un nombre con un fichero. El servicio de directorios sirve para identificar a los ficheros (objetos), por lo tanto, ha de garantizar que la relación [nombre → fichero] sea unívoca. Es decir, un mismo nombre no puede identificar a dos ficheros. Por el contrario, que un fichero tenga varios nombres no presenta ningún problema, son simples sinónimos.

El servicio de directorios también presenta una visión lógica y una visión física. La **visión lógica** consiste, habitualmente, en el bien conocido esquema jerárquico de nombres mostrado en la figura 2.20.

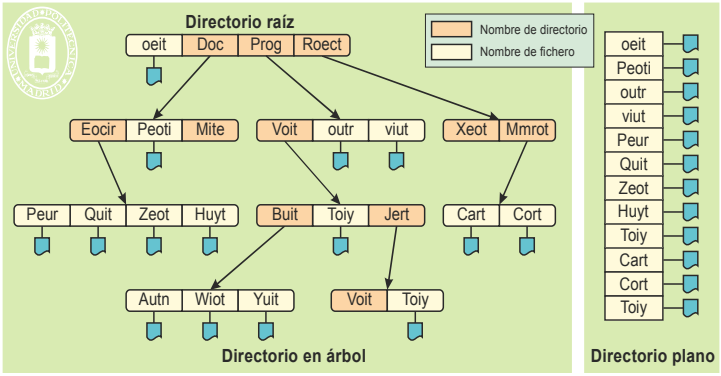


Figura 2.20 *Esquema jerárquico de directorios.*

Se denomina **directorio raíz** al primer directorio de la jerarquía, recibiendo los demás el nombre de subdirectorios o directorios. El directorio raíz se representa por el carácter «/» o «\», dependiendo del sistema operativo. En la figura 2.20, el directorio raíz incluye los siguientes nombres de subdirectorios: `Doc`, `Prog` y `Roect`.

Se diferencia el **nombre local**, que es el nombre asignando al fichero dentro del subdirectorio en el que está el fichero, del **nombre** o **camino absoluto**, que incluye todos los nombres de todos los subdirectorios que hay que recorrer desde el directorio raíz hasta el objeto considerado, concatenados por el símbolo «/» o «\». Un ejemplo de nombre local es «`Toiy`», mientras que su nombre absoluto es «`/Prog/Voit/Jert/Toiy`».

El sistema operativo mantiene un directorio de trabajo para cada proceso y un directorio *home* para cada usuario. El **directorio de trabajo** especifica un punto en el árbol que puede utilizar el proceso para definir ficheros sin más que especificar el nombre **relativo** desde ese punto. Por ejemplo, si el directorio de trabajo es «`/Prog/Voit/`», para nombrar el fichero Ar11 basta con poner «`Jert/Toiy`». El sistema operativo incluye servicios para cambiar el directorio de trabajo. El **directorio *home*** es el directorio asignado a un usuario para su uso. Es donde irá creando sus subdirectorios y ficheros.

La ventaja del esquema jerárquico es que permite una gestión distribuida de los nombres, al garantizar de forma sencilla que no existan nombres repetidos. En efecto, basta con que los nombres locales de cada subdirectorio sean distintos entre sí. Aunque los nombres locales de dos subdirectorios distintos coincidan, su nombre absoluto será distinto (p. ej.: «`/Prog/Voit/Toiy`» y «`/Prog/Voit/Jert/Toiy`»)

La **visión física** del sistema de directorios se basa en unas estructuras de información que permiten relacionar cada nombre lógico con la descripción física del correspondiente fichero. En esencia, se trata de una tabla NOMBRE-IDENTIFICADOR por cada subdirectorio, tabla que se almacena, a su vez, como un fichero. El NOMBRE no es más que el nombre local del fichero, mientras que el IDENTIFICADOR es una información que permite localizar la descripción física del fichero.

Servicios de directorios

Un objeto directorio es básicamente una tabla que relaciona nombres con ficheros. El servidor de ficheros incluye una serie de servicios que permiten manipular directorios. Estos son:

- **Crear un directorio**. Crea un objeto directorio y lo sitúa en el árbol de directorios. Ejemplos: `mkdir` en UNIX y `CreateDirectory` en Windows.
- **Borrar un directorio**. Elimina un objeto directorio, de forma que nunca más pueda ser accesible, y borra su entrada del árbol de directorios. Normalmente, sólo se puede borrar un directorio vacío, es decir, un directorio sin entradas. Ejemplos: `rmdir` en UNIX y `RemoveDirectory` en Windows.
- **Abrir un directorio**. Abre un directorio para leer los datos del mismo. Al igual que un fichero, un directorio debe ser abierto para poder acceder a su contenido. Esta operación devuelve al usuario un identificador, descriptor o manejador de directorio de carácter temporal que permite su manipulación. Ejemplos: `opendir` en UNIX y `FindFirstFile` en Windows.
- **Leer un directorio**. Extrae la siguiente entrada de un directorio abierto previamente. Devuelve una estructura de datos como la que define la entrada de directorios. Ejemplos: `readdir` en UNIX y `FindNextFile` en Windows.
- **Cambiar el directorio de trabajo**. Cambia el directorio de trabajo del proceso. Ejemplos `chdir` en UNIX y `SetCurrentDirectory` en Windows.
- **Cerrar un directorio**. Cierra un directorio, liberando el identificador devuelto en la operación de apertura, así como los recursos de memoria y del sistema operativo relativos al mismo. Ejemplos: `closedir` en UNIX y `FindClose` en Windows.

En el capítulo "5 E/S y Sistema de ficheros" se estudiará en detalle la gestión de ficheros y directorios, presentando los conceptos, los servicios y los principales aspectos de implementación.

2.6. SEGURIDAD Y PROTECCIÓN

La seguridad es uno de los elementos fundamentales en el diseño de los sistemas operativos de propósito general. La seguridad tiene por objetivo evitar la pérdida de bienes (datos o equipamiento) y controlar el uso de los mismos (privacidad de los datos y utilización de equipamiento). Es necesario proteger unos usuarios de otros, de forma que los programas de un usuario no interfieran con los programas de otro y que no puedan acceder a la información de otro.

El sistema operativo está dotado de unos mecanismos y políticas de **protección** con los que se trata de evitar que se haga un uso indebido de los recursos del computador. La protección reviste dos aspectos: garantizar la identidad de los usuarios y definir lo que puede hacer cada uno de ellos. El primer aspecto se trata bajo el término de autenticación, mientras que el segundo se basa en los privilegios.

Sin embargo, como el SO es un conjunto de programas, no puede supervisar las acciones de los programas cuando estos están ejecutando. Es necesario supervisar cada una de las instrucciones de máquina que ejecuta el programa, para garantizar que son instrucciones permitidas, y hay que supervisar cada uno de los accesos a memoria del programa, para comprobar que la dirección pertenece al proceso y que el tipo de acceso está permitido para esa dirección.

En un monoprocesador, cuando ejecuta un programa de usuario NO ejecuta el sistema operativo, por lo que éste no puede supervisar a los programas de usuario. Además, un programa no puede supervisar la ejecución de cada instrucción de máquina y cada acceso a memoria de otro programa.

La **supervisión** de la ejecución de los programas de usuario la **tiene que hacer un *hardware*** específico. Un computador diseñado para soportar sistemas operativos con protección ha de incluir unos mecanismos que detecten y avisen cuando los programas de los usuarios intentan realizar operaciones contrarias a la seguridad. En este sentido, tanto el procesador como la unidad de memoria tienen mecanismos de protección. Estos mecanismos se han estudiado en la sección "1.7 Protección".

Autenticación

El objetivo de la autenticación es determinar que un usuario (persona, servicio o proceso) es quien dice ser. El sistema operativo dispone de un módulo de autenticación que se encarga de validar la identidad de los usuarios. La contraseña (*password*) es, actualmente, el método de autenticación más utilizado.

Privilegios

Los privilegios especifican las operaciones que puede hacer un usuario sobre cada recurso. Para simplificar la información de privilegios es corriente organizar los usuarios en grupos y asignar los mismos privilegios a los componentes de cada grupo. La información de los privilegios se puede asociar a los recursos o a los usuarios.

- **Información por recurso**. En este caso se asocia una lista, denominada lista de control de acceso o ACL (*Access Control List*), a cada recurso. Esta lista especifica los grupos y usuarios que pueden acceder al recurso.
- **Información por usuario**. Se asocia a cada usuario, o grupo de usuarios, la lista de recursos que puede acceder, lista que se llama de capacidades (*capabilities*).

Dado que hay muchas formas de utilizar un recurso, la lista de control de acceso, o la de capacidades, ha de incluir el **modo** en que se puede utilizar el recurso. Ejemplos de modos de utilización son: leer, escribir, ejecutar, eliminar, test, control y administrar.

En su faceta de máquina extendida, el sistema operativo siempre comprueba, antes de realizar un servicio, que el proceso que lo solicita tiene los permisos adecuados para realizar la operación solicitada sobre el recurso solicitado.

Para llevar a cabo su función de protección, el sistema operativo ha de apoyarse en mecanismos *hardware* que supervisen la ejecución de los programas, entendiendo como tal a la función de vigilancia que hay que realizar sobre cada instrucción máquina que ejecuta el proceso de usuario. Esta vigilancia solamente la puede hacer el *hardware*, dado que mientras ejecuta el proceso de usuario el sistema operativo no está ejecutando, está «dormido», y consiste en comprobar que cada código de operación es permitido y que cada dirección de memoria y el tipo de acceso están también permitidos.

Privilegios UNIX

En los sistemas UNIX cada fichero, ya sea un fichero de usuario o un fichero directorio, incluye los siguientes 9 bits para establecer los privilegios de acceso:

Dueño	Grupo	Mundo
rwx	rwx	rwx

El primer grupo de 3 bits se aplica al dueño del fichero, el segundo al grupo del dueño y el tercer grupo al resto de usuarios.

Para los ficheros de usuario, el significado de estos bits es el siguiente:

- r: Especifica que el fichero se puede leer.
- w: Especifica que el fichero se puede escribir.
- x: Especifica que el fichero se puede ejecutar.

Para los ficheros de directorio, el significado de estos bits es el siguiente:

- r: Especifica que el directorio se puede leer, es decir, se puede ejecutar un "ls" para listar su contenido.
- w: Especifica que el directorio se puede escribir, es decir, se puede añadir, cambiar de nombre o borrar un fichero del directorio.
- x: Especifica que el directorio se puede atravesar para seguir analizando un nombre de fichero. (/home/datsi/asignaturas/ssoo/practicas/leeme.txt).

La secuencia que se utiliza para determinar si un proceso puede abrir un determinado fichero es la representada en la figura 2.21.

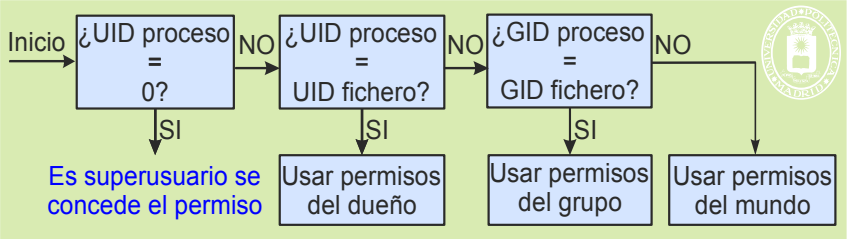


Figura 2.21 *Secuencia seguida para analizar si se puede abrir un fichero.*

2.7. INTERFAZ DE PROGRAMACIÓN

La interfaz de programación o API que ofrece un sistema operativo es una de sus características más importantes, ya que define la visión de máquina extendida que tiene el programador del mismo. En este libro se presentan dos de las interfaces más utilizadas en la actualidad: UNIX y Windows.

Actualmente, una gran cantidad de aplicaciones utilizan la codificación Unicode, por lo que los sistemas operativos también soportan este tipo de codificación. Sin embargo, esta codificación no aporta ningún concepto adicional desde el punto de vista de los sistemas operativos, por lo que, para simplificar, los ejemplos presentados en el presente libro no soportan Unicode.

2.7.1. Single UNIX Specification

La definición de UNIX ha ido evolucionando a lo largo de sus 30 años de existencia, siendo el estándar actual el "Single UNIX Specification UNIX 03" mantenido por un equipo (llamado el "Austin Group") formado por miembros de "IEEE Portable Applications Standards Committee", miembros de "The Open Group" y miembros de ISO/IEC Joint Technical Committee 1". Este estándar engloba los estándares anteriores XPG4 de X/Open, POSIX del IEEE y C de ISO.

Aclaración 2.3. UNIX 03 es una especificación estándar, no define una implementación. Los distintos sistemas operativos pueden ofrecer los servicios UNIX con diferentes implementaciones.

La parte básica del estándar "Single UNIX Specification UNIX 03" se divide en los siguientes documentos:

- "Base Definitions (XBD)". Incluye una serie de definiciones comunes a los demás documentos, por lo que es necesario conocerlo antes de abordar los otros documentos.
- "System Interfaces (XSH)". Describe una serie de servicios ofrecidos a los programas de aplicación.
- "*shell* and Utilities (XCU)". Describe el intérprete de mandatos (*shell*) y las utilidades disponibles a los programas de aplicación.
- "Rationale (XRAT)". Este documento es una ayuda para entender el resto del estándar.

UNIX es una interfaz ampliamente utilizada. Se encuentra disponible en todas las versiones de UNIX y Linux, incluso Windows 200X ofrece un subsistema que permite programar aplicaciones utilizando UNIX.

Algunas de las características de UNIX son las siguientes:

- Algunos tipos de datos utilizados por los servicios no se definen como parte del estándar, pero se definen como parte de la implementación. Estos tipos se encuentran definidos en el fichero de cabecera `<sys/types.h>` y acaban con el sufijo `_t`. Por ejemplo `uid_t` es el tipo que se emplea para almacenar un identificador de usuario.

- Los nombres de los servicios en UNIX son en general cortos y con todas sus letras en minúsculas. Ejemplos de servicios UNIX son `fork`, `read` y `close`.
- Los servicios, normalmente, devuelven cero si se ejecutaron con éxito o -1 en caso de error. Cuando un servicio devuelve -1, el código de error se almacena en una variable global (definida como externa) denominada `errno`. Este código de error es un valor entero. Los valores de la variable `errno` se encuentran definidos en el fichero de cabecera `<errno.h>`, donde se asocia cada número de error con su descripción.
- La mayoría de los recursos manipulables por los procesos se referencian mediante descriptores. Un descriptor es un número entero mayor o igual que cero.

2.7.2. Windows

Windows define los servicios ofrecidos por los sistemas operativos de la familia Windows. No se trata de un estándar genérico sino de los servicios establecidos por la empresa Microsoft.

El API de Windows es totalmente diferente al estándar UNIX. A continuación, se citan algunas de las principales características de Windows:

- Prácticamente todos los recursos gestionados por el sistema operativo se tratan como objetos, que se referencian por medio de manejadores. Estos manejadores son similares a los descriptores de UNIX. Aunque Windows sigue los principios de la programación orientada a objetos, no es orientado a objetos.
- Los nombres de los servicios en Windows son largos y descriptivos. Ejemplos de servicios en Windows son `CreateFile` y `WriteFile`:
- Windows tiene una serie de tipos de datos predefinidos, por ejemplo: `BOOL`, objeto de 32 bits que almacena un valor lógico, `DWORD`, entero sin signo de 32 bits, `TCHAR`, tipo carácter de dos bytes y `LPSTR`, puntero a una cadena de caracteres.
- Los tipos predefinidos en Windows evitan el uso del operador de indirección del lenguaje C (`*`). Así por ejemplo, `LPSTR` está definido como `*TCHAR`.
- Los nombres de las variables, al menos en los prototipos de los servicios, también siguen una serie de convenciones como son la codificación húngara (que incluye un prefijo que especifica el tipo de dato) y la codificación *CamelCase* (en la que pone en mayúscula la primera letra de cada palabra que forma el nombre). Por ejemplo, `lpszFileName` representa un puntero largo a una cadena de caracteres terminada por el carácter nulo.
- En Windows los servicios devuelven, en general, `true` si la llamada se ejecutó con éxito o `false` en caso contrario.

Aunque Windows incluye muchos servicios, que suelen tener numerosos argumentos (muchos de los cuales normalmente no se utilizan), este libro se va a centrar en los servicios más importantes del API. Además, Windows define servicios gráficos que no serán tratados en este libro.

2.8. INTERFAZ DE USUARIO DEL SISTEMA OPERATIVO

La interfaz de servicios del sistema operativo está dirigida a los programas. Para sacar partido de dicha interfaz hay que escribir programas. Sin embargo, la mayoría de los usuarios de un sistema informático no pretende ni desea realizar ninguna tarea de programación; son simplemente usuarios de aplicaciones. Por ello, el sistema operativo incluye una interfaz de usuario que ofrece un conjunto de operaciones típicas, que son las que necesitan normalmente llevar a cabo los usuarios. Así, para borrar un fichero, en vez de tener que realizar un programa, el usuario sólo tendrá que teclear un mandato (`rm` en UNIX o `del` en MS-DOS) o, en el caso de una interfaz gráfica, manipular un icono que representa al fichero.

La interfaz de usuario de los sistemas operativos, al igual que la de cualquier otro tipo de aplicación, ha sufrido una gran evolución. Esta evolución ha venido condicionada, en gran parte, por la enorme difusión del uso de los computadores, que ha tenido como consecuencia que un inmenso número de usuarios sin conocimientos informáticos trabajen cotidianamente con ellos. Se ha pasado de interfaces alfanuméricas, que requerían un conocimiento bastante profundo de los mandatos disponibles en el sistema, a interfaces gráficas, que ocultan al usuario la complejidad del sistema proporcionándole una visión intuitiva del mismo.

Ha existido también una evolución en la integración de la interfaz de usuario con el resto del sistema operativo. Se ha pasado de sistemas en los que el módulo que maneja la interfaz de usuario estaba dentro del núcleo del sistema operativo (la parte del mismo que ejecuta en modo privilegiado) a sistemas en los que esta función es realizada por un conjunto de programas externos al núcleo, que ejecutan en modo no privilegiado y usan los servicios del sistema operativo como cualquier otro programa. Esta estrategia de diseño proporciona una gran flexibilidad, permitiendo que cada usuario utilice un programa de interfaz que se ajuste a sus preferencias o que, incluso, cree uno propio. El sistema operativo, por tanto, se caracteriza principalmente por los servicios que proporciona al programador y no por su interfaz de usuario que, al fin y al cabo, puede ser diferente para los distintos usuarios.

A continuación se comentan las principales funciones de la interfaz de usuario de un sistema operativo.

2.8.1. Funciones de la interfaz de usuario

La principal misión de la interfaz, sea del tipo que sea, es permitir al usuario acceder y manipular los objetos del sistema. En esta sección se presentarán, de forma genérica, las operaciones que típicamente ofrece el sistema operativo a sus usuarios, con independencia de cómo lleven éstos a cabo el diálogo con el mismo.

A la hora de realizar esta enumeración surge una cuestión sobre la que no hay un acuerdo general: ¿cuáles de los programas que hay en un determinado sistema se consideran parte de la interfaz del sistema y cuáles no? ¿Un compilador es parte de la interfaz de usuario del sistema operativo? ¿Y un navegador web?

Una alternativa sería considerar que forman parte de la interfaz del sistema todos los programas que se incluyen durante la instalación del sistema operativo y dejar fuera de dicha categoría a los programas que se instalan posteriormente. Sin embargo, no hay un criterio único ya que diferentes fabricantes siguen distintas políticas. Por ejemplo, algunos sistemas incluyen uno o más compiladores de lenguajes de alto nivel, mientras que otros no lo hacen. Prueba de esta confusión ha sido el famoso contencioso legal de Microsoft sobre si el navegador web debería o no formar parte de su sistema operativo.

En la clasificación que se plantea a continuación se han seleccionado aquellas funciones sobre las que parece que hay un consenso general en cuanto a que forman parte de la interfaz del sistema. Se han distinguido las siguientes categorías.

- Manipulación de ficheros y directorios. La interfaz debe proporcionar operaciones para crear, borrar, renombrar y, en general, procesar ficheros y directorios.
- Ejecución de programas. El usuario debe poder ejecutar programas y controlar la ejecución de los mismos (por ejemplo, parar temporalmente su ejecución o terminarla incondicionalmente).
- Herramientas para el desarrollo de las aplicaciones. El usuario debe disponer de utilidades tales como ensambladores, enlazadores y depuradores, para construir sus propias aplicaciones. Observe que se han dejado fuera de esta categoría a los compiladores por los motivos antes expuestos.
- Comunicación con otros sistemas. Existirán herramientas para acceder a recursos localizados en otros sistemas accesibles a través de una red de conexión. En esta categoría se consideran herramientas básicas, tales como `ftp` y `telnet` (aclaración 2.4), dejando fuera aplicaciones de más alto nivel como un navegador web.
- Información de estado del sistema. El usuario dispondrá de utilidades para obtener informaciones tales como la fecha, la hora, el número de usuarios que están trabajando en el sistema o la cantidad de memoria disponible.
- Configuración de la propia interfaz y del entorno. Cada usuario tiene que poder configurar el modo de operación de la interfaz de acuerdo a sus preferencias. Un ejemplo sería la configuración de los aspectos relacionados con las características específicas del entorno geográfico del usuario (el idioma, el formato de fechas, de números y de dinero, etc.). La flexibilidad de configuración de la interfaz será una de las medidas que exprese su calidad.
- Intercambio de datos entre aplicaciones. El usuario va a disponer de mecanismos que le permitan especificar que, por ejemplo, una aplicación utilice los datos que genera otra.
- Control de acceso. En sistemas multiusuario, la interfaz debe encargarse de controlar el acceso de los usuarios al sistema para mantener la seguridad del mismo. Normalmente, el mecanismo de control estará basado en que cada usuario autorizado tenga una contraseña que deba introducir para acceder al sistema.
- Sistema de ayuda interactivo. La interfaz debe incluir un completo entorno de ayuda que ponga a disposición del usuario toda la documentación del sistema.
- Copia de datos entre aplicaciones. Al usuario se le proporciona un mecanismo de tipo copiar y pegar (*copy-and-paste*) para poder pasar información de una aplicación a otra.

Aclaración 2.4. La aplicación `ftp` (*file transfer protocol*) permite transferir ficheros entre computadores conectadas por una red de conexión. La aplicación `telnet` permite a los usuarios acceder a computadores remotas, de tal manera que el computador en la que se ejecuta la aplicación `telnet` se convierte en un terminal del computador remoto.

Para concluir esta sección, es importante resaltar que en un sistema, además de las interfaces disponibles para los usuarios normales, pueden existir otras específicas destinadas a los administradores del sistema. Más aún, el propio programa (residente normalmente en ROM) que se ocupa de la carga del sistema operativo proporciona generalmente una interfaz de usuario muy simplificada y rígida que permite al administrador realizar operaciones tales como pruebas y diagnósticos del *hardware* o la modificación de los parámetros almacenados en la memoria no volátil de la máquina que controlan características de bajo nivel del sistema.

2.8.2. Interfaces alfanuméricas

La característica principal de este tipo de interfaces es su modo de trabajo basado en líneas de texto. El usuario, para dar instrucciones al sistema, escribe en su terminal un mandato terminado con un carácter de final de línea. Cada mandato está normalmente estructurado como un nombre de mandato (por ejemplo, borrar) y unos argumentos (por ejemplo, el nombre del fichero que se quiere borrar). Observe que en algunos sistemas se permite que se introduzcan varios mandatos en una línea. El intérprete de mandatos lee la línea escrita por el usuario y lleva a cabo las acciones

especificadas por la misma. Una vez realizadas, el intérprete escribe una indicación (*prompt*) en el terminal para notificar al usuario que está listo para recibir otro mandato. Este ciclo repetitivo define el modo de operación de este tipo de interfaces.

El carácter «|» se utiliza para enlazar mandatos mediante tuberías (mandato1 | mandato2). El *shell* crea un proceso por mandato y los une mediante una tubería, de forma que la salida estándar del primero queda conectada a la entrada estándar del segundo: lo que escribe el primero por su salida es lo que lee el segundo por su entrada. Por otro lado, los caracteres «<» y «>» se utilizan para redirigir la entrada y salida estándar, respectivamente, es decir, para cambiar el fichero o fichero especial al que están asignadas.

Esta forma de operar, basada en líneas de texto, viene condicionada en parte por el tipo de dispositivo que se usaba como terminal en los primeros sistemas de tiempo compartido. Se trataba de teletipos que imprimían la salida en papel y que, por tanto, tenían intrínsecamente un funcionamiento basado en líneas. La disponibilidad posterior de terminales más sofisticados que, aunque seguían siendo de carácter alfanumérico, usaban una pantalla para mostrar la información y ofrecían, por tanto, la posibilidad de trabajar con toda la pantalla, no cambió, sin embargo, la forma de trabajo de la interfaz que continuó siendo en modo línea. Como reflejo de esta herencia, observe que en el mundo UNIX se usa el término `tty` (abreviatura de *teletype*) para referirse a un terminal, aunque no tenga nada que ver con los primitivos teletipos. Sin embargo, muchas aplicaciones sí que se aprovecharon del modo pantalla. Como ejemplo, se puede observar la evolución de los editores en UNIX: se pasó de editores en modo línea como el `ed` a editores orientados a pantalla como el `vi` y el `emacs`.

La tabla 2.1 contiene algunos de los mandatos que se encuentran en los *shell* de UNIX y el cmd-line de Windows. Se han agrupado en la misma línea mandatos que son similares, pero sin olvidar que, a veces, las diferencias son sustanciales. El lector que desee utilizar estos mandatos deberá consultar los correspondientes manuales o ayudas *online*, para su descripción detallada.

Tabla 2.1 *Algunos mandatos que se encuentran en los shell de UNIX y en el cmd-line de Windows.*

UNIX	cmd-line	Descripción
echo	echo	Muestra un mensaje por pantalla
ls	dir	Lista los ficheros de un directorio
mkdir	mkdir	Crea un directorio
rmdir	rmdir	Borra directorios
rm	del y erase	Borra ficheros
mv	move	Traslada ficheros de un directorio a otro
cp	copy y xcopy	Copia ficheros
cat	type	Visualiza el contenido de un fichero
chmod	attrib	Cambia los atributos de un fichero
mv	rename	Permite cambiar el nombre de los ficheros
lpr	print	Imprime un fichero
date	date	Muestra y permite cambiar la fecha
time	time	Muestra y permite cambiar la hora
grep	find y findstr	Busca una cadena de caracteres en un fichero
find		Busca ficheros que cumplen determinadas características
sort	sort	Lee la entrada, ordena los datos y los escribe en pantalla, fichero u otro dispositivo
mkfs	format	Da formato a un disco
diff	comp	Compara los contenidos de dos ficheros
cd	cd, pushd y popd	Cambia el directorio de trabajo
	path	Muestra y cambia el camino de búsqueda para ficheros ejecutables
env	set	Muestra variables de entorno
sleep	sleep	Espera que transcurran los segundos indicados
id	whoami	Muestra la identidad del usuario que lo ejecuta
pwd		Visualiza el directorio actual de trabajo
tty		Indica si la entrada estándar es un terminal
#	rem	Permite incluir comentarios en un fichero de mandatos
<mifich>	call <mifich>	Permite llamar a un fichero de mandatos desde otro
read	pause	Suspende la ejecución del fichero de mandatos hasta que se pulse una tecla
read	Set /P	Lee una línea del teclado
exit	exit	Finaliza la ejecución del fichero de mandatos

2.8.3. Interfaces gráficas

El auge de las interfaces gráficas de usuario o GUI se debe principalmente a la necesidad de proporcionar a los usuarios no especializados una visión sencilla e intuitiva del sistema que oculte toda su complejidad. Esta necesidad ha surgido por la enorme difusión de los computadores en todos los ámbitos de la vida cotidiana. Sin embargo, el

desarrollo de este tipo de interfaces más amigables ha requerido un avance considerable en la potencia y capacidad gráfica de los computadores, dada la gran cantidad de recursos que consumen durante su operación.

Estas interfaces están basadas en **ventanas** que permiten al usuario trabajar simultáneamente en distintas actividades. Asimismo, se utilizan **iconos** para representar los recursos del sistema y **menús** para poder realizar operaciones sobre los mismos. El usuario utiliza un ratón (o dispositivo equivalente) para interaccionar con estos elementos. Así, por ejemplo, para arrancar una aplicación el usuario tiene que apuntar a un icono con el ratón y apretar un botón del mismo, o para copiar un fichero señalar al icono que lo representa y, manteniendo el botón del ratón apretado, moverlo hasta ponerlo encima de un icono que representa el directorio destino. Generalmente, para agilizar el trabajo de los usuarios más avanzados, estas interfaces proporcionan la posibilidad de realizar estas mismas operaciones utilizando ciertas combinaciones de teclas. Dado el carácter intuitivo de estas interfaces, y el amplio conocimiento que posee de ellas todo el mundo, no parece necesario entrar en más detalles sobre su forma de trabajo. Además de la funcionalidad comentada, otros aspectos que conviene resaltar son los siguientes:

- Uso generalizado del mecanismo del copiar y pegar (*copy-and-paste*).
- Sistema de ayuda interactivo. Los sistemas de ayuda suelen ser muy sofisticados, basándose muchos de ellos en hipertexto.
- Oferta de servicios a las aplicaciones (**API gráfico**). Además de encargarse de atender al usuario, estos entornos gráficos proporcionan a las aplicaciones una biblioteca de primitivas gráficas que permiten que los programas creen y manipulen objetos gráficos.
- Posibilidad de acceso a la interfaz alfanumérica. Muchos usuarios se sienten encorsetados dentro de la interfaz gráfica y prefieren usar una interfaz alfanumérica para realizar ciertas operaciones. La posibilidad de acceso a dicha interfaz desde el entorno gráfico ofrece al usuario un sistema con lo mejor de los dos mundos.

2.8.4. Ficheros de mandatos o *shell-scripts*

Un fichero de mandatos o *shell-script* permite tener almacenados una secuencia de mandatos de *shell* que se pueden ejecutar simplemente invocando el nombre del fichero, como si de un mandato más se tratase. Estos ficheros son muy empleados por los administradores de sistemas, puesto que permiten automatizar sus labores de administración.

Esta sección presenta, de forma muy resumida, las características más importantes de los ficheros de mandatos empleados en entornos UNIX y Windows. El lector interesado deberá completar la sección con los manuales de los fabricantes.

UNIX

Para UNIX existen varios lenguajes de mandatos, entre los que destaca el de Bourne, al que nos referiremos en esta sección. Algunos aspectos importantes son los siguientes:

- La primera línea del fichero de mandatos debe especificar el intérprete utilizado. Para establecer que se trata de un fichero Bourne debe escribirse: `#!/bin/sh`.
- Hay que definir el fichero de mandatos como ejecutable. Ejemplo: `chmod +x mifichero`.

Redirección y concatenación

La salida y entrada estándar de los procesos que ejecutan los mandatos se puede redireccionar, por ejemplo, para leer o escribir de ficheros.

- La salida estándar se redirecciona mediante >. Ejemplo: `ls > archiv`
- La entrada estándar se redirecciona mediante <. Ejemplo: `miprog < archiv`
- La salida de error estándar se redirecciona mediante 2>. Ejemplo: `miprog 2> archiv`
- La salida estándar se puede añadir a un fichero con >>. Ejemplo: `miprog >> archiv`
- La tubería `|` permite unir dos mandatos de forma que la salida estándar del primero se conecta a la entrada estándar del segundo. Ejemplo: `ls | grep viejo`
- El carácter `&` permite ejecutar un mandato en segundo plano. Ejemplo: `prog2 &`
- El carácter `;` permite escribir dos mandatos en una misma línea. Ejemplo: `prog1 ; prog2`
- El conjunto `&&` permite unir dos mandatos `prog1 && prog2`, de forma que si `prog1` termina bien sí se ejecuta `prog2`. En caso contrario `prog2` no se ejecuta.
- El conjunto `||` permite unir dos mandatos `prog1 || prog2`, de forma que si `prog1` termina bien no se ejecuta `prog2`. En caso contrario `prog2` sí se ejecuta.

Variables

- Existen variables de entorno como: `USER, TERM, HOME, PATH`, etc.
- Las definición de nuevas variables y la asignación de valores a las variables existentes se hace mediante: `nombre=valor`. Ejemplo: `mivariable=16`
- Las variables se referencian mediante: `$nombre` o `${nombre}`. Ejemplo: `echo $msg1 $DATE`

Argumentos

Cuando se invoca al fichero de mandatos se pueden añadir argumentos, separados por espacios. Estos argumentos se referencian mediante: `$0`, `$1`, `$2`, `$3`, `$4`, `$5`, `$6`, `$7`, `$8` y `$9`. El `$0` es especial, puesto que es el propio nombre del fichero de mandatos.

`$*` y `$@` representan una cadena de caracteres con todos los argumentos existentes. Sin embargo, "`$*`" equivale a "`$1 $2 $3` …", mientras que "`$@`" equivale a "`$1`" "`$2`" "`$3`"…

Comillas

Las comillas son necesarias para poder construir cadenas de caracteres que contengan espacios y otros caracteres especiales.

- El carácter \ elimina el significado especial del carácter que le sigue.
- Comillas simples `'cadena'`. Permite escribir cadenas que contengan caracteres especiales salvo la comilla simple (para incluirla es necesario poner \').
- Comillas dobles "`cadena`". Preserva la mayoría de los caracteres especiales, pero las variables y las comillas invertidas se evalúan y se sustituyen en la cadena.
- Comillas invertidas `` `mandato` ``. Se ejecuta el mandato y se sustituye por lo que escriba por su salida estándar.

Control de flujo

La condición, en todos los casos, debe ser un mandato: Si devuelve cero como estado de salida la condición es cierta, en caso contrario es falsa.

- IF. La sintaxis es la siguiente:

```
if condición ; then
   mandatos
[elif condición; then
   mandatos]...
[else
   mandatos]
fi
```

- WHILE. El bucle WHILE puede contener mandatos `break`, que terminan el bucle, y mandatos `continue`, que saltan al principio del bucle, ignorando el resto. La sintaxis es la siguiente:

```
while condición; do
   mandatos
done
```

- FOR. El bucle WHILE puede contener mandatos `break` y `continue`. La sintaxis es la siguiente:

```
for var in list; do
   mandatos
done
```

- CASE. La construcción `case` es similar a la sentencia `switch` del lenguaje C. Sin embargo, en vez de comprobar valores numéricos comprueba patrones.

Funciones

Se pueden definir funciones que se invocan como cualquier mandato. La definición es:

```
nombre () {
   mandatos
}
```

El programa 2.2 presenta un ejemplo de fichero de mandatos Bourne.

Programa 2.2 Ejemplo de *script* que traslada los ficheros especificados en los argumentos a un directorio llamado `basket` y que está ubicado en el `home`. ;Se pone a

```
#! /bin/sh
PATH=/usr/bin: /bin                # Es conveniente dar valor a PATH por seguridad
IFS=                               # Es conveniente dar valor a ISF por seguridad
BASKET=$HOME/.basket               # El directorio basket se crea en el HOME
ME=`basename $0`
# Se crea una función que permite preguntar por Y o y.
function ask  yes() {
   if tty –s                       # ¿Interactivo?
   then
      echo -n "$ME: $@ "
      read ANS
```

```
        case "$ANS" in
        y|Y) return 0;;
        esac
    fi
    return 1
}
# Cuerpo del script
if [ $# -eq 0 ]                    # Comprueba que hay argumentos
then
    echo Uso: $ME files >&2
    exit 64
fi
if [ ! -d $BASKET ]                # Comprueba si existe el directorio basket
then
    echo "$ME: $BASKET no existe." >&2
    ask  yes "create $BASKET?" || exit 69      # Confirma y aborta
    # Crea el directorio basket en el HOME
    mkdir $BASKET || {
        echo $ME: abortado
        exit 72
    } >&2
fi
# Se realiza el traslado de ficheros especificados por los argumentos
echo mv -biuv "$@" $BASKET
# Se sale con el estado del último mandato (mv).
```

Windows

El lenguaje clásico de mandatos de Windows se llama cmd-line. Los nombres de los ficheros de mandatos deben tener la extensión «.bat». Existen otros lenguajes de mandatos como son el VBScript, el JScript y el nuevo Powershell bastante más sofisticados que el cmd-line. Seguidamente se desarrollan brevemente las principales características del cmd-line.

Redirección y concatenación

Las salidas y entrada estándar de los mandatos se puede redireccionar.

- La salida estándar se redirecciona mediante >. Ejemplo: `dir > archiv`
- La entrada estándar se redirecciona mediante <. Ejemplo: `miprog < archiv`
- La salida de error estándar se redirecciona mediante 2>. Ejemplo: `miprog 2> archiv`
- La salida estándar se puede añadir a un fichero con >>. Ejemplo: `miprog >> archiv`
- La tubería | permite unir dos mandatos de forma que la salida estándar del primero se conecta a la entrada estándar del segundo. Ejemplo: `find "Pepe" listacorreo.txt | sort`
- El carácter `&` permite escribir dos mandatos en una misma línea. Ejemplo: `prog1 & prog2`
- El conjunto `&&` permite unir dos mandatos `prog1 && prog2` de forma que si `prog1` devuelve cero como estado de salida se ejecuta `prog2`. En caso contrario `prog2` no se ejecuta.
- El conjunto `||` permite unir dos mandatos `prog1 || prog2` de forma que si `prog1` devuelve cero como estado de salida no se ejecuta `prog2`. En caso contrario `prog2` sí se ejecuta.

Variables

- Existen unas 30 variables de entorno como: `DATE`, `ERRORLEVEL`, `HOMEPATH`, etc.
- Las definición de nuevas variables y la asignación de valores a las variables existentes se hace mediante: `set nombre=valor`. Ejemplo: `set mivariable=16`
- Las variables se referencian mediante: `%nombre%`. Ejemplo: `echo %msg1% %DATE%`
- Para asignar un valor a un variable mediante una expresión aritmética o lógica hay que usar la opción /a. Ejemplo para incrementar una variable: `set /a mivar=%mivar% + 1`

Argumentos

Cuando se invoca al fichero de mandatos se pueden añadir argumentos, separados por el carácter «,» o el carácter «;». Estos argumentos se referencian mediante: `%0`, `%1`, `%2`, `%3`, `%4`, `%5`, `%6`, `%7`, `%8` y `%9`. El `%0` es especial, puesto que es el propio nombre del fichero de mandatos.

Control de flujo

El cmd-line solamente dispone de GOTO, CALL, IF y FOR para establecer mandatos condicionales. Las sintaxis son las siguientes:

- GOTO nombreetiqueta
- CALL nombrefichero [argumentos]
- IF [NOT] ERRORLEVEL número mandato [argumentos]

- IF [NOT] string1==string2 mandato [argumentos]. Ejemplo: if "%1"=="3" echo Tercera vez
- IF [NOT] EXIST filename mandato [argumentos]
- FOR %%variable IN (conjunto) DO mandato [argumentos]. conjunto es uno o un conjunto de ficheros. Ejemplo: FOR %%f IN (*.txt) DO TYPE %%f >> bigtxt
- FOR /L %%variable IN (start,step,end) DO mandato [argumentos]. Ejemplo: FOR /L %%i IN (0 2 100) DO echo %%i

El programa 2.3 presenta un ejemplo de fichero de mandatos cmd-line.

Programa 2.3 Ejemplo que traslada los ficheros definidos mediante un argumento a un directorio basket creado en el HOMEPATH. Para su ejecución almacenarlo en un fichero `traslado.bat` y llamarlo mediante `traslado` *`ficheros`*

```
@echo off
set BASKET=%HOMEPATH%\basket
set ME=%0
set FINAL=El script termina sin ejecutar el traslado
   rem Cuerpo del script
   rem Si no existe argumento, el script termina
if "%1"==""" echo Uso: %ME% ficheros... & goto fin
   rem pushd cambia de directorio y guarda el antiguo
pushd "%BASKET%"
   rem Si pushd no da error, ERRORLEVEL <> 0, el directorio existe
   rem Hay que restituir el directorio de trabajo con popd
if "%ERRORLEVEL%"=="0" popd & goto traslado
   rem Se trata el caso de que no exista directorio
echo %ME%: El directorio basket no existe
SET /P SINO=%ME%: Desea crearlo? S^|N =
   rem Si no se pulsa S el script termina
if not "%SINO%"=="S" goto fin
   rem Se crea el directorio
mkdir "%BASKET%"
: traslado
   rem Esta parte es la que hace el traslado de los ficheros
move /Y "%1" "%BASKET%"
set FINAL=El script termina correctamente
: fin
echo %ME%: "%FINAL%"
```

2.9. DISEÑO DE LOS SISTEMAS OPERATIVOS

2.9.1. Estructura del sistema operativo

Un sistema operativo es un programa extenso y complejo que está compuesto, como se ha visto en las secciones anteriores, por una serie de componentes con funciones bien definidas. Cada sistema operativo estructura estos componentes de distinta forma. Los sistemas operativos se pueden clasificar, de acuerdo a su estructura, en sistemas operativos monolíticos y sistemas operativos estructurados.

Analizaremos en esta sección estas dos alternativas, así como la estructura de las máquinas virtuales y de los sistemas distribuidos.

Sistemas operativos monolíticos

Un sistema operativo monolítico no tiene una estructura clara y bien definida. Todos sus componentes se encuentran integrados en un único programa (el sistema operativo) que ejecuta en un único espacio de direcciones. Además, todas las funciones que ofrece se ejecutan en modo privilegiado.

Ejemplos claros de este tipo de sistemas son el OS-360, el MS-DOS y el UNIX. Estos dos últimos comenzaron siendo pequeños sistemas operativos, que fueron haciéndose cada vez más grandes, debido a la gran popularidad que adquirieron.

El problema que plantea este tipo de sistema radica en lo complicado que es modificarlos para añadir nuevas funcionalidades y servicios. En efecto, añadir una nueva característica al sistema operativo implica la modificación de un gran programa, compuesto por miles o millones de líneas de código fuente y de infinidad de funciones, cada una de las cuales puede invocar a otras cuando así los requiera. Además, en este tipo de sistemas no se sigue el prin-

cipio de ocultación de la información. Para solucionar estos problemas es necesario dotar de cierta estructura al sistema operativo.

Sistemas operativos estructurados

Cuando se quiere dotar de estructura a un sistema operativo, normalmente se recurren a dos tipos de soluciones: sistemas por capas y sistemas cliente-servidor.

Sistemas por capas

En un sistema por capas, el sistema operativo se organiza como una jerarquía de capas, donde cada capa ofrece una interfaz clara y bien definida a la capa superior y solamente utiliza los servicios que le ofrece la capa inferior.

La principal ventaja que ofrece este tipo de estructura es la modularidad y la ocultación de la información. Una capa no necesita conocer cómo se ha implementado la capa sobre la que se construye, únicamente necesita conocer la interfaz que ésta ofrece. Esto facilita enormemente la depuración y verificación del sistema, puesto que las capas se pueden ir construyendo y depurando por separado.

Este enfoque lo utilizó por primera vez el sistema operativo THE, un sistema operativo sencillo formado por seis capas, como se muestra en la figura 2.22. Otro ejemplo de sistema operativo diseñado por capas fue el OS/2, descendiente de MS-DOS.

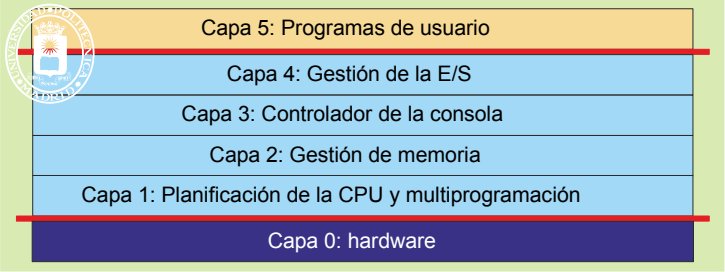


Figura 2.22 *Estructura por capas del sistema operativo THE.*

Modelo cliente-servidor

En este tipo de modelo, el enfoque consiste en implementar la mayor parte de los servicios y funciones del sistema operativo para que ejecute como procesos en modo usuario, dejando sólo una pequeña parte del sistema operativo ejecutando en modo privilegiado. Esta parte se denomina micronúcleo y los procesos que ejecutan el resto de funciones se denominan servidores. Cuando se lleva al extremo esta idea se habla de nanonúcleo. La figura 2.23 presenta la estructura de un sistema operativo con estructura cliente-servidor. Como puede apreciarse en la figura, el sistema operativo está formado por diversos módulos, cada uno de los cuales puede desarrollarse por separado.

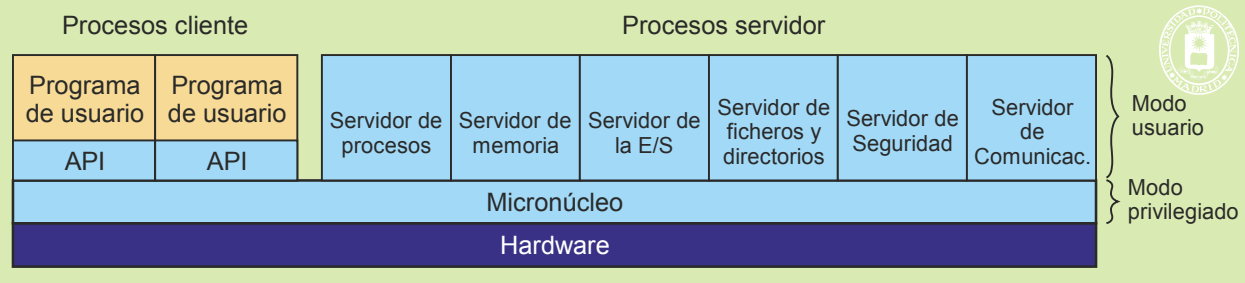


Figura 2.23 *Estructura cliente-servidor en un sistema operativo.*

No hay una definición clara de las funciones que debe llevar a cabo un micronúcleo. La mayoría incluyen la gestión de interrupciones, gestión básica de procesos y de memoria, y servicios básicos de comunicación entre procesos. Para solicitar un servicio en este tipo de sistema, como por ejemplo crear un proceso, el proceso de usuario (proceso denominado cliente) solicita el servicio al servidor del sistema operativo correspondiente, en este caso al servidor de procesos. A su vez, el proceso servidor puede requerir los servicios de otros servidores, como es el caso del servidor de memoria. En este caso, el servidor de procesos se convierte en cliente del servidor de memoria.

Una ventaja de este modelo es la gran flexibilidad que presenta. Cada proceso servidor sólo se ocupa de una funcionalidad concreta, lo que hace que cada parte pueda ser pequeña y manejable. Esto a su vez facilita el desarrollo y depuración de cada uno de los procesos servidores. Por otro lado, al ejecutar los servicios en espacios separados aumenta la fiabilidad del conjunto, puesto que un fallo solamente afecta a un módulo.

En cuanto a las desventajas, citar que estos sistemas presentan una mayor sobrecarga en el tratamiento de los servicios que los sistemas monolíticos. Esto se debe a que los distintos componentes de un sistema operativo de este tipo ejecutan en espacios de direcciones distintos, lo que hace que su activación requiera mayor tiempo.

Minix, Mach, Amoeba y Mac OS X, son ejemplos de sistemas operativos que siguen este modelo. Windows NT también sigue esta filosofía de diseño, aunque muchos de los servidores (el gestor de procesos, gestor de E/S, gestor de memoria, etc.) se ejecutan en modo privilegiado por razones de eficiencia (véase la figura 2.24).

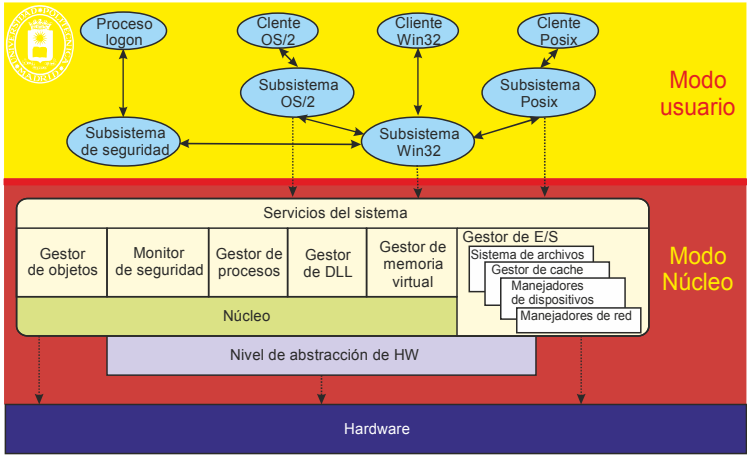


Figura 2.24 *Estructura del sistema operativo Windows NT.*

Máquina virtual

El concepto de máquina virtual se basa en un monitor capaz de suministrar **m** versiones del *hardware*, es decir **m** máquinas virtuales. Cada copia es una réplica exacta del *hardware*, incluso con modo privilegiado y modo usuario, por lo que sobre cada una de ellas se puede instalar un sistema operativo convencional. Como muestra la figura 2.25, una petición de servicio es atendida por la copia de sistema operativo sobre la que ejecuta. Cuando dicho sistema operativo desea acceder al *hardware*, por ejemplo, para leer de un periférico, se comunica con su máquina virtual, como si de una máquina real se tratase. Sin embargo, con quien se está comunicando es con el monitor de máquina virtual que es el único que accede a la máquina real.

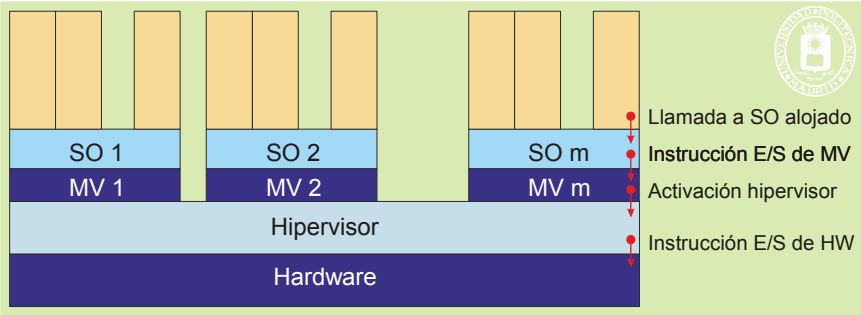


Figura 2.25 *Esquema de máquina virtual soportando varios sistemas operativos.*

El ejemplo más relevantes de máquina virtual es el VM/370, que genera **m** copias de la arquitectura IBM370.

Otra forma distinta de construir una máquina virtual es la mostrada en la figura 2.26. En este caso, el monitor de máquina virtual no ejecuta directamente sobre el *hardware*, por el contrario, lo hace sobre un sistema operativo.

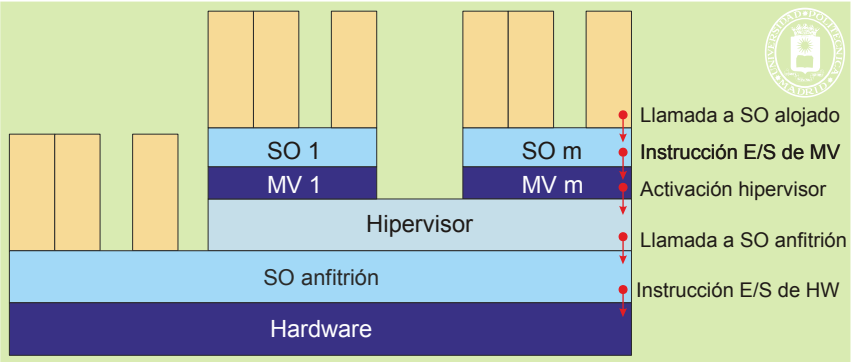


Figura 2.26 *Esquema de máquina virtual sobre sistema operativo.*

La máquina virtual generada puede ser una copia exacta del *hardware* real, como es el caso del VMware, que genera una copia virtual de la arquitectura Pentium, o puede ser una máquina distinta como es el caso de la máquina virtual de Java (JVM *Java Virtual Machine*) o la CLI (*Common Language Infrastructure*) incluida en Microsoft.-NET.

Las ventajas e inconvenientes de la máquina virtual son los siguientes:

- Añade un nivel de multiprogramación, puesto que cada máquina virtual ejecuta sus propios procesos. IBM introdujo de esta forma la multiprogramación en sus sistemas 370.
- Añade un nivel adicional de aislamiento, de forma que si se compromete la seguridad en un sistema operativo no se compromete en los demás. Esta funcionalidad puede ser muy importante cuando se están haciendo pruebas.

- Si lo que se genera es una máquina estándar, como es el caso de JVM o CLI, se consigue tener una plataforma de ejecución independiente del *hardware*, por lo que no hay que recompilar los programas para pasar de un computador a otro.
- Tiene el inconveniente de añadir una sobrecarga computacional, lo que puede hacer más lenta la ejecución.

En algunos diseños se incluye una capa, llamada ***exokernel***, que ejecuta en modo privilegiado y que se encarga de asignar recursos a las máquinas virtuales y de garantizar que ninguna máquina utilice recursos de otra.

Por otro lado, dada la importancia que está tomando el tema de las máquinas virtuales, algunos fabricantes de procesadores están incluyendo mecanismos *hardware*, tales como modos especiales de ejecución, para facilitar la construcción de las mismas.

Sistema operativo distribuido

Un sistema operativo distribuido es un sistema operativo diseñado para gestionar un multicomputador, como muestra la figura 2.27. El usuario percibe un único sistema operativo centralizado, haciendo, por tanto, más fácil el uso de la máquina. Un sistema operativo de este tipo sigue teniendo las mismas características que un sistema operativo convencional pero aplicadas a un sistema distribuido. Estos sistemas no han tenido éxito comercial y se han quedado en la fase experimental.

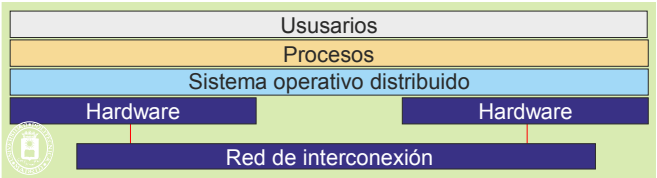


Figura 2.27 *Estructura de un sistema operativo distribuido.*

Middleware

Un *middleware* es una capa de *software* que se ejecuta sobre un sistema operativo ya existente y que se encarga de gestionar un sistema distribuido o un multicomputador, como muestra la figura 2.28. En este sentido, presenta una funcionalidad similar a la de un sistema operativo distribuido. La diferencia es que ejecuta sobre sistemas operativos ya existentes que pueden ser, además, distintos, lo que hace más atractiva su utilización. Como ejemplos de *middleware* se puede citar: DCE, DCOM, COM+ y Java RMI.

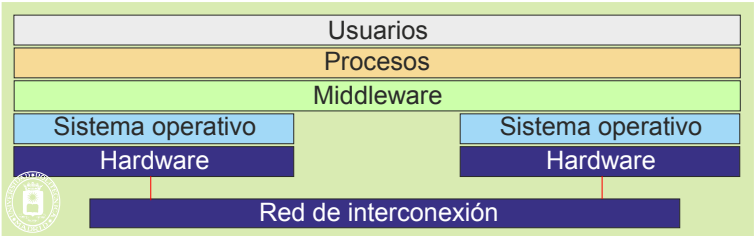


Figura 2.28 *Estructura de un middleware.*

2.9.2. Carga dinámica de módulos

En los sistemas operativos convencionales su configuración tenía que incluir la definición de todos los elementos *hardware* del sistema y de todos los módulos *software*. Cualquier cambio en estos elementos *hardware* o módulos *software* exigía la reconfiguración y rearranque del sistema operativo. En este sentido, se puede decir que la configuración era estática.

Con el desarrollo de los buses con capacidad de conexionado en caliente (esto es sin apagar el sistema), como son el bus USB, el bus PCMCIA o el bus Firewire, aparece la necesidad de la carga dinámica de módulos del sistema operativo. En efecto, al conectar un nuevo periférico es necesario cargar los módulos del sistema operativo que le dan servicio y dar de alta a dicho recurso para poder ser utilizado por los procesos. Una situación parecida puede producirse al instalarse un nuevo *software*. Todo ello da lugar a la configuración dinámica del sistema operativo.

Esta funcionalidad es especialmente interesante para los sistemas personales, que suelen sufrir cambios continuos. En los servidores no es muy recomendable usar reconfiguración dinámica por la posible inestabilidad que se puede producir. Es más recomendable comprobar primero en una máquina de pruebas que la nueva configuración funciona correctamente, antes de cambiar la configuración de la máquina de producción.

2.9.3. Prestaciones y fiabilidad

El sistema operativo es un *software* pesado, es decir, que requiere una importante cantidad de tiempo de procesador. Esto es inevitable, puesto que ha de realizar todo tipo de comprobaciones y acciones auxiliares antes de realizar cualquier función. Evidentemente, mientras más complejo sea y más protecciones tenga, mayor será el tiempo de procesador que exija.

Por eso es muy importante adecuar el sistema operativo a las necesidades reales del sistema y a la potencia de cálculo y almacenamiento del computador. No es lo mismo un sistema operativo para un computador personal que para un teléfono móvil.

Por otro lado, una consideración importante a la hora de programar es tener en cuenta que la invocación de los servicios del sistema operativo consume un tiempo apreciable de procesador. Nuestro programa ejecutará más rápido si reducimos el número de llamadas al sistema operativo, o dicho de otra manera, es menos costoso computacionalmente ejecutar una función por el proceso que solicitar que lo haga el sistema operativo.

2.9.4. Diseño del intérprete de mandatos

A pesar de que el modo de operación básico del intérprete de mandatos apenas ha cambiado con el tiempo, su estructura e implementación han evolucionado notablemente desde la aparición de los primeros sistemas de tiempo compartido hasta la actualidad. Como ya se ha comentado anteriormente, se pasó de tener el intérprete incluido en el sistema operativo a ser un módulo externo que usa los servicios del mismo, lo cual proporciona una mayor flexibilidad facilitando su modificación o incluso su reemplazo. Dentro de esta opción existen básicamente dos formas de estructurar el módulo que maneja la interfaz de usuario: intérprete con mandatos internos e intérprete con mandatos externos.

Intérprete con mandatos internos

El intérprete de mandatos es un único programa que contiene el código para ejecutar todos los mandatos. El intérprete, después de leer la línea tecleada por el usuario, determina de qué mandato se trata y salta a la parte de su código que lleva a cabo la acción especificada por dicho mandato. Si no se trata de ningún mandato, se interpreta que el usuario quiere arrancar una determinada aplicación, en cuyo caso el intérprete iniciará la ejecución del programa correspondiente en el contexto de un nuevo proceso y esperará hasta que termine. Con esta estrategia, mostrada en el programa 2.4, los mandatos son internos al intérprete. Observe que en esta sección se está suponiendo que hay un único mandato en cada línea.

Programa 2.4 Esquema de un intérprete con mandatos internos.

```
Repetir Bucle
   Escribir indicación de preparado
   Leer e interpretar línea. Obtiene operación y argumentos
   Caso operación
      Si "fin"
         Terminar ejecución de intérprete
      Si "renombrar"
         Renombrar ficheros según especifican argumentos
      Si "borrar"
         Borrar ficheros especificados por argumentos
      ...................................
      Si no (No se trata de un mandato conocido)
         Arrancar programa "operación" pasándole "argumentos"
         Esperar a que termine el programa
Fin Bucle
```

Intérprete con mandatos externos

En este caso, el intérprete no incluye a los mandatos y existe un programa independiente (fichero ejecutable) por cada mandato. El intérprete de mandatos no analiza la línea tecleada por el usuario, sino que directamente inicia la ejecución del programa correspondiente en el contexto de un nuevo proceso y espera que éste termine. Se realiza el mismo tratamiento ya se trate de un mandato o de cualquier otra aplicación. Con esta estrategia, mostrada en el programa 2.5, los mandatos son externos al intérprete y la interfaz de usuario está compuesta por un conjunto de programas del sistema: un programa por cada mandato más el propio intérprete.

Programa 2.5 Esquema de un intérprete con mandatos externos.

```
Repetir Bucle
   Escribir indicación de preparado
   Leer e interpretar línea. Obtiene operación y argumentos
      Si operación="fin"
         Terminar ejecución de intérprete
      Si no
         Arrancar programa "operación" pasándole "argumentos"
         Esperar a que termine el programa
```

Fin Bucle

La principal ventaja de la primera estrategia de diseño es ligeramente más eficiente, ya que los mandatos los lleva a cabo el propio intérprete sin necesidad de ejecutar programas adicionales. Sin embargo, el intérprete puede llegar a ser muy grande y la inclusión de un nuevo mandato, o la modificación de uno existente, exige cambiar el código del intérprete y recompilarlo. La segunda solución es más recomendable ya que proporciona un tratamiento y visión uniforme de los mandatos del sistema y las restantes aplicaciones. El intérprete no se ve afectado por la inclusión o la modificación de un mandato.

En los sistemas reales puede existir una mezcla de las dos estrategias. El intérprete de mandatos de MS-DOS (COMMAND.COM) se enmarca dentro de la primera categoría, esto es, intérprete con mandatos internos. El motivo de esta estrategia se debe a que este sistema operativo se diseñó para poder usarse en computadores sin disco duro y, en este tipo de sistema, el uso de un intérprete con mandatos externos exigiría que el disquete correspondiente estuviese insertado para ejecutar un determinado mandato. Sin embargo, dadas las limitaciones de memoria de MS-DOS, para mantener el tamaño del intérprete dentro de un valor razonable, algunos mandatos de uso poco frecuente, como por ejemplo DISKCOPY, se implementaron como externos.

Los intérpretes de mandatos de UNIX, denominados *shells*, se engloban en la categoría de intérpretes con mandatos externos. Sin embargo, algunos mandatos se tienen que implementar como internos debido a que su efecto sólo puede lograrse si es el propio intérprete el que ejecuta el mandato. Así, por ejemplo, el mandato `cd`, que cambia el directorio actual de trabajo del usuario usando la llamada `chdir`, requiere cambiar a su vez el directorio actual de trabajo del proceso que ejecuta el intérprete, lo cual sólo puede conseguirse si el mandato lo ejecuta directamente el intérprete.

Por su lado, las interfaces gráficas normalmente están formadas por un conjunto de programas que, usando los servicios del sistema, trabajan conjuntamente para llevar a cabo las peticiones del usuario. Así, por ejemplo, existirá un **gestor de ventanas** para mantener el estado de las mismas y permitir su manipulación, un **administrador de programas** que permita al usuario arrancar aplicaciones, un **gestor de ficheros** que permita manipular ficheros y directorios, o una **herramienta de configuración** de la propia interfaz y del entorno. Observe la diferencia con las interfaces alfanuméricas, en las que existía un programa por cada mandato.

2.10. HISTORIA DE LOS SISTEMAS OPERATIVOS

Como se decía al comienzo del capítulo, el sistema operativo lo forman un conjunto de programas que ayudan a los usuarios en la explotación de un computador, simplificando, por un lado, su uso, y permitiendo, por otro, obtener un buen rendimiento de la máquina. Es difícil tratar de dar una definición precisa de sistema operativo, puesto que existen muchos tipos, según sea la aplicación deseada, el tamaño del computador usado y el énfasis que se dé a su explotación. Por ello, se va a realizar un bosquejo de la evolución histórica de los sistemas operativos, ya que así quedará plasmada la finalidad que se les ha ido atribuyendo.

Se pueden encontrar las siguientes etapas en el desarrollo de los sistemas operativos, que coinciden con las cuatro generaciones de los computadores.

Prehistoria

Durante esta etapa, que cubre los años cuarenta, se construyen los primeros computadores. Como ejemplo de computadores de esta época se pueden citar el ENIAC (*Electronic Numerical Integrator Analyzer and Computer*) financiado por el Laboratorio de Investigación Balística de los Estados Unidos. El ENIAC era una máquina enorme con un peso de 30 toneladas, que era capaz de realizar 5.000 sumas por segundo, 457 multiplicaciones por segundo y 38 divisiones por segundo. Otro computador de esta época fue el EDVAC (*Electronic Discrete Variable Automatic Computer*).

En esta etapa no existían sistemas operativos. El usuario debía codificar su programa a mano y en instrucciones máquina, y debía introducirlo personalmente en el computador, mediante conmutadores o tarjetas perforadas. Las salidas se imprimían o se perforaban en cinta de papel para su posterior impresión. En caso de errores en la ejecución de los programas, el usuario tenía que depurarlos examinando el contenido de la memoria y los registros del computador.

En esta primera etapa todos los trabajos se realizaban en serie. Se introducía un programa en el computador, se ejecutaba y se imprimían los resultados. A continuación, se repetía este proceso con otro programa. Otro aspecto importante de esta época es que se requería mucho tiempo para preparar y ejecutar un programa, ya que el programador debía encargarse de todo: tanto de codificar todo el programa como de introducirlo en el computador de forma manual y de ejecutarlo.

Primera generación (años cincuenta)

Con la aparición de la primera generación de computadores (años cincuenta), se hace necesario racionalizar su explotación, puesto que ya empieza a haber una mayor base de usuarios. El tipo de operación seguía siendo en serie, como en el caso anterior, esto es, se trataba un trabajo detrás de otro, teniendo cada trabajo las fases siguientes:

- Instalación de cintas o fichas perforadas en los dispositivos periféricos. En su caso, instalación del papel en la impresora.

- Lectura mediante un programa **cargador** del programa a ejecutar y de sus datos.
- Ejecución del programa.
- Impresión o grabación de los resultados.
- Retirada de cintas, fichas y papel.

La realización de la primera fase se denominaba **montar** el trabajo.

El problema básico que abordaban estos sistemas operativos primitivos era optimizar el flujo de trabajos, minimizando el tiempo empleado en retirar un trabajo y montar el siguiente. También empezaron a abordar el problema de la E/S, facilitando al usuario paquetes de rutinas de E/S, para simplificar la programación de estas operaciones, apareciendo así los primeros **manejadores** de dispositivos. Se introdujo también el concepto de *system file name* que empleaba un nombre o número simbólico para referirse a los periféricos, haciendo que su manipulación fuera mucho más flexible que mediante las direcciones físicas.

Para minimizar el tiempo de montaje de los trabajos, estos se agrupaban en **lotes** (*batch*) del mismo tipo (p. ej.: programas Fortran, programas Cobol, etc.), lo que evitaba tener que montar y desmontar las cintas de los compiladores y montadores, aumentando el rendimiento (figura 2.29).

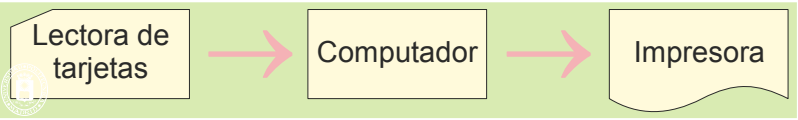


Figura 2.29 *Tratamiento por lotes. Se cargaba, por ejemplo, el compilador de Fortran y se ejecutaban varios trabajos.*

Como se muestra en la figura 2.30, en las grandes instalaciones se utilizaban computadores auxiliares o satélites para realizar las funciones de montar y retirar los trabajos. Así se mejoraba el rendimiento del computador principal, puesto que se le suministraban los trabajos montados en cinta magnética y ésta se limitaba a procesarlos y a grabar los resultados también en cinta magnética. En este caso se decía que la E/S se hacía **fuera de línea** (*off-line*).

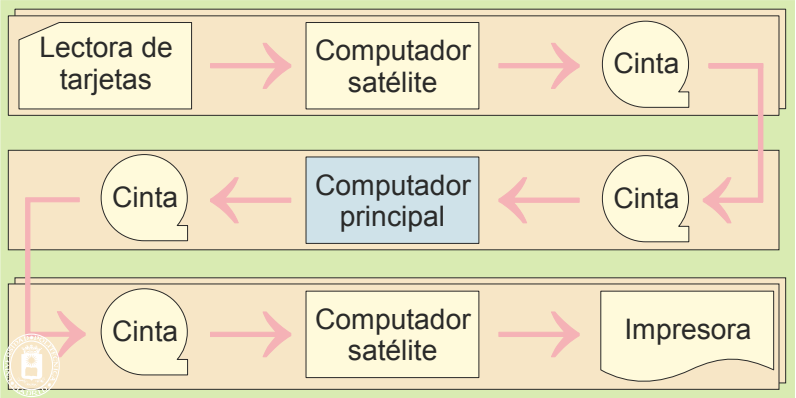


Figura 2.30 *Sistema de lotes con E/S off-line.*

Los sistemas operativos de las grandes instalaciones tenían las siguientes características:

- Procesaban un único flujo de trabajos en lotes.
- Disponían de un conjunto de rutinas de E/S.
- Usaban mecanismos rápidos para pasar de un trabajo al siguiente.
- Permitían la recuperación del sistema si un trabajo acababa en error.

Tenían un lenguaje de control de trabajos denominado JCL (*Job Control Language*). Las instrucciones JCL formaban la cabecera del trabajo y especificaban los recursos a utilizar y las operaciones a realizar por cada trabajo.

Como ejemplos de sistemas operativos de esta época se pueden citar el FMS (*Fortran Monitor System*), el IBYSS y el sistema operativo de la IBM 7094.

Segunda generación (años sesenta)

Con la aparición de la segunda generación de computadores (principios de los sesenta), se hizo más necesario, dada la mayor competencia entre los fabricantes, mejorar la explotación de estas máquinas de altos precios. La multiprogramación se impuso en sistemas de lotes como una forma de aprovechar el tiempo empleado en las operaciones de E/S. La base de estos sistemas reside en la gran diferencia que existe, como se vio en el capítulo "1 Conceptos arquitectónicos del computador", entre las velocidades de los periféricos y de la UCP, por lo que esta última, en las operaciones de E/S, se pasa mucho tiempo esperando a los periféricos. Una forma de aprovechar ese tiempo consiste en mantener varios trabajos simultáneamente en memoria principal (técnica llamada de multiprogramación), y en realizar las operaciones de E/S por acceso directo a memoria. Cuando un trabajo necesita una operación de E/S la solicita al sistema operativo que se encarga de:

- Congelar el trabajo solicitante.
- Iniciar la mencionada operación de E/S por DMA.
- Pasar a realizar otro trabajo residente en memoria. Estas operaciones las realiza el sistema operativo multiprogramado de forma transparente al usuario.

La disponibilidad de periféricos que trabajan por DMA permitió difundir la técnica del SPOOL (*Simultaneous Peripheral Operations On-Line*). Los trabajos se van cargando en disco y posteriormente se ejecutan en lote. Los resultados se depositan en disco, para ser enviados posteriormente a la impresora.

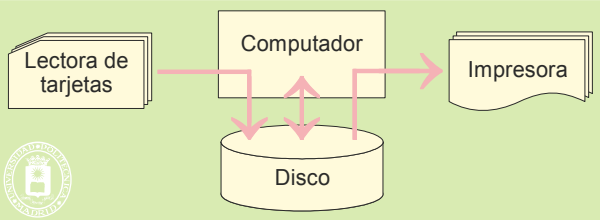


Figura 2.31 *Técnica de SPOOL basada en disco magnético.*

También en esta época aparecieron otros modos de funcionamiento muy importantes:

- Se construyeron los primeros **multiprocesadores**, en los que varios procesadores formaban una sola máquina de mayores prestaciones.
- Surgió el concepto de **servicio** del sistema operativo, siendo el sistema operativo Atlas I Supervisor de la Universidad de Manchester el primero en utilizarlo.
- Se introdujo el concepto de **independencia de dispositivo**. El usuario ya no tenía que referirse en sus programas a una unidad de cinta magnética o a una impresora en concreto. Se limitaba a especificar que quería grabar un fichero determinado o imprimir unos resultados. El sistema operativo se encargaba de asignarle, de forma dinámica, una unidad disponible, y de indicar al operador del sistema la unidad seleccionada, para que éste montara la cinta o el papel correspondiente.
- Comenzaron los sistemas de **tiempo compartido** o ***timesharing***. Estos sistemas, a los que estamos muy acostumbrados en la actualidad, permiten que varios usuarios trabajen de forma **interactiva** o **conversacional** con el computador desde terminales, que en aquellos días eran teletipos electromecánicos. El sistema operativo se encarga de repartir el tiempo de la UCP entre los distintos usuarios, asignando de forma rotativa pequeños intervalos de tiempo de UCP denominadas **rodajas** (*time slice*). En sistemas bien dimensionados, cada usuario tiene la impresión de que el computador le atiende exclusivamente a él, respondiendo rápidamente a sus órdenes. Aparecen así los primeros planificadores. El primer sistema de tiempo compartido fue el CTSS (*Compatible Time-Sharing System*) desarrollado por el MIT en 1961. Este sistema operativo se utilizó en un IBM 7090 y llegó a manejar hasta 32 usuarios interactivos.
- En esta época aparecieron los primeros sistemas de **tiempo real**. Se trataba de aplicaciones militares, en concreto para detección de ataques aéreos. En este caso, el computador está conectado a un sistema externo y debe responder velozmente a las necesidades de ese sistema externo. En este tipo de sistema las respuestas deben producirse en periodos de tiempo previamente especificados, que en la mayoría de los casos son pequeños. Los primeros sistemas de este tipo se construían en ensamblador y ejecutaban sobre máquina desnuda, lo que hacía de estas aplicaciones sistemas muy complejos.

Finalmente, cabe mencionar que Burroughs introdujo, en 1963, el «*Master Control Program*», que además de ser multiprograma y multiprocesador incluía memoria virtual y ayudas para depuración en lenguaje fuente.

Durante esta época se desarrollaron, también, los siguientes sistemas operativos: El OS/360, sistema operativo utilizado en las máquinas de la línea 360 de IBM; y el sistema MULTICS, desarrollado en el MIT con participación de los laboratorios Bell. MULTICS fue diseñado para dar soporte a cientos de usuarios; sin embargo, aunque una versión primitiva de este sistema operativo ejecutó en 1969 en un computador GE 645, no proporcionó los servicios para los que fue diseñado y los laboratorios Bell finalizaron su participación en el proyecto.

Tercera generación (años setenta)

La tercera generación es la época de los sistemas de propósito general y se caracteriza por los sistemas operativos multimodo de operación, esto es, capaces de operar en lotes, en multiprogramación, en tiempo real, en tiempo compartido y en modo multiprocesador. Estos sistemas operativos fueron costosísimos de realizar e interpusieron entre el usuario y el *hardware* una gruesa capa de *software*, de forma que éste sólo veía esta capa, sin tenerse que preocupar de los detalles de la circuitería.

Uno de los inconvenientes de estos sistemas operativos era su complejo lenguaje de control, que debían aprenderse los usuarios para preparar sus trabajos, puesto que era necesario especificar multitud de detalles y opciones. Otro de los inconvenientes era el gran consumo de recursos que ocasionaban, esto es, los grandes espacios de memoria principal y secundaria ocupados, así como el tiempo de UCP consumido, que en algunos casos superaba el 50% del tiempo total.

Esta década fue importante por la aparición de dos sistemas que tuvieron una gran difusión, UNIX y MVS de IBM. De especial importancia es UNIX, cuyo desarrollo empieza en 1969 por Ken Thompson, Dennis Ritchie y otros sobre un PDP-7 abandonado en un rincón en los Bell Laboratories. Su objetivo fue diseñar un sistema sencillo en reacción contra la complejidad del MULTICS. Pronto se transportó a una PDP-11, para lo cual se reescribió utilizando el lenguaje de programación C. Esto fue algo muy importante en la historia de los sistemas operativos, ya que hasta la fecha ninguno se había escrito utilizando un lenguaje de alto nivel, recurriendo para ello a los lenguajes ensambladores propios de cada arquitectura. Sólo una pequeña parte de UNIX, aquella que accedía de forma directa al *hardware*, siguió escribiéndose en ensamblador. La programación de un sistema operativo utilizando un lenguaje de alto nivel como es C, hace que un sistema operativo sea fácilmente transportable a una amplia gama de computado-

res. En la actualidad, prácticamente todos los sistemas operativos se escriben en lenguajes de alto nivel, fundamentalmente en C.

La primera versión ampliamente disponible de UNIX fue la versión 6 de los Bell Laboratories, que apareció en 1976. A esta le siguió la versión 7 distribuida en 1978, antecesora de prácticamente todas las versiones modernas de UNIX. En 1982 aparece una versión de UNIX desarrollada por la Universidad de California en Berkeley, la cual se distribuyó como la versión BSD (*Berkeley Software Distribution*) de UNIX. Esta versión de UNIX introdujo mejoras importantes como la inclusión de memoria virtual y la interfaz de *sockets* para la programación de aplicaciones sobre protocolos TCP/IP.

Más tarde AT&T (propietario de los Bell Laboratories) distribuyó la versión de UNIX conocida como System V o RVS4. Desde entonces, muchos han sido los fabricantes de computadores que han adoptado a UNIX como sistema operativo de sus máquinas. Ejemplos de estas versiones son: Solaris de SUN, HP UNIX, IRIX de SGI y AIX de IBM.

Cuarta generación (años ochenta hasta la actualidad)

La cuarta generación se caracteriza por una evolución de los sistemas operativos de propósito general de la tercera generación, tendente a su especialización, a su simplificación y a dar más importancia a la productividad del usuario que al rendimiento de la máquina.

Adquiere cada vez más importancia el tema de las redes de computadores, tanto de redes de largo alcance como locales. En concreto, la disminución del coste del *hardware* hace que se difunda el **proceso distribuido**, en contra de la tendencia centralizadora anterior. El proceso distribuido consiste en disponer de varios computadores, cada uno situado en el lugar de trabajo de las personas que las emplean, en lugar de una sola central. Estos computadores suelen estar unidos mediante una red, de forma que puedan compartir información y periféricos.

Se difunde el concepto de **máquina virtual**, consistente en que un computador X, que incluye su sistema operativo, sea simulado por otro computador Y. Su ventaja es que permite ejecutar, en el computador Y, programas preparados para el computador X, lo que posibilita el empleo de *software* elaborado para el computador X, sin necesidad de disponer de dicho computador.

Durante esta época, los sistemas de **bases de datos** sustituyen a los ficheros en multitud de aplicaciones. Estos sistemas se diferencian de un conjunto de ficheros en que sus datos están estructurados de tal forma que permiten acceder a la información de diversas maneras, evitar datos redundantes, y mantener su integridad y coherencia.

La difusión de los computadores personales ha traído una humanización en los sistemas informáticos. Aparecen los sistemas «amistosos» o ergonómicos, en los que se evita que el usuario tenga que aprenderse complejos lenguajes de control, sustituyéndose éstos por los sistemas dirigidos por menú, en los que la selección puede incluso hacerse mediante un manejador de cursor. En estos sistemas, de orientación monousuario, el objetivo primario del sistema operativo ya no es aumentar el rendimiento del sistema, sino la productividad del usuario.

Los sistemas operativos que dominaron el campo de los computadores personales fueron UNIX, MS-DOS y los sucesores de Microsoft para este sistema: Windows 95/98, Windows NT y Windows 200X. El incluir un intérprete de BASIC en la ROM del PC de IBM ayudó a la gran difusión del MS-DOS, al permitir a las máquinas sin disco que arrancasen directamente dicho intérprete. La primera versión de Windows NT (versión 3.1) apareció en 1993 e incluía la misma interfaz de usuario que Windows 3.1. En 1996 apareció la versión 4.0, que se caracterizó por incluir, dentro del ejecutivo de Windows NT, diversos componentes gráficos que ejecutaban anteriormente en modo usuario. Durante el año 2000, Microsoft distribuyó la versión denominada Windows 2000 que más tarde pasó a ser Windows 2002, Windows XP y Windows 2003.

También tiene importancia durante esta época el desarrollo de GNU/Linux. Linux es un sistema operativo de la familia UNIX, desarrollado de forma desinteresada durante la década de los 90 por miles de voluntarios conectados a Internet. Linux está creciendo fuertemente debido sobre todo a su bajo coste y a su gran estabilidad, comparable a cualquier otro sistema UNIX. Una de las principales características de Linux es que su código fuente está disponible, lo que le hace especialmente atractivo para el estudio de la estructura interna de un sistema operativo. Su aparición ha tenido también mucha importancia en el mercado del *software* ya que ha hecho que se difunda el concepto de *software* libre y abierto.

Durante esta etapa se desarrollaron también los **sistemas operativos de tiempo real**, encargados de ofrecer servicios especializados para el desarrollo de aplicaciones de tiempo real. Algunos ejemplos son: QNX, RTEMS y VRTX.

A mediados de los ochenta, aparecieron varios **sistemas operativos distribuidos** experimentales, que no despegaron como productos comerciales. Como ejemplo de sistemas operativos distribuidos se puede citar: Mach, Chorus y Amoeba.

Los sistemas operativos distribuidos dejaron de tener importancia y fueron evolucionando durante la década de los 90 a lo que se conoce como *middleware*. Dos de los *middleware* más importantes de esta década han sido DCE y CORBA. Microsoft también ofreció su propio *middleware* conocido como DCOM que ha evolucionado al COM+. En el entorno Java se ha desarrollado el RMI.

Durante esta etapa es importante el desarrollo del estándar UNIX, que define la interfaz de programación del sistema operativo. Este estándar persigue que las distintas aplicaciones que hagan uso de los servicios de un sistema operativo sean portables sin ninguna dificultad a distintas plataformas con sistemas operativos diferentes. Cada vez es mayor el número de sistemas operativos que ofrecen esta interfaz. Otra de las interfaces de programación más utilizada es la interfaz Windows, interfaz de los sistemas operativos Windows 95/98, Windows NT, Windows 2000 y sucesivos.

Interfaces gráficas

Las primeras experiencias con este tipo de interfaces se remontan a los primeros años de la década de los setenta. En Xerox PARC (un centro de investigación de Xerox) se desarrolló lo que actualmente se considera la primera estación de trabajo a la que se denominó *Alto*. Además de otros muchos avances, esta investigación estableció los primeros pasos en el campo de los GUI.

Con la aparición, al principio de los ochenta, de los computadores personales dirigidos a usuarios no especializados, se acentuó la necesidad de proporcionar este tipo de interfaces. Así, la compañía Apple adoptó muchas de las ideas de la investigación de Xerox PARC para lanzar su computador personal Macintosh (1984) con una interfaz gráfica que simplificaba enormemente el manejo del computador. El otro gran competidor en este campo, el sistema operativo MS-DOS, tardó bastante más en dar este paso. En sus primeras versiones proporcionaba una interfaz alfanumérica similar a la de UNIX pero muy simplificada. Como paso intermedio, hacia 1988, incluyó una interfaz denominada DOS-*shell* que, aunque seguía siendo alfanumérica, no estaba basada en líneas, sino que estaba orientada al uso de toda la pantalla y permitía realizar operaciones mediante menús. Por fin, ya en los noventa, lanzó una interfaz gráfica, denominada Windows, que tomaba prestadas muchas de las ideas del Macintosh.

En el mundo UNIX se produjo una evolución similar. Cada fabricante incluía en su sistema una interfaz gráfica además de la convencional. La aparición del sistema de ventanas X a mediados de los ochenta y su aceptación generalizada, que le ha convertido en un estándar *de facto*, ha permitido que la mayoría de los sistemas UNIX incluyan una interfaz gráfica común. Como resultado de este proceso, prácticamente todos los computadores de propósito general existentes actualmente poseen una interfaz de usuario gráfica.

La tendencia actual consiste en utilizar sistemas operativos multiprogramados sobre los que se añade un **gestor de ventanas**, lo que permite que el usuario tenga activas, en cada momento, tantas tareas como desee y que los distribuya, a su antojo, sobre la superficie del terminal. Este tipo de interfaces tiene su mayor representante en los sistemas operativos Windows de Microsoft. En la figura 2.32 se muestra uno de los elementos clave de la interfaz gráfica de este tipo de sistemas, el explorador de Windows, que da acceso a los recursos de almacenamiento y ejecución del sistema.

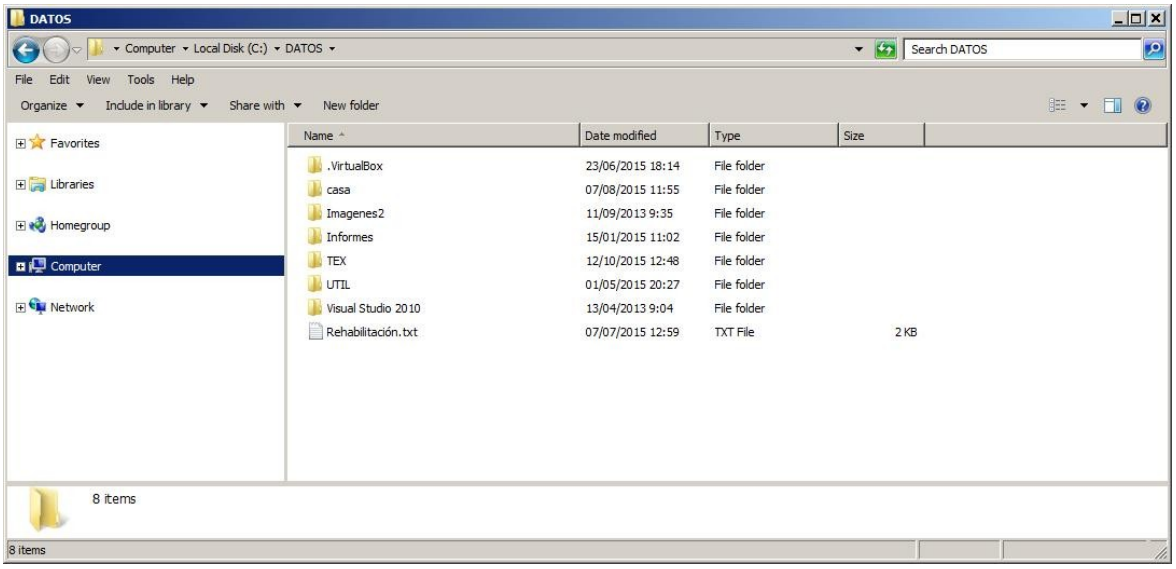


Figura 2.32 *Explorador de Windows*

La revolución móvil (desde finales de los noventa)

El final de los años noventa se han caracterizado por el despegue de los sistemas móviles y conectados por radio. Buena prueba de ello son los dispositivos PDA, *pocket* PC, teléfonos móviles y los innumerables dispositivos industriales remotos. Para estos sistemas personales aparece una nueva clase de sistema operativo, más simple pero incorporando características multimedia.

Por su lado, los dispositivos industriales remotos utilizan sistemas operativos de tiempo real.

El futuro previsible es que la evolución de los sistemas operativos se va a seguir orientando hacia las plataformas distribuidas y la computación móvil. Gran importancia consideramos que tendrá la construcción de sistemas operativos y entornos que permitan utilizar sistemas de trabajo heterogéneos (computadores fijos y dispositivos móviles de diferentes fabricantes con sistemas operativos distintos), conectados por redes de interconexión. Estos sistemas se han de poder utilizar como una gran máquina centralizada, lo que permitirá disponer de una mayor capacidad de cómputo, y han de facilitar el trabajo cooperativo entre los distintos usuarios.

La ley de Edholm [Cherry, 2004] estipula que el ancho de banda disponible en un computador se duplica cada 18 meses y que la tendencia es a tener el 100% de los computadores conectados. La figura 2.33 muestra esta ley para comunicación por cable y por radio.

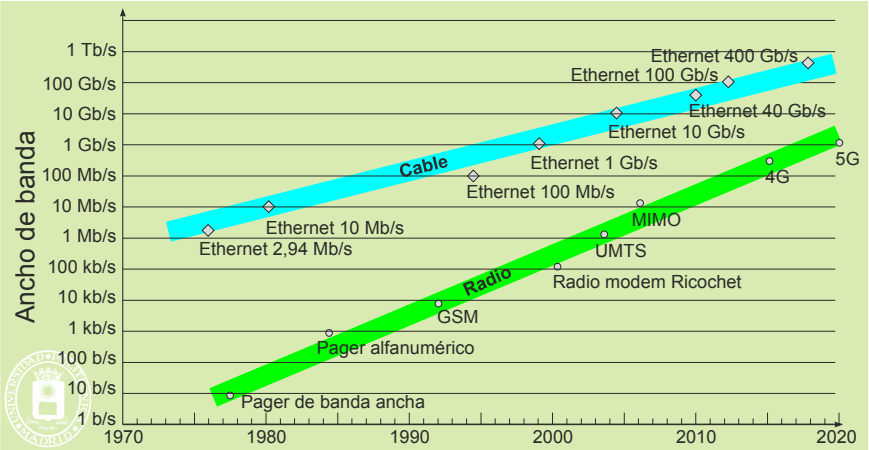


Figura 2.33 *Evolución de los anchos de banda disponibles por cable y por radio.*

2.11. LECTURAS RECOMENDADAS

Son muchos los libros sobre sistemas operativos que cubren los temas tratados en este capítulo. Algunos de ellos son [Crowley, 1997], [Milenkovic, 1992], [Silberchatz, 2005], [Stallings, 2001] y [Tanenbaum, 2006]. Sobre sistemas operativos distribuidos puede consultarse [Tanenbaum, 2002] y [Galli, 1999]. En cuanto a sistemas operativos concretos, en [Bach, 1986] y [McKusick, 1996] se describe UNIX System V y la versión 4.4 BSD de UNIX respectivamente; en [Solomon, 1998] se describe la estructura interna de Windows NT, y en [Beck, 1998] se describe la de Linux.

2.12. EJERCICIOS

1. ¿Cuáles son las principales funciones de un sistema operativo?

2. ¿Qué diferencia existe entre un mandato y una llamada al sistema?

3. Definir los términos de visión externa e interna de un sistema operativo. ¿Cuál de las dos determina mejor a un sistema operativo concreto? ¿Por qué?

4. ¿Cuántas instrucciones de la siguiente lista deben ejecutarse exclusivamente en modo privilegiado? Razone su respuesta.
   a) Inhibir todas las interrupciones.
   b) Escribir en los registros de control de un controlador de DMA.
   c) Leer el estado de un controlador de periférico.
   d) Escribir en el reloj del computador.
   e) Provocar un TRAP o interrupción *software*.
   f) Escribir en los registros de la MMU.

5. Sea un sistema multitarea sin memoria virtual que tiene una memoria principal de 24 MiB. Conociendo que la parte residente del sistema operativo ocupa 5 MiB y que cada proceso ocupa 3 MiB, calcular el número de procesos que pueden estar activos en el sistema.

6. ¿Cómo se solicita una llamada al sistema operativo?

7. Indique algunos ejemplos que muestren la necesidad de que el sistema operativo ofrezca mecanismos de comunicación y sincronización entre procesos.

8. ¿Por qué no se puede invocar al sistema operativo utilizando una instrucción de tipo CALL?

9. ¿Cómo indica UNIX en un programa C el tipo de error que se ha producido en una llamada al sistema? ¿y Windows?

10. ¿Cuál de las siguientes técnicas *hardware* tiene mayor influencia en la construcción de un sistema operativo? Razone su respuesta.
   a) Microprogramación del procesador.
   b) cache de la memoria principal.
   c) DMA.
   d) RISC.

11. ¿Qué diferencias existe entre una lista de control de acceso y una *capability*?

12. ¿El intérprete de mandatos de UNIX es interno o externo? Razone su respuesta con un ejemplo.

13. ¿Dónde es más compleja una llamada al sistema, en un sistema operativo monolítico o en uno por capas?

14. ¿Qué tipo de sistema operativo es más fácil de modificar, uno monolítico o uno por capas? ¿Cuál es más eficiente?

1. ¿Es un proceso un fichero ejecutable? Razone su respuesta.

2. ¿Debe un sistema operativo multitarea ser de tiempo compartido? ¿Y viceversa? Razone su respuesta.

3. ¿Qué diferencias existen entre un fichero y un directorio?

4. ¿Qué ventajas considera que tiene escribir un sistema operativo utilizando un lenguaje de alto nivel?

5. ¿En qué época se introdujeron los primeros manejadores de dispositivos? ¿Y los sistemas operativos de tiempo compartido?

3

PROCESOS

Este capítulo se dedica a estudiar todos los aspectos relacionados con los procesos, conceptos fundamentales para comprender el funcionamiento de un sistema operativo. Los temas que se tratan en este capítulo son:

- *Procesos.*
- *Multitarea.*
- *Información del proceso.*
- *Vida del proceso.*
- *Threads.*
- *Planificación.*
- *Señales y excepciones.*
- *Temporizadores.*
- *Tipos de procesos.*
- *Conceptos de ejecución del sistema operativo.*
- *Tratamiento de las interrupciones.*
- *Estructuras del sistema operativo.*
- *Servicios UNIX y Windows relacionados con la gestión de procesos.*

3.1. CONCEPTO DE PROCESO

El objetivo fundamental de un computador es el de ejecutar programas, por lo que el objetivo principal del sistema operativo será facilitar la ejecución de esos programas. El proceso se puede definir como un programa puesto en ejecución por el sistema operativo y, de una forma más precisa, como **la unidad de procesamiento gestionada por el sistema operativo**.

Para ejecutar un programa, como se vio en capítulo "1 Conceptos arquitectónicos del computador", éste ha de residir con sus datos en el mapa de memoria, formando lo que se denomina **imagen de memoria**. Además, el sistema operativo mantiene una serie de estructuras de información por cada proceso, estructuras que permiten identificar al proceso y conocer sus características y los recursos que tiene asignados. Una parte muy importante de estas informaciones se encuentra en el **bloque de control del proceso** (BCP) que tiene asignado cada proceso. En la sección "3.3 Información del proceso" se analiza con detalle la información asociada a cada proceso.

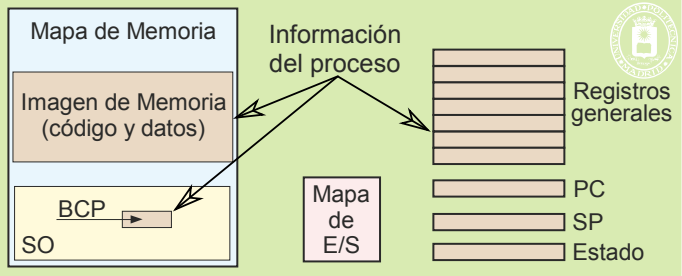


Figura 3.1 *Elementos que constituyen un proceso.*

Jerarquía de procesos

La secuencia de creación de procesos vista en la sección "2.2.2 Arranque del sistema operativo" genera un árbol de procesos como el incluido en la figura 3.2.

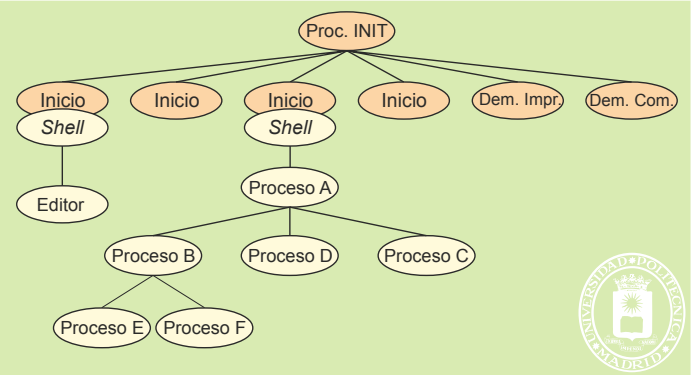


Figura 3.2 *Jerarquía de procesos.*

Para referirse a las relaciones entre los procesos de la jerarquía se emplean los términos de padre e hijo (a veces se emplea el de hermano y abuelo). Cuando el proceso A solicita al sistema operativo que cree el proceso B se dice que A es padre de B y que B es hijo de A. Bajo esta óptica la jerarquía de procesos puede considerarse como un árbol genealógico.

Algunos sistemas operativos como UNIX mantienen de forma explícita esta estructura jerárquica de procesos —un proceso sabe quién es su padre—, mientras que otros sistemas operativos como Windows no la mantienen.

Vida de un proceso

La vida de un proceso se puede descomponer en las siguientes fases:

- El proceso **se crea**, cuando otro proceso solicita al sistema operativo un servicio de creación de proceso (o al arrancar el sistema operativo).
- El proceso **ejecuta**. El objetivo del proceso es precisamente ejecutar. Esta ejecución consiste en:
  - rachas de procesamiento.
  - rachas de espera.
- El proceso **termina** o **muere**. Cuando el proceso termina de una forma no natural se puede decir que el proceso aborta.

Más adelante se retomará el tema de la vida de un proceso, para analizar con más detalle cada una de sus fases.

Entorno del proceso

El entorno del proceso consiste en un conjunto de variables que se le pasan al proceso en el momento de su creación. El entorno está formado por una tabla NOMBRE=VALOR que se incluye en la pila del proceso. El NOMBRE especifica el nombre de la variable y el VALOR su valor. Un ejemplo de entorno en UNIX es el siguiente:

PATH=/usr/bin /home/pepe/bin
TERM=vt100
HOME=/home/pepe
PWD=/home/pepe/libros/primero

PATH indica la lista de directorios en los que el sistema operativo busca los programas ejecutables, TERM el tipo de terminal, HOME el directorio inicial asociado al usuario y PWD el directorio de trabajo actual.

El sistema operativo ofrece servicios para leer, modificar, añadir o eliminar variables de entorno. Por lo que los procesos pueden utilizar las variables del entorno para definir su comportamiento. Por ejemplo, un programa de edición analizará la variable TERM, que especifica el terminal, para interpretar adecuadamente las teclas que pulse el usuario.

Grupos de procesos

Los procesos forman grupos que tienen diversas propiedades. El conjunto de procesos creados a partir de un *shell* puede formar un grupo de procesos. También pueden formar un grupo los procesos asociados a un terminal.

El interés del concepto de grupo de procesos es que hay determinadas operaciones que se pueden hacer sobre todos los procesos de un determinado grupo, como se verá al estudiar algunos de los servicios. Un ejemplo es la posibilidad de terminar todos los procesos pertenecientes a un mismo grupo.

3.2. MULTITAREA

Como se vio en la sección “2.4 Tipos de sistemas operativos”, dependiendo del número de procesos que pueda ejecutar simultáneamente, un sistema operativo puede ser monotarea o multitarea.

3.2.1. Base de la multitarea

La multitarea se basa en las tres características siguientes:

- Paralelismo real entre E/S y procesador.
- Alternancia en los procesos de fases de E/S y de procesamiento.
- Memoria capaz de almacenar varios procesos.

En la sección “1.6.4 Conceptos arquitectónicos del computador” se vio que existe concurrencia real entre el procesador y las funciones de E/S realizadas por los controladores de los periféricos. Esto significa que, mientras se está realizando una operación de E/S de un proceso, se puede estar ejecutando otro proceso.

Como se muestra en la figura 3.3, la ejecución de un proceso alterna fases de procesamiento con fases de E/S, puesto que, cada cierto tiempo, necesita leer o escribir datos en un periférico. En un sistema monotarea el procesador no tiene nada que hacer durante las fases de entrada/salida, por lo que desperdicia su potencia de procesamiento. En un sistema multitarea se aprovechan las fases de entrada/salida de unos procesos para realizar las fases de procesamiento de otros.

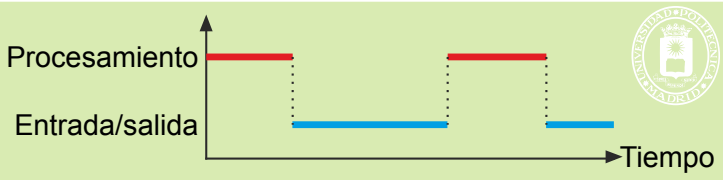


Figura 3.3 *Un proceso alterna fases de procesamiento y de entrada/salida.*

La figura 3.4 presenta un ejemplo de ejecución multitarea con cuatro procesos activos. Observe que, al finalizar la segunda fase de procesamiento del proceso C, hay un intervalo de tiempo en el que no hay trabajo para el procesador, puesto que todos los procesos están bloqueados.

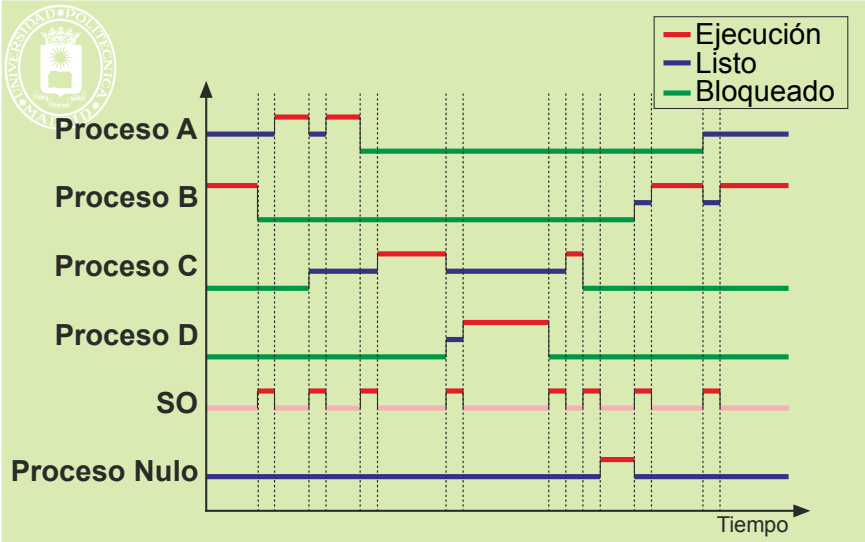


Figura 3.4 *Ejemplo de ejecución en un sistema multitarea.*

Como muestra la figura anterior, el sistema operativo entra a ejecutar al final de las fases de procesamiento y al final de las fases de entrada/salida. Esto es así puesto que las operaciones de E/S no las gobiernan directamente los procesos, sino que se limitan a pedir al sistema operativo que las realice. Del mismo modo, el sistema operativo trata las interrupciones externas que generan los controladores para avisar que han completado una operación.

Finalmente, es importante destacar que la multitarea exige tener más de un proceso activo y cargado en memoria. Por tanto, hay que disponer de suficiente memoria para albergar a estos procesos.

Proceso nulo

Como se ha indicado en la sección "1.2.2 Secuencia de funcionamiento del procesador", el procesador no para nunca de ejecutar. Esto parece que contradice la figura 3.4, puesto que muestra un intervalo en el que el procesador no tiene nada que hacer. Para evitar esta contradicción los sistemas operativos incluyen el denominado proceso nulo. Este proceso consiste en un bucle infinito que no realiza ninguna operación útil. El objetivo de este proceso es «entretener» al procesador cuando no hay ninguna otra tarea. En una máquina multiprocesadora es necesario disponer de un proceso nulo por procesador.

Estados de los procesos

De acuerdo con la figura 3.3, un proceso puede estar en determinadas situaciones (ejecución, listo y bloqueado), que denominaremos estados. A lo largo de su vida, el proceso va cambiando de estado según evolucionan sus necesidades. En la sección "3.4.5 Estados básicos del proceso", página 81, se describirán con mayor detalle los estados de un proceso.

3.2.2. Ventajas de la multitarea

La multiprogramación presenta varias ventajas, entre las que se pueden resaltar las siguientes:

- Facilita la programación. Permite dividir las aplicaciones en varios procesos, lo que beneficia su modularidad.
- Permite prestar un buen servicio, puesto que se puede atender a varios usuarios de forma eficiente, interactiva y simultánea.
- Aprovecha los tiempos muertos que los procesos pasan esperando a que se completen sus operaciones de E/S.
- Aumenta el uso de la UCP, al aprovechar los intervalos de tiempo que los procesos están bloqueados.

Todas estas ventajas hacen que, salvo para situaciones muy especiales, no se conciba actualmente un sistema operativo que no soporte la multitarea.

Grado de multiprogramación y necesidades de memoria principal

Se denomina grado de multiprogramación al número de procesos activos que mantiene un sistema. El grado de multiprogramación es un factor que afecta de forma importante al rendimiento que se obtiene de un computador. Mientras más procesos activos haya en un sistema, mayor es la probabilidad de encontrar siempre un proceso en estado de listo para ejecutar, por lo que entrará a ejecutar menos veces el proceso nulo. Sin embargo, a mayor grado de multiprogramación, mayores son las necesidades de memoria. Veamos este fenómeno con más detalle para los dos casos de tener o no tener memoria virtual.

En un sistema con **memoria real** los procesos activos han de residir totalmente en memoria principal. Por tanto, el grado de multiprogramación viene limitado por el tamaño de los procesos y por la memoria principal disponible. Además, como se indica en la figura 3.5, el rendimiento de la utilización del procesador aumenta siempre con el

grado de multiprogramación (pero, al ir aumentando el grado de multiprogramación hay que aumentar la memoria principal). Esto es así ya que los procesos siempre residen en memoria principal.

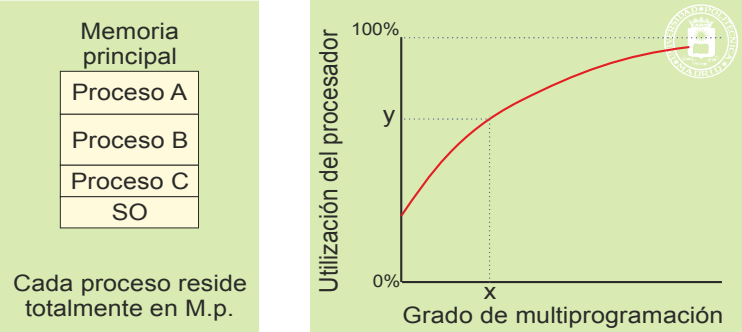


Figura 3.5 *Grado de multiprogramación y rendimiento del procesador. En un sistema real con suficiente memoria al aumentar el grado de multiprogramación aumenta la utilización del procesador.*

En los sistemas **con memoria virtual** la situación es más compleja, puesto que los procesos sólo tienen en memoria principal su conjunto residente (recordatorio 3.1), lo que hace que quepan más procesos. Sin embargo, al aumentar el número de procesos, sin aumentar la memoria principal, disminuye el conjunto residente de cada uno, situación que se muestra en la figura 3.6.

Recordatorio 3.1. Se denomina **conjunto residente** a las páginas que un proceso tiene en memoria principal y **conjunto de trabajo** al conjunto de páginas que un proceso está realmente utilizando en un instante determinado.

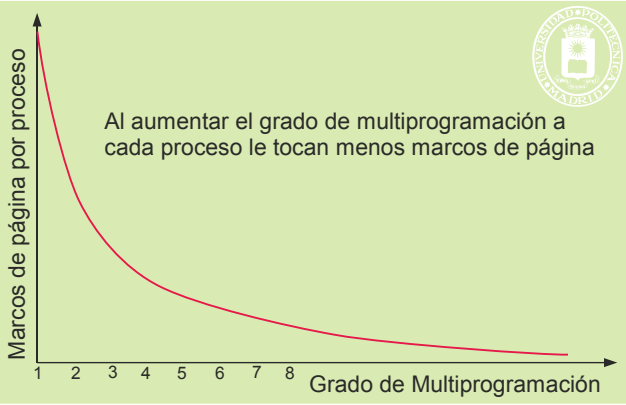


Figura 3.6 *Para una cantidad de memoria principal dada, el conjunto residente medio decrece con el grado de multiprogramación*

Cuando el conjunto residente de un proceso se hace menor de un determinado valor, ya no representa adecuadamente al conjunto de trabajo del proceso, lo que tiene como consecuencia que se produzcan muchos fallos de página. Cada fallo de página consume tiempo de procesador, porque el sistema operativo ha de tratar el fallo, y tiempo de E/S, puesto que hay que hacer una migración de páginas. Todo ello hace que, al crecer los fallos de páginas, el sistema dedique cada vez más tiempo al improductivo trabajo de resolver estos fallos de página. Se denomina **hiperpaginación** (*trashing*) a la situación de alta paginación producida cuando los conjuntos residentes de los procesos son demasiado pequeños.

3.3. INFORMACIÓN DEL PROCESO

Como se indicó anteriormente, el proceso es la unidad de procesamiento gestionada por el sistema operativo. Para poder realizar este cometido el proceso tiene asociado una serie de elementos de información, que se resumen en la figura 3.7, y que se organizan en tres grupos: estado del procesador, imagen de memoria y tablas del sistema operativo.

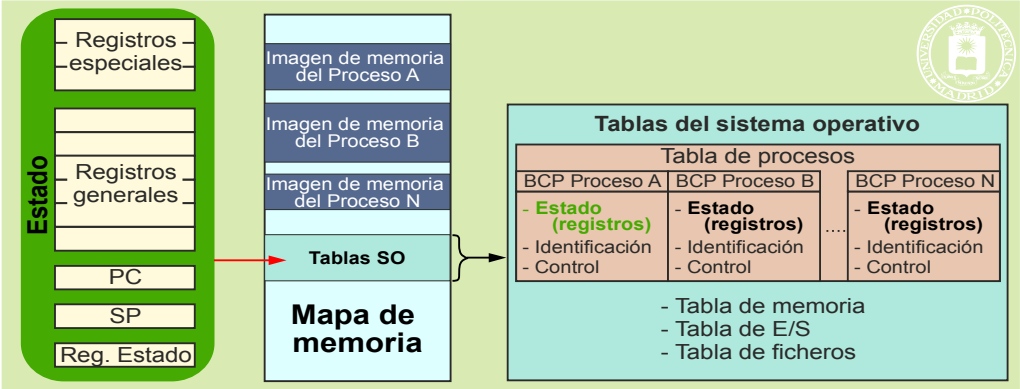


Figura 3.7 *Información del proceso.*

Es de destacar que el proceso no incluye información de E/S, puesto que ésta suele estar reservada al sistema operativo.

3.3.1. Estado del procesador

Como ya sabemos, el estado del procesador está formado por el contenido de todos sus registros (aclaración 3.1), que se enumeran seguidamente, mientras que el estado visible del procesador se refiere al contenido de los registros accesibles en modo usuario.

- Registros accesibles en modo usuario:
  - Registros generales. De existir registros específicos de coma flotante también se incluyen aquí.
  - Contador de programa.
  - Puntero o punteros de pila.
  - Parte del registro de estado accesible en modo usuario.
- Registros especiales solamente accesibles en modo privilegiado:
  - Parte del registro de estado accesible en modo privilegiado.
  - Registros de control de memoria. Como puede ser los registros de borde o el RIED (Registro Identificador de Espacio de Direccionamiento).

Aclaración 3.1. No confundir el estado del procesador con el estado del proceso.

Cuando el proceso no está en ejecución, su estado debe estar almacenado en el bloque de control de proceso (BCP). Por el contrario, cuando el proceso está ejecutando, el estado del procesador reside en los registros y varía de acuerdo al flujo de instrucciones máquina ejecutado. En este caso, la copia que reside en el BCP no está actualizada. Téngase en cuenta que los registros de la máquina se utilizan para no tener que acceder a la información de memoria, dado que es mucho más lenta que éstos.

Sin embargo, cuando se detiene la ejecución de un proceso, para ejecutar otro proceso, es muy importante que el sistema operativo actualice la copia del estado del procesador en el BCP. En términos concretos, la rutina del sistema operativo que trata las interrupciones lo primero que ha de hacer es salvar el estado del procesador del proceso interrumpido en su BCP.

3.3.2. Imagen de memoria del proceso

La imagen de memoria del proceso está formada por los espacios de memoria que éste está autorizado a usar. Las principales características de la imagen de memoria son las siguientes:

- El proceso solamente puede tener información en su imagen de memoria y no fuera de ella. Si genera una dirección que esté fuera de ese espacio, el *hardware* de protección deberá detectarlo y generar una excepción *hardware* síncrona de violación de memoria. Esta excepción activará la ejecución del sistema operativo, que se encargará de tomar las acciones oportunas, como puede ser abortar la ejecución del proceso.
- Dependiendo del computador, la imagen de memoria estará referida a memoria virtual o a memoria real. Observe que esto es transparente (irrelevante) para el proceso, puesto que él genera direcciones de memoria, que serán tratados como virtuales o reales según el caso.
- Los procesos suelen necesitar asignación dinámica de memoria. Por tanto, la imagen de memoria de los mismos se deberá adaptar a estas necesidades, creciendo o decreciendo adecuadamente.
- No hay que confundir la asignación de memoria con la asignación de marcos de memoria. El primer término contempla la modificación de la imagen de memoria y se refiere al espacio virtual en los sistemas con este tipo de espacio. El segundo término sólo es de aplicación en los sistemas con memoria virtual y se refiere a la modificación del conjunto residente del proceso.

El sistema operativo asigna la memoria al proceso, para lo cual puede emplear distintos modelos de imagen de memoria, que se analizan seguidamente.

Proceso con una única región de tamaño fijo

Es el modelo más sencillo de imagen de memoria y su uso se suele restringir a los sistemas con memoria real. El proceso recibe un único espacio de memoria que, además, no puede variar de tamaño.

Proceso con una única región de tamaño variable

Se puede decir que esta solución no se emplea. En sistemas con memoria real las regiones no pueden crecer, a menos que se deje espacio de memoria principal de reserva —en caso contrario se solaparía con el proceso siguiente—. Ahora bien, la memoria principal es muy cara como para dejarla de reserva. En sistemas con memoria virtual sí se podría emplear, dado que el espacio de reserva no tiene asignados recursos físicos, pero es más conveniente usar un modelo de varias regiones, pues es mucho más flexible y se adapta mejor a las necesidades reales de los procesos.

Proceso con un número fijo de regiones de tamaño variable

Un proceso contiene varios tipos de información, cuyas características se analizan seguidamente:

- **Texto o código**. Bajo este nombre se considera el programa máquina que ha de ejecutar el proceso. Aunque el programa podría automodificarse, no es ésta una práctica recomendada, por lo cual se considerará que esta información es fija y que solamente se harán operaciones de ejecución y lectura sobre ella (aclaración 3.2).
- **Datos**. Este bloque de información depende mucho de cada proceso. Los lenguajes de programación actuales permiten asignación dinámica de memoria, lo que hace que varíe el tamaño del bloque de datos al avanzar la ejecución del proceso. Cada programa estructura sus datos de acuerdo a sus necesidades, pudiendo existir los siguientes tipos:
  - Datos con valor inicial. Estos datos son estáticos y su valor se fija al cargar el proceso desde el fichero ejecutable. Estos valores se asignan en tiempo de compilación.
  - Datos sin valor inicial. Estos datos son estáticos, pero no tienen valor asignado, por lo que no están presentes en el fichero ejecutable. Será el sistema operativo el que, al cargar el proceso, rellene o no rellene esta zona de datos con valores predefinidos.
  - Datos dinámicos. Estos datos se crean y se destruyen de acuerdo a las directrices del programa.
  - Los datos podrán ser de lectura-escritura o solamente de lectura.
- **Pila**. A través del puntero de pila, los programas utilizan una estructura de pila residente en memoria. En ella se almacenan, por ejemplo, los bloques de activación de los procedimientos llamados. La pila es una estructura dinámica, puesto que crece y decrece según avanza la ejecución del proceso. Recordemos que hay procesadores que soportan una pila para modo usuario y otra para modo privilegiado.

Aclaración 3.2. Dado que también se pueden incluir cadenas inmutables de texto en el código, también se han de permitir las operaciones de lectura, además de las de ejecución.

Para adaptarse a estas informaciones, se puede utilizar un modelo de imagen de memoria con un número fijo de regiones de tamaño variable.

El modelo tradicional utilizado en UNIX contempla tres regiones: texto, pila y datos. La figura 3.8 presenta esta solución. Observe que la región de texto es de tamaño fijo (el programa habitualmente no se modifica) y que las regiones de pila y de datos crecen en direcciones contrarias.

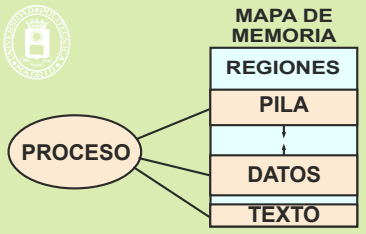


Figura 3.8 *Modelo de imagen de memoria con estructura de regiones fija.*

Este modelo se adapta bien a los sistemas con memoria virtual, puesto que el espacio virtual reservado para que puedan crecer las regiones de datos y pila no existe físicamente. Solamente se crea, gastando recursos de disco y memoria, cuando se asigna. No ocurrirá lo mismo en un sistema real, en el que el espacio reservado para el crecimiento ha de existir como memoria principal, dando como resultado que hay un recurso costoso que no se está utilizando.

Si bien este modelo es más flexible que los dos anteriores, tiene el inconveniente de no prestar ayuda para la estructuración de los datos. El sistema operativo ofrece un espacio de datos que puede crecer y decrecer, pero deja al programa la gestión interna de este espacio.

Proceso con un número variable de regiones de tamaño variable

Esta solución es más avanzada que la anterior, al permitir que existan las regiones que desee el proceso. La figura 3.9 presenta un caso de 7 regiones, que podrán ser de texto, de pila o de datos.

Es la solución más flexible y, por tanto, la utilizada en las versiones actuales de Windows y UNIX.

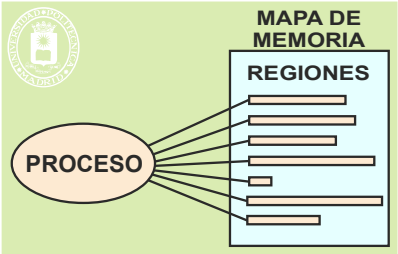


Figura 3.9 *Modelo de imagen de memoria con estructura de regiones variable.*

3.3.3. Información del bloque de control de proceso (BCP)

Cada proceso dispone, en la tabla de procesos, de un BCP que contiene la información básica del proceso, entre la que cabe destacar la siguiente:

Información de identificación

Información de identificación del usuario y del proceso. Como ejemplo, en UNIX se incluyen los siguientes datos:

- Identificador del proceso: pid del proceso.
- Identificador del padre del proceso: pid del padre.
- Identificador de usuario real: uid real.
- Identificador de grupo real: gid real.
- Identificador de usuario efectivo: uid efectivo.
- Identificador de grupo efectivo: gid efectivo.
- Identificadores de grupos de procesos a los que pertenece (el proceso puede pertenecer a un único grupo o a varios grupos).

Estado del procesador

Contiene los valores iniciales del estado del procesador o su valor en el instante en que fue expulsado el proceso.

Información de control del proceso

En esta sección se incluye diversa información que permite gestionar al proceso. Destacaremos los siguientes datos, muchos de los cuales se detallarán a lo largo del libro:

- Información de planificación y estado.
  - Estado del proceso.
  - Evento por el que espera el proceso cuando está bloqueado.
  - Prioridad del proceso.
  - Información de planificación.
- Descripción de las regiones de memoria asignadas al proceso.
- Recursos asignados, tales como:
  - Ficheros abiertos (tabla de descriptores o manejadores de fichero).
  - Puertos de comunicación asignados.
  - Temporizadores.
- Punteros para estructurar los procesos en colas o anillos. Por ejemplo, los procesos que están en estado de listo pueden estar organizados en una cola, de forma que se facilite la labor del planificador.
- Comunicación entre procesos. El BCP puede contener espacio para almacenar las señales y para algún mensaje enviado al proceso.
- Señales
  - Señales armadas.
  - Máscara de señales.
- Información de contabilidad o uso de recursos
  - Tiempo de procesador consumido.
  - Operaciones de E/S realizadas.

3.3.4. Información del proceso fuera del BCP

No toda la información que mantiene el sistema operativo en relación con un proceso se almacena en el BCP. Existen las dos siguientes razones para ello. Si la información se quiere compartir entre varios procesos, no puede estar en el BCP, puesto que esta estructura es privativa de cada proceso. Si la información tiene tamaños muy diferentes de un proceso a otro, no es eficiente almacenarla en el BCP. Veamos algunos ejemplos:

- Tabla de páginas. La tabla de páginas de un proceso, que describe en detalle la imagen de memoria del mismo en un sistema de memoria virtual, se pone fuera del BCP. En el BCP se mantiene información que permite acceder a dicha tabla y, en algunos casos, información global de la imagen de memoria. Hay que tener en cuenta que el tamaño de la tabla de páginas varía enormemente de un proceso a otro y que, además, para que dos procesos compartan memoria han de compartir un trozo de tabla de páginas.
- Punteros de posición de ficheros. Para realizar las operaciones de escritura y lectura sobre un fichero el sistema operativo mantiene un puntero que indica la posición por la que está accediendo al fichero. Dado que, en algunos casos, este puntero se quiere compartir entre varios procesos, esta información no puede almacenarse en el BCP. Por otro lado, no siempre se quiera compartir, por lo que tampoco puede asociarse al nodo-i en UNIX o el equivalente en otros sistemas. Por ello, se establece una estructura compartida por todos los procesos, denominada tabla intermedia, que contiene dichos punteros. Dicha tabla se estudiará en el capítulo “5 E/S y Sistema de ficheros”.

3.4. VIDA DE UN PROCESO

Como ya sabemos, la vida de un proceso consiste en su creación, su ejecución y su muerte o terminación. Sin embargo, la ejecución del proceso no se suele hacer de un tirón, puesto que surgen interrupciones y el propio proceso puede necesitar servicios del sistema operativo. De acuerdo con ello, en esta sección se analizarán de forma general la creación, interrupción, activación y terminación del proceso.

3.4.1. Creación del proceso

La creación de un proceso la realiza el sistema operativo bajo petición expresa de otro proceso (con excepción del o de los procesos iniciales creados en el arranque del sistema operativo). Esta creación consiste en completar todas las informaciones que constituyen un proceso, como se muestra en la figura 3.10.

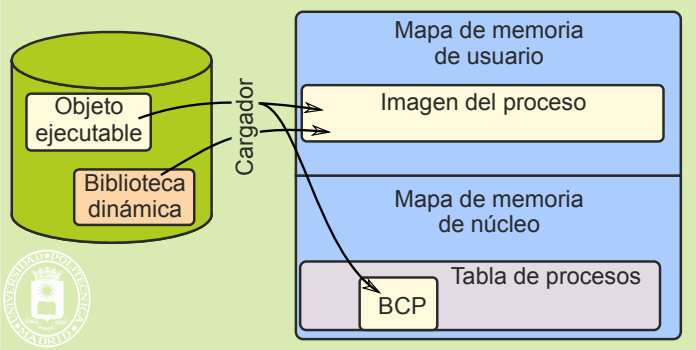


Figura 3.10 *Formación de un proceso.*

Formación de un proceso.

De forma más específica, las operaciones que debe hacer el sistema operativo son las siguientes:

- Asignar un espacio de memoria para albergar la imagen de memoria. En general, este espacio será virtual y estará compuesto de varias regiones.
- Seleccionar un BCP libre de la tabla de procesos.
- Rellenar el BCP con la información de identificación del proceso, con la descripción de la memoria asignada, con los valores iniciales de los registros indicados en el fichero objeto, etc.
- Cargar en la región de texto el código más las rutinas de sistema y en región de datos los datos iniciales contenidos en el fichero objeto.
- Crear en la región de pila la pila inicial del proceso. La pila incluye inicialmente el entorno del proceso y los argumentos que se pasan en la invocación del mandato.

Una vez completada toda la información del proceso, se puede marcar como listo para ejecutar, de forma que el planificador, cuando lo considere oportuno, lo seleccione para su ejecución. Una vez seleccionado, el activador lo pone en ejecución.

Creación de un proceso en UNIX

En UNIX la creación de un nuevo proceso consiste en clonar el proceso padre, por lo que no se sigue el modelo general planteado anteriormente. Para ello se utiliza el servicio fork que se detalla en la página 113.

3.4.2. Interrupción del proceso

Mientras está ejecutando un proceso puede ocurrir interrupciones. Veamos detalladamente los pasos involucrados:

- Un proceso está en ejecución, por lo tanto, parte de su información reside en los registros de la máquina, que están siendo constantemente modificados por la ejecución de sus instrucciones máquina.
- Bien sea porque llega una interrupción externa, una excepción *hardware* o porque el proceso solicita un servicio del sistema operativo, el proceso para su ejecución.
- Inmediatamente entra a ejecutar el sistema operativo ya sea para atender la interrupción o para atender el servicio demandado.
- La ejecución del sistema operativo, como la de todo programa, modifica los contenidos de los registros de la máquina, destruyendo sus valores anteriores.

Según la secuencia anterior, si más adelante se desea continuar con la ejecución del proceso, se presenta un grave problema: los registros ya no contienen los valores que deberían. Supongamos que el proceso está ejecutando la secuencia siguiente:

LD .5,#CANT

⇐ En este punto llega una interrupción y se pasa al SO

LD .1,[.5]

Supongamos que el valor de `CANT` es `HexA4E78`, pero que el sistema operativo al ejecutar modifica el registro `.5`, dándole el valor `HexEB7A4`. Al intentar, más tarde, que siga la ejecución del mencionado proceso, la instrucción «`LD .1,[.5]`» cargará en el registro `.1` el contenido de la dirección `HexEB7A4` en vez del contenido de la dirección `HexA4E78`.

Para evitar esta situación, lo primero que hace el sistema operativo al entrar a ejecutar es salvar el contenido de todos los registros de usuario, teniendo cuidado de no modificar el valor de ninguno de ellos antes de salvarlo. Como muestra la figura 3.11, al interrumpirse la ejecución de un proceso el sistema operativo almacena los contenidos de los registros en el BCP de ese proceso (recordatorio 3.2). Para más detalles ver la sección "3.10.2 Detalle del tratamiento de interrupciones".

Recordatorio 3.2. Como sabemos, la propia interrupción modifica el contador de programa y el registro de estado (cambia el bit que especifica el modo de ejecución para pasar a modo privilegiado, así como los bits de inhibición de interrupciones). Sin embargo, esto no presenta ningún problema puesto que el propio *hardware* se encarga de salvar estos registros en la pila antes de modificarlos.

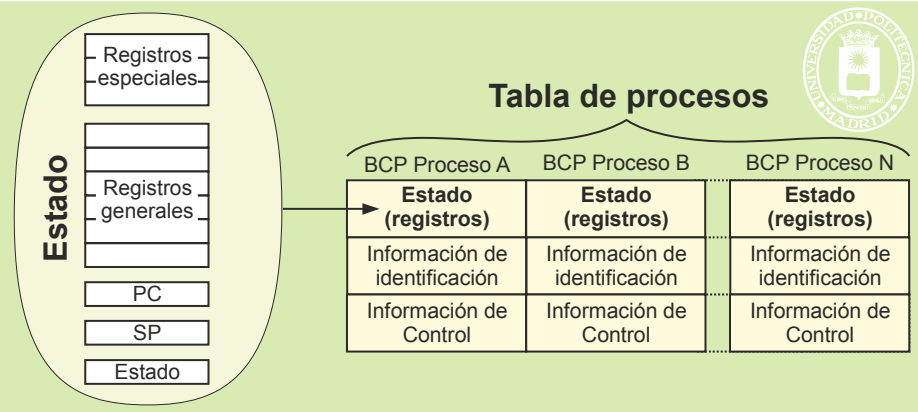


Figura 3.11 *Al interrumpirse la ejecución de un proceso, se salva su estado en el BCP.*

3.4.3. Activación del proceso

Activar un proceso es ponerlo en ejecución. Para que la activación funcione correctamente ha de garantizar que el proceso encuentra el computador exactamente igual a como lo dejó, para que pueda seguir ejecutando sin notar ninguna diferencia.

El módulo del sistema operativo que pone a ejecutar un proceso se denomina **activador** o ***dispatcher***. La activación de un proceso consiste en copiar en los registros del procesador el estado del procesador, que está almacenado en su BCP. De esta forma, el proceso continuará su ejecución en las mismas condiciones en las que fue parado. El activador termina con una instrucción de retorno de interrupción (p. ej.: `RETI`). El efecto de esta instrucción es restituir el registro de estado y el contador de programa, lo cual tiene los importantes efectos siguientes:

- Al restituir el registro de estado, se restituye el bit que especifica el modo de ejecución. Dado que cuando fue salvado este registro el bit indicaba modo usuario, puesto que estaba ejecutando un proceso de usuario, su restitución garantiza que el proceso seguirá ejecutando en modo usuario.
- Igualmente, al restituir el registro de estado se restituyen los bits de inhibición de interrupción, según los tenía el proceso cuando fue interrumpido.
- Al restituir el contador de programa, se consigue que la siguiente instrucción máquina que ejecute el procesador sea justo la instrucción en la que fue interrumpido el proceso. En este momento es cuando se ha dejado de ejecutar el sistema operativo y se pasa a ejecutar el proceso.

3.4.4. Terminación del proceso

Cuando el proceso termina, ya sea porque ha completado su ejecución o porque se ha decidido que debe morir, el sistema operativo tiene que recuperar los recursos que tiene asignando el proceso. Al hacer esta recuperación hay que tener en cuenta dos posibles situaciones:

- Si el recurso está asignado en exclusividad al proceso, como puede ser el BCP o una región de datos no compartidos, el sistema operativo debe añadir el recurso a sus listas de recursos libres.
- Si el recurso es compartido, el sistema operativo tiene que tener asociado un contador, para llevar la cuenta del número de procesos que lo están utilizando. El sistema operativo decrementará dicho contador y, solamente cuando alcance el valor «0», deberá añadir el recurso a sus listas de recursos libres.

3.4.5. Estados básicos del proceso

Como muestra la figura 3.3, página 73, no todos los procesos activos de un sistema multitarea están en la misma situación. Se diferencian, por tanto, tres estados básicos en los que puede estar un proceso, estados que detallamos seguidamente:

- **Ejecución**. El proceso está ejecutando en el procesador, es decir, está en fase de procesamiento. En esta fase el estado del procesador reside en los registros del procesador.
- **Bloqueado**. Un proceso bloqueado está esperando a que ocurra un evento y no puede seguir ejecutando hasta que suceda dicho evento. Una situación típica de proceso bloqueado se produce cuando el proceso solicita una operación de E/S u otra operación que requiera tiempo. Hasta que no termina esta operación el proceso queda bloqueado. En esta fase el estado del procesador está almacenado en el BCP.
- **Listo**. Un proceso está listo para ejecutar cuando puede entrar en fase de procesamiento. Dado que puede haber varios procesos en este estado, una de las tareas del sistema operativo será seleccionar aquél que debe pasar a ejecución. El módulo del sistema operativo que toma esta decisión se denomina **planificador**. En esta fase el estado del procesador está almacenado en el BCP.

La figura 3.12 presenta estos tres estados, indicando algunas de las posibles transiciones entre ellos. Puede observarse que sólo hay un proceso en estado de ejecución, puesto que el procesador solamente ejecuta un programa en cada instante (aclaración 3.3). Del estado de ejecución se pasa al estado de bloqueado al solicitar, por ejemplo, una operación de E/S (flecha de Espera evento). También se puede pasar del estado de ejecución al de listo cuando el sistema operativo decida que ese proceso lleva mucho tiempo en ejecución o cuando pase a listo un proceso más prioritario (flecha de Expulsado). Del estado de bloqueado se pasa al estado de listo cuando se produce el evento por el que estaba esperando el proceso (p. ej.: cuando se completa la operación de E/S solicitada). Finalmente, del estado de listo se pasa al de ejecución cuando el planificador lo seleccione para ejecutar. Todas las transiciones anteriores están gobernadas por el sistema operativo, lo que implica la ejecución del mismo en dichas transiciones.

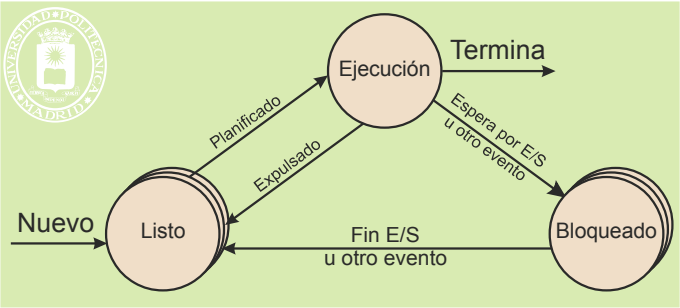


Figura 3.12 *Estados básicos de un proceso.*

Aclaración 3.3. En una máquina multiprocesador se tendrán simultáneamente en estado de ejecución tantos procesos como procesadores.

3.4.6. Estados de espera y suspendido

Además de los estados básicos vistos en las secciones anteriores, los procesos pueden estar en los estados de espera y de suspendido. Completando el esquema de la figura 3.12 con estos nuevos estados se obtiene el diagrama representado en la figura 3.13.

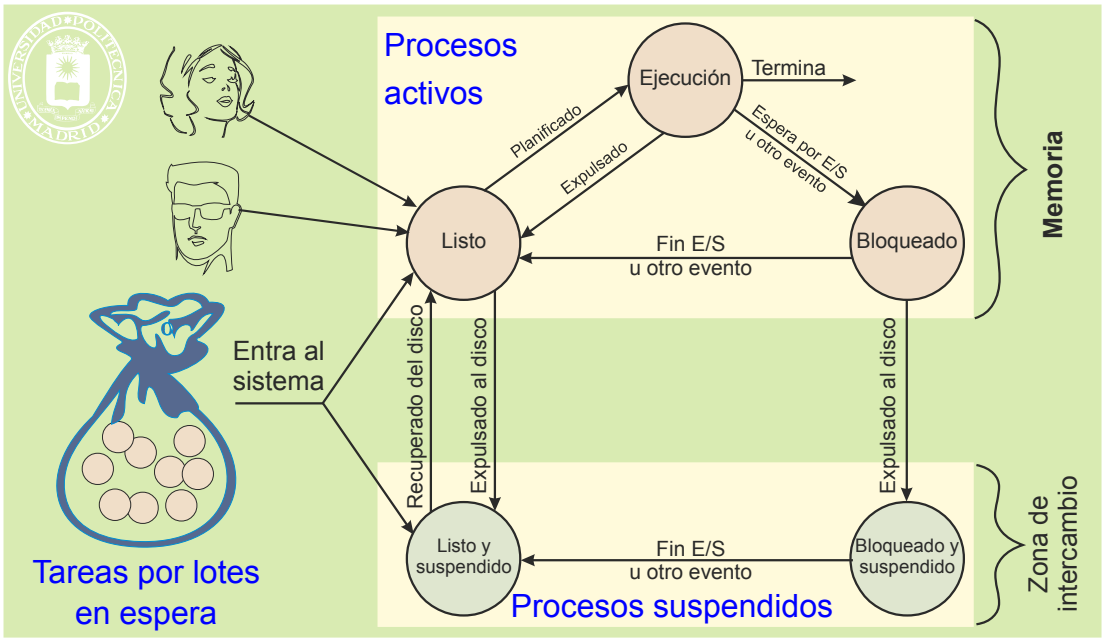


Figura 3.13 *Diagrama completo con los estados de un proceso.*

Los procesos entran en el sistema porque lo solicita un proceso, como puede ser el *shell*, o porque está prevista su ejecución *batch*. Es frecuente tener una lista de procesos *batch* **en espera** para ser ejecutados cuando se pueda. El sistema operativo ha de ir analizando dicha lista para lanzar la ejecución de los procesos a medida que disponga de los recursos necesarios.

Los procesos salen del sistema cuando mueren, es decir, al ejecutar el servicio correspondiente o al producir algún error irrecuperable.

El sistema operativo puede **suspender** algunos procesos para disminuir el grado de multiprogramación efectivo, lo que implica que les retira todos sus marcos de páginas, dejándolos enteramente en la zona de intercambio. En la figura 3.13 se muestra como los procesos listos o bloqueados pueden suspenderse. El objetivo de la suspensión estriba en dejar suficiente memoria principal a los procesos no suspendidos para que su conjunto residente tenga un tamaño adecuado que evite la hiperpaginación (ver capítulo "4 Gestión de memoria").

No todos los sistemas operativos tienen la opción de suspensión. Por ejemplo, un sistema operativo monousuario puede no incluir la suspensión, dejando al usuario la labor de cerrar procesos si observa que no ejecutan adecuadamente.

Los procesos *batch* que entran en el sistema lo pueden hacer pasando al estado de listo o al de listo suspendido.

3.4.7. Cambio de contexto

El cambio de contexto consiste en pasar de ejecutar un proceso A a ejecutar otro proceso B. El cambio de contexto exige dos cambios de modo. El primer cambio de modo viene producido por una interrupción, pasándose a ejecutar el sistema operativo. El segundo cambio de modo lo realiza el sistema operativo al activar el proceso B.

El cambio de contexto es una operación relativamente costosa, puesto que hay que cambiar de tablas de páginas, hay que renovar la TLB, etc.

Cambio de modo por interrupción (proceso → SO)

Este cambio de modo se produce cuando está ejecutando un proceso A y llega una interrupción, lo que implica pasar a ejecutar el sistema operativos.

Este cambio de modo exige almacenar el estado del procesador, es decir, los contenidos de los registros del procesador, puesto que el sistema operativo los va a utilizar, modificando su contenido. Por ejemplo, un programa que sea interrumpido entre las dos instrucciones de máquina siguientes:

```
LD  .5,#CANT
            → llega una interrupción y se pasa al SO.
LD  .1,[.5]
```

Si no se salvase y restituyese posteriormente el contenido del registro 5, la instrucción LD .1,[.5] estría utilizando un valor erróneo.

La figura 3.14 muestra el BCP con el espacio para salvar los registros.

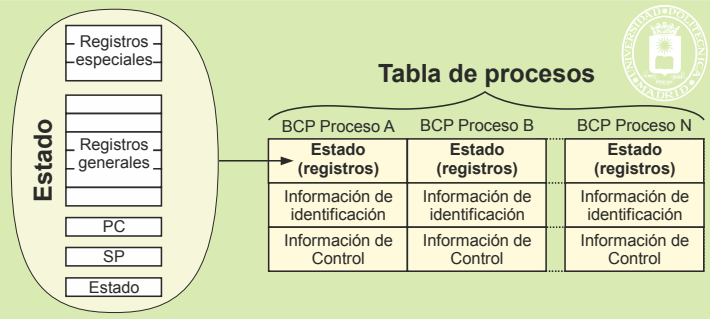


Figura 3.14 *El cambio de modo exige almacenar el estado del procesador. Por ello, el BCP incluye espacio para realizar esta operación.*

Cambio de modo por activación (SO→ proceso)

El cambio de modo se produce cuando el sistema operativo activa el proceso B para su ejecución. Este cambio de modo implica restaurar los registros del procesador con los valores almacenados anteriormente en el BCP del proceso B.

3.4.8. Privilegios del proceso UNIX

Como muestra la figura 3.15, un proceso UNIX tiene un UID real y otro efectivo, así como un GID real y otro efectivo. El UID y GID real se corresponden con los del dueño que creó el proceso y, en general, los valores efectivos son iguales que los reales. Sin embargo, si un ejecutable UNIX tiene activo el bit SETUID al ejecutar un servicio exec con dicho fichero, el proceso cambia su UID efectivo, pasando a tomar el valor del UID del dueño del fichero. De forma similar, si está activo el bit SETGID del fichero ejecutable, el proceso cambia su GID efectivo por el valor del GID del dueño del fichero.

Figura 3.15 *Un proceso UNIX tiene UID y GID real y efectivo.*

El SO utiliza el UID y GID efectivos para determinar los privilegios del proceso. Así si el proceso quiere abrir un fichero se seguirá el algoritmo de la figura 2.21, página 52, con los valores UID y GID efectivos del proceso.

3.5. SEÑALES Y EXCEPCIONES

Cuando un sistema operativo desea notificar a un proceso la aparición de un determinado evento, o error, recurre al mecanismo de las señales en UNIX o al de las excepciones en Windows.

3.5.1. Señales UNIX

Desde el punto de vista del proceso, una señal es un evento que recibe (a través del sistema operativo). La señal:

- Interrumpe al proceso
- Le transmite información muy limitada (un número, que identifica el tipo de señal)
- Un proceso también puede enviar señales a otros procesos (del mismo grupo), mediante el servicio kill().

Desde el punto de vista del sistema operativo una señal se envía a un único proceso. El origen puede ser el propio sistema operativo u otro proceso:

- SO → proceso
- proceso → proceso

Las señales tienen frente al proceso un efecto similar al que tienen las interrupciones frente al procesador. Las señales se utilizan para avisar al proceso de un evento. Por ejemplo, el proceso padre recibe una señal SIGCHLD cuando termina un hijo o una seña SIGILL cuando intenta ejecutar una instrucción máquina no permitida.

El proceso que recibe una señal se comporta, como muestra la figura 3.16, de la siguiente forma:

- En caso de estar ejecutando, el proceso detiene su ejecución en la instrucción máquina actual.
- Bifurca a ejecutar una rutina de tratamiento de la señal, cuyo código ha de formar parte del propio proceso.
- Una vez ejecutada la rutina de tratamiento, si ésta no termina el proceso, sigue la ejecución del proceso en la instrucción en la que fue interrumpido.

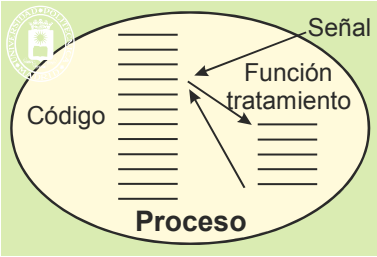


Figura 3.16 *Recepción de una señal por parte de un proceso.*

El origen de una señal puede ser un proceso o el sistema operativo.

Señal proceso → proceso

Un proceso puede enviar una señal a otro proceso que tenga el mismo identificador de usuario (uid), pero no a los que lo tengan distinto (aclaración 3.4). Un proceso también puede mandar una señal a un grupo de procesos, que han de tener su mismo uid.

Aclaración 3.4. Un proceso del superusuario puede mandar una señal a cualquier proceso, con independencia de su uid.

Señal sistema operativo → proceso

El sistema operativo también toma la decisión de enviar señales a los procesos cuando ocurren determinadas condiciones. Por ejemplo, ciertas excepciones *hardware* síncronas las convierte el sistema operativo en las correspondientes señales equivalentes destinadas al proceso que ha causado la excepción.

Tipos de señales

Existen muchas señales diferentes, cada una de las cuales tiene su propio significado, indicando un evento distinto. A título de ejemplo, se incluyen aquí las tres categorías de señales siguientes:

- Excepciones *hardware* síncronas. por ejemplo:
  - Instrucción ilegal.
  - Violación de memoria.
  - Desbordamiento en operación aritmética
- Comunicación.
- E/S asíncrona o no bloqueantes.

Efecto, armado y máscara de la señal

Como se ha indicado anteriormente, el efecto de la señal es ejecutar una rutina de tratamiento. Para que esto sea así, el proceso debe tener armado ese tipo de señal, es decir, ha de estar preparado para recibir dicho tipo de señal.

Armar una señal significa indicar al sistema operativo el nombre de la rutina del proceso que ha de tratar ese tipo de señal, lo que, como veremos, se consigue en UNIX con el servicio `sigaction`. Algunas señales admiten que se las ignore, lo cual ha de ser indicado al sistema operativo mediante el servicio `sigaction`. En este caso, el sistema operativo simplemente desecha las señales ignoradas por ese proceso.

El proceso tiene una **máscara** que permite enmascarar diversos tipos de señales. Cuando llega una señal correspondiente a uno de los tipos enmascarados, la señal queda bloqueada (no se desecha), a la espera de que el proceso desenmascare ese tipo de señal o las ignore.

Cuando un proceso recibe una señal sin haberla armado o enmascarado previamente, se ejecuta la acción por defecto, que en la mayoría de los casos consiste en terminar al proceso (3.5).

Aclaración 3.5. Conviene notar en este punto que el servicio UNIX para enviar señales se llama `kill`, porque puede usarse para terminar o matar procesos.

La figura 3.17 resume las alternativas de comportamiento del proceso frente a una señal. La señal puede ser ignorada, con lo que el proceso sigue su ejecución sin hacer caso de la señal. La señal puede ser tratada: el proceso ejecuta la rutina de tratamiento establecida y termina o continua con su ejecución en el punto en el que llegó la señal. Cuando el proceso no trata ni ignora la señal, se realiza el tratamiento por defecto, que suele consistir en terminar el proceso, generando o no un volcado de memoria, aunque para algunas señales consiste en no hacer nada y seguir con la ejecución del proceso.

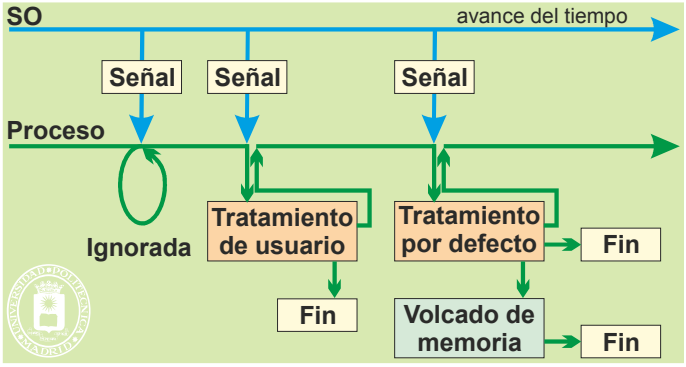


Figura 3.17 *Efecto de las señales sobre la ejecución de un proceso.*

Comportamiento de las señales en el fork y exec

Al ejecutar un FORK

- El hijo hereda el armado de señales del padre.
- El hijo hereda las señales ignoradas.
- El hijo hereda la máscara de señales.
- La alarma se cancela en el hijo.
- Las señales pendientes no son heredadas.

Al ejecutar un EXEC

- El armado desaparece pasándose a la acción por defecto (ya no existe la función de armado).
- Las señales ignoradas se mantienen.
- La máscara de señales se mantiene.
- La alarma se mantiene.
- Las señales pendientes siguen pendientes.

3.5.2. Excepciones Windows

Una excepción es un evento que ocurre durante la ejecución de un proceso y que requiere la ejecución de un fragmento de su código situado fuera del flujo normal de ejecución.

Las excepciones puede tener su origen en una excepción *hardware* síncrona, producida al ejecutar el proceso (véase la sección "1.3 Interrupciones", página 14). También pueden ser generadas directamente por el propio proceso o por el sistema operativo, cuando detectan una situación singular o errónea. En este caso utilizaremos el término de **excepción *software***. Si el proceso contiene un manejador para tratar la excepción generada, el sistema operativo transfiere el control a dicho manejador. En caso contrario aborta la ejecución del proceso.

El manejo de excepciones necesita ser soportado por el lenguaje de programación para que el programador pueda generar excepciones (por ejemplo mediante un *throw*) y pueda definir el o los manejadores que han de tratar la o las distintas excepciones. Un esquema habitual es el que se presenta a continuación:

```
try {
    Bloque donde puede producirse una excepción
}
except {
    Bloque que se ejecutará si se produce una excepción
    en el bloque anterior
}
```

En el esquema anterior, el programador encierra dentro del bloque `try` el fragmento de código que quiere proteger de la generación de excepciones. En el bloque `except` sitúa el manejador de excepciones. En caso de generarse una excepción en el bloque `try`, se transfiere el control al bloque `except`, que se encargará de manejar la correspondiente excepción.

Windows utiliza el mecanismo de excepciones, similar al anterior, para notificar a los procesos los eventos o errores que surgen como consecuencia del programa que están ejecutando. Windows hace uso del concepto de **manejo de excepciones estructurado** que permite a las aplicaciones tomar el control cuando ocurre una determinada excepción. Este mecanismo será tratado en la sección "3.13.7 Servicios Windows para el manejo de excepciones", página 137.

3.6. TEMPORIZADORES

El sistema operativo mantiene en cada BCP uno o varios temporizadores que suelen estar expresados en segundos. Cada vez que la rutina del sistema operativo que trata las interrupciones de reloj comprueba que ha transcurrido un

segundo, decrementa todos los temporizadores que no estén a «0» y comprueba si han llegado a «0». Para aquellos procesos cuyo temporizador acaba de llegar a «0», el sistema operativo notifica al proceso que el temporizador ha vencido. En UNIX se genera una señal SIGALRM. En Windows se planifica la ejecución una función definida por el usuario y que se asocia al temporizador.

El proceso activa el temporizador mediante un servicio en el que especifica el número de segundos o milisegundos que quiere temporizar. Cuando vence la temporización recibirá la correspondiente señal o se ejecutará la función asociada al mismo.

En UNIX el proceso hijo no hereda los temporizadores del padre. Tampoco se conservan los temporizadores después del `exec`.

3.7. PROCESOS ESPECIALES

En esta sección analizaremos los procesos servidores, los demonios y los procesos de núcleo.

3.7.1. Proceso servidor

Un servidor es un proceso que está pendiente de recibir órdenes de trabajo que provienen de otros procesos, que se denominan clientes. Una vez recibida la orden, la ejecuta y responde al peticionario con el resultado de la orden. La figura 3.18 muestra cómo el proceso servidor atiende a los procesos clientes.

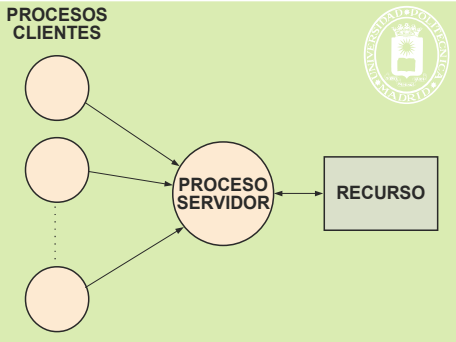


Figura 3.18: *Proceso servidor.*

El proceso servidor tiene la siguiente estructura de bucle infinito:

- Recepción de orden. El proceso está bloqueado esperando a que llegue una orden.
- Recibida la orden, el servidor la ejecuta.
- Finalizada la ejecución, el servidor responde con el resultado al proceso cliente y vuelve al punto de recepción de orden.

Una forma muy difundida de realizar la comunicación entre el proceso cliente y el servidor es mediante puertos. El proceso servidor tiene abierto un puerto, del que lee las peticiones. En la solicitud, el proceso cliente envía el identificador del puerto en el que el servidor debe contestar.

Un servidor será **secuencial** cuando siga estrictamente el esquema anterior. Esto implica que hasta que no ha terminado el trabajo de una solicitud no admite otra. En muchos casos interesa que el servidor sea **paralelo**, es decir, que admita varias peticiones y las atienda simultáneamente. Para conseguir este paralelismo se puede proceder de la siguiente manera, según muestra la figura 3.19:

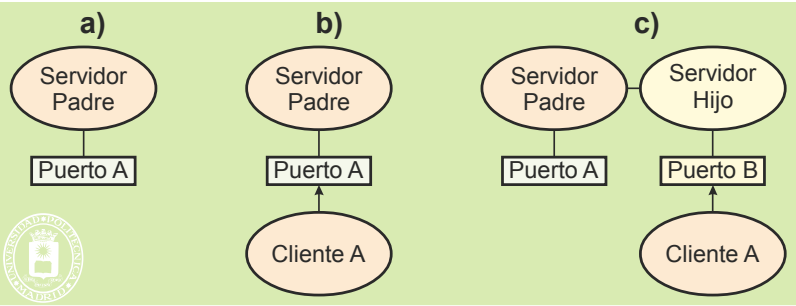


Figura 3.19 *Funcionamiento de un proceso servidor.*

- Lectura de la orden. El proceso está bloqueado esperando a que llegue una orden.
- Asignación un nuevo puerto para el nuevo cliente.
- Generación de un proceso hijo que realiza el trabajo solicitado por el cliente.
- Vuelta al punto de lectura de orden.

De esta forma, el proceso servidor dedica muy poco tiempo a cada cliente, puesto que el trabajo lo realiza un nuevo proceso, y puede atender rápidamente nuevas peticiones.

La figura 3.26, página 92, presenta arquitecturas *multithread* que se suelen utilizar para diseñar servidores paralelos, mientras que la figura 3.20 muestra cómo los procesos cliente y servidor pueden estar en máquinas distintas.

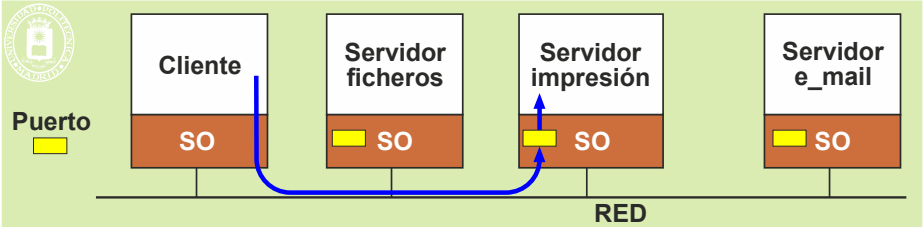


Figura 3.20 *Procesos cliente y servidor en máquinas distintas.*

3.7.2. Demonio

Un proceso demonio es un proceso que suele tener las siguientes características:

- Se arranca al iniciar el sistema, puesto que debe estar siempre activo.
- No muere. En caso de que un demonio muera por algún imprevisto es muy común que exista un mecanismo que detecte la muerte y lo rearranque.
- En muchos casos están a la espera de un evento. En el caso frecuente de que el demonio sea un servidor, el evento por el que espera es que llegue una petición al correspondiente puerto.
- En otros casos, el demonio tiene encomendada una labor que hay que hacer de forma periódica, ya sea cada cierto tiempo o cada vez que una variable alcanza un determinado valor.
- Es muy frecuente que no haga directamente el trabajo: lanza otros procesos para que lo realicen.
- Los procesos servidores suelen tener el carácter de demonios.
- Los demonios ejecutan en *background* y no están asociados a un terminal o proceso *login*.

Como ejemplos de procesos servidores UNIX se pueden citar los siguientes:

- lpd line printer daemon.
- inetd arranca servidores de red: ftp, telnet, http, finger, talk, etc.
- smbd demonio de samba.
- atd ejecución de tareas a ciertas horas.
- crond ejecución de tareas periódicas.
- nfsd servidor NFS.
- httpd servidor WEB.

3.7.3. Proceso de usuario y proceso de núcleo

Frente al proceso de usuario que se ha descrito en las secciones anteriores, en la mayoría de los sistemas operativos existen otro tipo de procesos que se suelen denominar procesos o *threads* de sistema (el concepto de *threads* se desarrolla seguidamente en la sección "3.8 Threads"). Se trata de procesos que durante toda su vida ejecutan código del sistema operativo en modo privilegiado. Generalmente, se crean en la fase inicial del sistema operativo, aunque también pueden ser creados más adelante por un manejador u otro módulo del sistema operativo. Este tipo de procesos lleva a cabo labores del sistema que se realizan mejor usando una entidad activa, como es el proceso, que bajo el modelo de operación orientado a eventos de carácter pasivo con el que se rige la ejecución del sistema operativo. El uso de un proceso de núcleo permite realizar operaciones, como bloqueos, que no pueden llevarse a cabo en el tratamiento de una interrupción externa.

Habitualmente, estos procesos se dedican a labores vinculadas con la gestión de memoria (los "demonios de paginación" presentes en muchos sistemas operativos), el mantenimiento de la cache del sistema de ficheros, los manejadores de dispositivos complejos y al tratamiento de ciertas llamadas al sistema. En Windows, aunque cualquier módulo del sistema puede crear un proceso de núcleo, éste no es el método más recomendado. Existe un conjunto de procesos (*threads*) "trabajadores" de núcleo, cuya única misión es ejecutar peticiones de otros módulos. Por tanto, un módulo que requiera realizar una operación en el contexto de un proceso de núcleo no necesita crearlo sino que puede encolar su petición para que la ejecute uno de los procesos trabajadores de núcleo.

Hay que resaltar que, aunque en la mayoría de los casos estos procesos de núcleo tienen una prioridad alta, debido a la importancia de las labores que realizan, no siempre es así. Un ejemplo evidente es el proceso nulo, que es un proceso de núcleo pero que, sin embargo, tiene prioridad mínima, para de esta forma sólo conseguir el procesador cuando no hay ningún otro proceso listo en el sistema.

Es importante notar la gran diferencia que hay entre estos procesos de núcleo y los procesos de usuario creados por el superusuario del sistema. Los procesos de superusuario son procesos convencionales: ejecutan en modo usuario el código del ejecutable correspondiente. No pueden, por tanto, utilizar instrucciones máquina privilegiadas. Su "poder" proviene de tener como propietario al superusuario, por lo que no tienen restricciones a la hora de realizar llamadas al sistema aplicadas a cualquier recurso del sistema. Algunos ejemplos de procesos de superusuario son

los que proporcionan servicios de red y *spooling* (los típicos "demonios" de los sistemas UNIX). Nótese que, aunque muchos de estos procesos se crean también en el arranque del sistema, hay una diferencia significativa con los procesos de núcleo: los procesos de superusuario los crea siempre otro proceso, en muchos casos el proceso inicial (en UNIX, `init`), mientras que los procesos de núcleo los crea el propio sistema operativo en su fase inicial. Téngase en cuenta que, en el caso de un sistema operativo con una estructura de tipo microkernel, los procesos que proporcionan las funcionalidades básicas del sistema, como, por ejemplo, la gestión de ficheros, no son procesos de núcleo, sino que tienen las mismas características que los procesos de superusuario.

Las principales características de los procesos de núcleo son las siguientes:

- Se crean mediante un servicio especial distinto del utilizado para los procesos de usuario.
- Se crean dentro del dominio de protección del sistema operativo.
- Tienen acceso a todo el mapa de memoria del sistema operativo.
- Su imagen de memoria se encuentra dentro del mapa de memoria del sistema operativo.
- Pueden utilizar un conjunto restringido de llamadas al sistema, además de los servicios generales que utilizan los procesos de usuario.
- Ejecutan siempre en modo privilegiado.
- Son flujos de ejecución independientes dentro del sistema operativo.

El diseño de los procesos de núcleo ha de ser muy cuidadoso puesto que trabajan en el dominio de protección del sistema operativo y tienen todos los privilegios. Un proceso de núcleo, antes de terminar, ha de liberar toda la memoria dinámica y todos los cerrojos asignados, puesto que no se liberan automáticamente.

3.8. THREADS

Un proceso se puede considerar formado por un activo y por un flujo de ejecución. El activo es el conjunto de todos los bienes y derechos que tiene asignados el proceso. En el activo se incluye la imagen de memoria del proceso, el BCP, los ficheros abiertos, etc. El flujo de ejecución está formado por el conjunto ordenado de instrucciones máquina del programa que va ejecutando el proceso. Denominaremos *thread* a esta secuencia de ejecución (otros nombres utilizados son hilo o proceso ligero). Íntimamente ligado al *thread* está el estado del procesador, que va cambiando según avanza el *thread*, el estado de ejecución (listo, bloqueado o en ejecución) y la pila.

En los procesos clásicos, que hemos estudiado hasta ahora, solamente existe un único *thread* de ejecución, por lo que les denominamos ***monothread***. La extensión que se plantea en esta sección consiste en dotar al proceso de varios *threads*. Llamaremos proceso ***multithread*** a este tipo de proceso, mostrado la figura 3.21. El objetivo de disponer de varios *threads* es conseguir concurrencia dentro del proceso, al poder ejecutar los mismos simultáneamente.

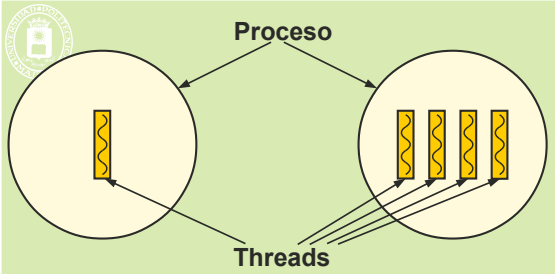


Figura 3.21: *Proceso monothread y multithread.*

Cada *thread* tiene informaciones que le son propias, informaciones que se refieren fundamentalmente al contexto de ejecución, pudiéndose destacar las siguientes:

- Estado del procesador: Contador de programa y demás registros visibles.
- Pila.
- Estado del *thread* (ejecutando, listo o bloqueado).
- Prioridad de ejecución.
- Bloque de control de *thread*.

Todos los *threads* de un mismo proceso comparten el activo del mismo, en concreto comparten:

- Espacio de memoria.
- Variables globales.
- Ficheros abiertos.
- Procesos hijos.
- Temporizadores.
- Señales y semáforos.
- Contabilidad.

Es importante destacar que todos los *threads* de un mismo proceso comparten el mismo espacio de direcciones de memoria, que incluye el código, los datos y las pilas de los diferentes *threads*. Esto hace que no exista protección

de memoria entre los *threads* de un mismo proceso y que un *thread* pueda, aunque no sea recomendable, acceder a la información propia de otro *thread*, como puede ser su pila.

La figura 3.22 muestra cómo cada *thread* requiere una estructura **BCT** (Bloque de Control de *Thread*) que contiene la identificación del *thread*, el estado de los registros e información de control, entre la que se encuentra el estado del *thread*, información de planificación y la máscara de señales.

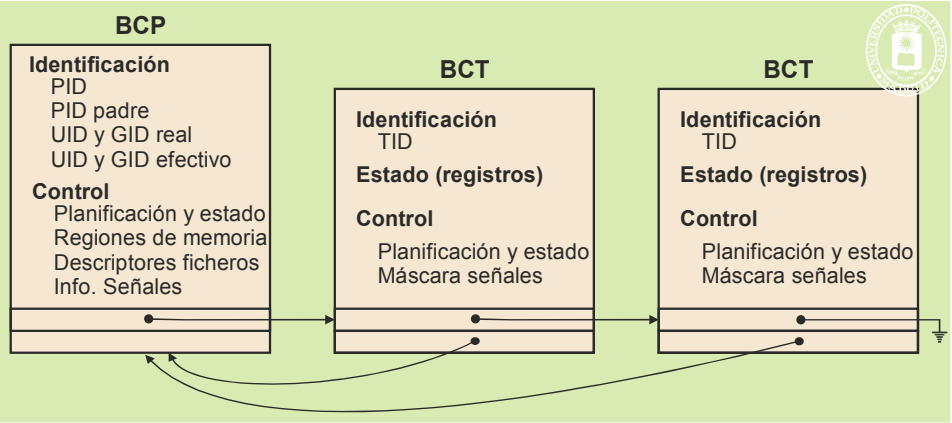


Figura 3.22 *Un proceso multithread incluye una estructura BCT por cada thread.*

El *thread* constituye la unidad ejecutable en Windows. La figura 3.23 representa de forma esquemática la estructura de un proceso de Windows con sus *threads*.

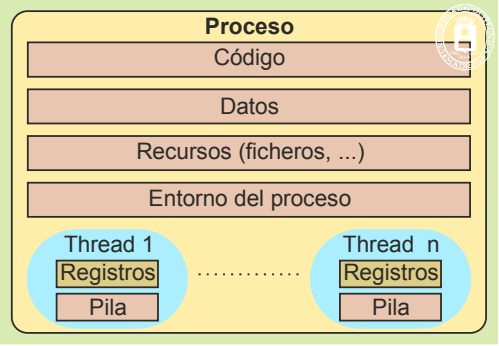


Figura 3.23 *Estructura de un proceso en Windows.*

3.8.1. Gestión de los *threads*

Hay dos formas básicas de gestionar los *threads*: la ULT (*User Level threads*) y la KLT (*Kernel Level threads*), que se diferencian por el conocimiento que tiene el sistema operativo de los mismos.

- **ULT**. Los ***threads* de nivel de usuario** o ***threads* de biblioteca** los gestiona totalmente una biblioteca de *threads*, que ha de incluirse en el proceso como una capa *software* sobre la que ejecutan los *threads*. El sistema operativo desconoce la existencia de estos *threads*, para él se trata de un proceso normal de un solo *thread*. Es la capa de *software* la que crea, planifica, activa y termina los distintos *threads*.
- **KLT**. Los ***threads* de nivel de núcleo** o ***threads* de sistema** los gestiona el sistema operativo. Es el núcleo del sistema operativo el que crea, planifica, activa y termina los distintos *threads*.

Para el programador puede ser muy diferente que la gestión la realice o no el sistema operativo, puesto que puede afectar profundamente a las prestaciones de la aplicación. Es muy importante conocer bien las diferencias de comportamiento que existen entre ambas alternativas, para poder seleccionar la más adecuada a cada aplicación (aclaración 3.6). En las siguientes secciones se irán analizando distintos aspectos de los *threads*, destacando las diferencias entre las dos soluciones de *threads* de biblioteca y de sistema.

Aclaración 3.6. Las bibliotecas actuales de *threads* están basadas en servicios asíncronos, por lo que permiten que los *threads* se bloqueen sin bloquear al proceso. Sin embargo, en el texto se utilizará el modelo básico de biblioteca de *threads* que no tiene esta funcionalidad.

Señales en *threads* de sistema

En algunos sistemas operativos con *threads* de sistema, el tratamiento de las señales no se realiza a nivel de proceso sino a nivel de *thread*, de acuerdo a las siguientes reglas:

- El sistema operativo mantiene una máscara de señales por *thread* de sistema.
- La acción asociada a una señal es compartida por todos los *threads*.
- Las señales síncronas producidas por la ocurrencia de una excepción *hardware* síncrona son enviadas al *thread* que causó la excepción.

- Si la acción de la señal es terminar, se termina el proceso con todos sus *threads*.

3.8.2. Creación, ejecución y terminación de *threads*

Desde el punto de vista de la programación, un *thread* se define como la ejecución de una función en paralelo con otras. El *thread* primario corresponde a la función `main`. Cuando se pone en ejecución un programa se activa únicamente el *thread* `main`. El resto de los *threads* son creados por el sistema operativo (caso de *threads* de sistema) o por la capa de gestión de *threads* (caso de *threads* de biblioteca) bajo petición del proceso mediante el correspondiente servicio de creación de *thread*. Por tanto, el `main` puede lanzar *threads*, que, a su vez, pueden lanzar otros *threads*.

La terminación de un *thread* se realiza cuando el *thread* ejecuta el retorno de la función asociada o cuando llama al servicio terminar el *thread*.

Cuando termina el último *thread* del proceso termina el proceso. Además, cuando termina el *thread* primario termina el proceso, aunque queden otros *threads* vivos.

3.8.3. Estados de un *thread*

El *thread* puede estar en uno de los tres estados siguientes: ejecución, listo o bloqueado. Como muestra la figura 3.24, cada *thread* de un proceso tiene su propio estado.

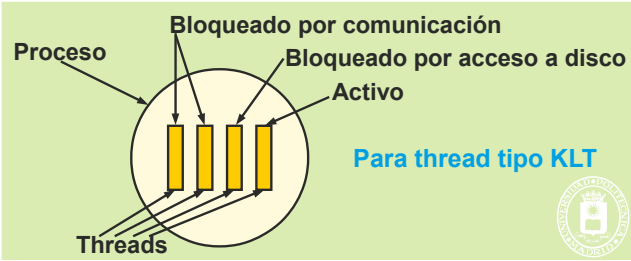


Figura 3.24 *Estados de un thread.*

El estado del proceso será la combinación de los estados de sus *threads*. La tabla 3.1 muestra la relación entre el estado del proceso y los de sus *threads*. Puede observarse la diferencia entre los *threads* de biblioteca y los de sistema. En el caso de sistema puede existir más de un *thread* en ejecución en el mismo instante (siempre que la máquina sea multiprocesadora), además, puede haber varios *threads* bloqueados al mismo tiempo, hecho que no ocurre en los *threads* de biblioteca.

Tabla 3.1 *Relación entre el estado del proceso y los estados de sus threads. [1-n] indica uno o más, mientras que [0-n] indica cero o más.*

Estado del proceso	**Estados de los *threads* de biblioteca**			**Estados de los *threads* de sistema**		
	Ejecución	**Bloqueado**	**Listo**	**Ejecución**	**Bloqueado**	**Listo**
Ejecución	1	0	[0-n]	[1-n]	[0-n]	[0-n]
Bloqueado	0	1	[0-n]	0	[1-n]	0
Listo	0	0	[n]	0	[0-n]	[1-n]

La planificación de los *threads* de sistema la realiza el planificador del sistema operativo, por lo que es externa al proceso, mientras que en los *threads* de biblioteca la realiza la capa de *software* de *threads*, por lo que es interna al proceso.

3.8.4. Paralelismo con *threads*

Los *threads* de sistema, al estar gestionados por el núcleo, permiten paralelizar una aplicación, tal y como muestra la figura 3.25. En efecto, cuando un programa puede dividirse en procedimientos que pueden ejecutar de forma concurrente, el mecanismo de los *threads* de sistema permite lanzar, simultáneamente, la ejecución de todos ellos. De esta forma se consigue que el proceso avance más rápidamente (véase prestaciones 90). La base de este paralelismo estriba en que, mientras un *thread* está bloqueado, otro puede estar ejecutando. En máquinas multiprocesadores varios *threads* pueden estar ejecutando simultáneamente sobre varios procesadores.

Prestaciones 3.1. Los *threads* permiten que un proceso aproveche más el procesador, es decir, ejecute más deprisa. Sin embargo, esto no significa que el proceso aumente su tasa total de uso del procesador.

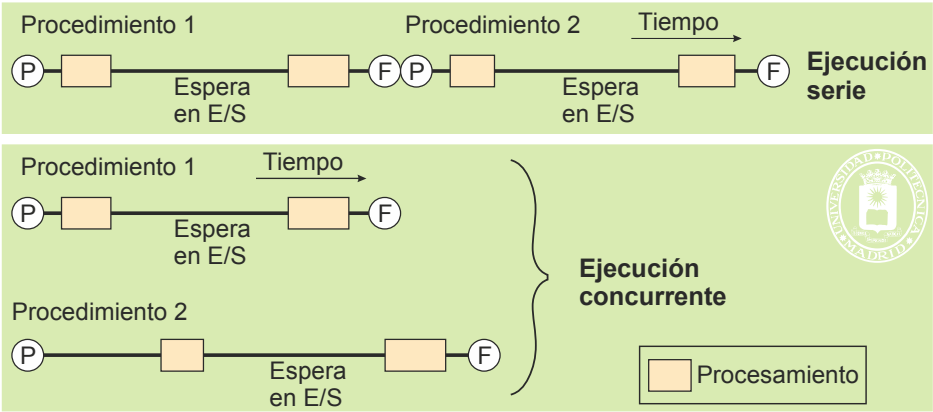


Figura 3.25 *Los threads permiten paralelizar la ejecución de una aplicación.*

Comparando, desde el punto de vista del paralelismo, el uso de los *threads* de sistema con otras soluciones se puede decir que:

- Los *threads* de sistema:
  - Permiten paralelismo y variables compartidas.
  - Utilizan llamadas al sistema bloqueantes, que solamente bloquean al *thread* que pide el servicio. Esta es la forma más sencilla de conseguir paralelismo desde el punto de vista de la programación.
- Un proceso convencional con un solo *thread* o *threads* de biblioteca:
  - No hay paralelismo.
  - Utiliza llamadas al sistema bloqueantes.
  - Los *threads* de biblioteca permiten compartir variables.
- Un proceso con un *thread* o *threads* de biblioteca, que usa llamadas no bloqueantes y estructurado en máquina de estados finitos:
  - Permite paralelismo entre procesador y E/S pero no de varios procesadores, y variables compartidas.
  - Utiliza llamadas al sistema no bloqueantes, lo que lleva a un diseño muy complejo y difícil de mantener.
- Varios procesos convencionales cooperando:
  - Permite paralelismo.
  - No comparte variables, por lo que la comunicación puede consumir mucho tiempo.

3.8.5. Diseño con *threads*

La utilización de *threads* ofrece las ventajas de división de trabajo que dan los procesos, pero con una mayor flexibilidad y ligereza, lo que se traduce en mejores prestaciones. En este sentido, es de destacar que los *threads* comparten memoria directamente, por lo que no hay que añadir ningún mecanismo adicional para utilizarla, y que la creación y destrucción de *threads* requiere menos trabajo que la de procesos.

Las ventajas de diseño que se pueden atribuir a los *threads* son las siguientes:

- Permite separación de tareas. Cada tarea se puede encapsular en un *thread* independiente.
- Facilita la modularidad, al dividir trabajos complejos en tareas.
- Los *threads* de sistema permiten aumentar la velocidad de ejecución del trabajo, puesto que permiten aprovechar los tiempos de bloqueo de unos *threads* para ejecutar otros (las bibliotecas actuales de *threads* permiten bloquear los *threads* sin bloquear el proceso).

El paralelismo que permiten los *threads* de sistema, unido a que comparten memoria (utilizan variables globales que ven todos ellos), permite la programación concurrente. Este tipo de programación tiene cierta dificultad, puesto que hay que garantizar que el acceso a los datos compartidos se haga de forma correcta. Los principios básicos que hay que aplicar son los siguientes:

- Hay variables globales que se comparten entre los *threads*. Dado que cada *thread* ejecuta de forma independiente a los demás, es fácil que ocurran accesos incorrectos a estas variables.
- Para ordenar la forma en que los *threads* acceden a los datos se emplean mecanismos de sincronización, como el *mutex*, que se describirán en el capítulo “6 Comunicación y sincronización de procesos”. El objetivo de estos mecanismos es garantizar el acceso coordinado a la información compartida.

Ventajas e inconvenientes de *threads* de biblioteca y de sistema

Analizaremos una serie de operaciones y propiedades para comparar entre sí los *threads* de biblioteca y los de sistema.

- Tiempo de procesador. El tiempo de procesador consumido por las operaciones relativas a los *threads* tales como creación, destrucción, sincronización y planificación de los *threads* es menor en el caso de los *threads* de biblioteca que en los de sistema. Para muchas de estas operaciones los tiempos de procesador relativos son del orden de 1 para *threads* de biblioteca, 10 para *threads* de sistema y 100 para los procesos convencionales. Esto es debido a que en el de *threads* de biblioteca no entra a ejecutar el núcleo, toda la gestión la realiza la capa de *software* de *threads* incluida en el propio proceso.
- Los *threads* de biblioteca se pueden utilizar en cualquier sistema operativo, basta con disponer de la capa de *software* de *threads*. Los *threads* de sistema sólo se pueden ejecutar en sistemas que los soporten.
- Los *threads* de biblioteca permiten utilizar una planificación específica, dentro del tiempo asignado por el sistema operativo. Planificación que es distinta de la del sistema operativo. En los *threads* de sistema se utiliza la planificación del sistema operativo.
- Los *threads* de sistema permiten paralelismo con varios *threads* bloqueados y *threads* en ejecución. Por el contrario, los *threads* de biblioteca no permiten este paralelismo.

Dependiendo de la aplicación será más interesante el uso de *threads* de sistema o de biblioteca. Es, por tanto, necesario hacer un estudio detallado de las ventajas que cada una de estas soluciones tiene para cada aplicación antes de seleccionar la solución a emplear.

Arquitecturas *software* basadas en *threads*

La figura 3.26 muestra tres arquitecturas *software* basadas en *threads*. En todas ellas, las órdenes a la aplicación se pueden recibir mediante algún sistema de E/S, como puede ser el ratón y el teclado, en el caso de aplicaciones interactivas, o mediante un puerto de comunicaciones como es el caso de los procesos servidores. Es de destacar que las técnicas *multithread* con *threads* de sistema están muy indicadas en el diseño de procesos servidores por el paralelismo y aprovechamiento de los procesadores que permite.

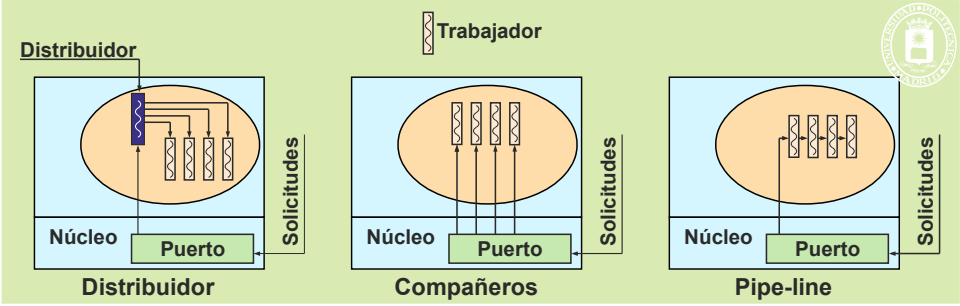


Figura 3.26 *Arquitecturas software basadas en threads.*

En la primera arquitectura se plantea un *thread* distribuidor cuya función es la recepción de las órdenes y su traspaso a un *thread* trabajador. El esquema puede funcionar creando un nuevo *thread* trabajador por cada solicitud de servicio, *thread* que muere al finalizar su trabajo, o teniendo un conjunto de ellos creados a los que se va asignando trabajo y que quedan libres al terminar la tarea encomendada. Esta segunda alternativa es más eficiente, puesto que se evita el trabajo de crear y destruir los *threads*, pero es más compleja de programar.

La segunda arquitectura consiste en disponer de un conjunto de *threads* iguales, todos los cuales pueden aceptar una orden. Cuando llega una solicitud, la recibe uno de los *threads* que están leyendo del correspondiente elemento de E/S. Este *thread* tratará la petición y, una vez finalizado el trabajo solicitado, volverá a la fase de leer una nueva petición.

La tercera arquitectura aplica la técnica denominada de segmentación (*pipe-line*). Cada trabajo se divide en una serie de fases, encargándose un *thread* especializado de cada una de ellas. El esquema permite tratar al mismo tiempo tantas solicitudes como fases tenga la segmentación, puesto que se puede tener una en cada *thread*.

3.9. ASPECTOS DE DISEÑO DEL SISTEMA OPERATIVO

En esta sección analizaremos algunos conceptos sobre el modo en que ejecuta el sistema operativo, planteando los siguientes puntos:

- Núcleo con ejecución independiente.
- Núcleo que ejecuta dentro de los procesos de usuario.
- Interrupciones y expulsión.
- Tablas del núcleo.

3.9.1. Núcleo con ejecución independiente

Esta es la alternativa de diseño que podemos llamar clásica, puesto que se ha utilizado en sistemas operativos antiguos y se sigue utilizando en sistemas operativos sencillos. Como muestra la figura 3.27, el sistema operativo tiene su propia región de memoria y su propia pila de sistema, en la que se anidan los bloques de activación de las funcio-

nes del sistema operativo. Decimos que el sistema operativo ejecuta fuera de todo proceso, puesto que tiene su propio contexto independiente de los contextos de los procesos.

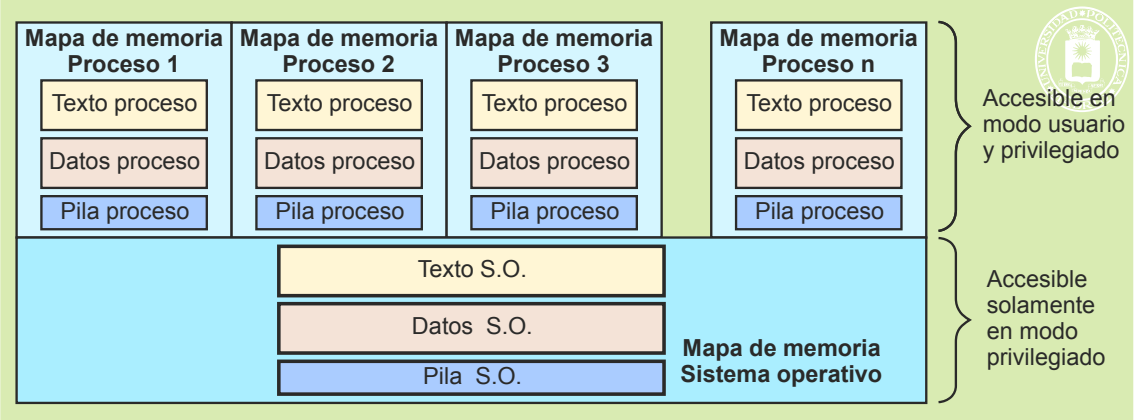


Figura 3.27 *Esquema de ejecución de un sistema operativo con ejecución independiente.*

Cuando un proceso es interrumpido o solicita un servicio, el sistema operativo salva el contexto del proceso, carga su propio contexto y ejecuta todas las funciones necesarias para tratar la interrupción. Al finalizar éstas, el sistema operativo restaura el contexto del proceso interrumpido o de otro proceso, y lo pone en ejecución.

El cambio de contexto implica dos acciones: cambiar el estado del procesador y cambiar el mapa de memoria. Dependiendo del tipo de computador, esta segunda operación puede ser bastante más costosa que la primera puesto que puede suponer, por ejemplo, tener que anular el contenido de la TLB (ver capítulo "4 Gestión de memoria").

Observe que, en este caso, el concepto de proceso se aplica únicamente a los programas de usuario. El sistema operativo es una entidad independiente que ejecuta en modo privilegiado.

El esquema de estados de los procesos es el de la 3.12, página 81, con los estados de Ejecución, Listo y Bloqueado.

Cambios de contexto

Como se ha visto, el sistema operativo ejecuta en su propio contexto, por lo que el paso de ejecutar el proceso a ejecutar el sistema operativo supone un cambio de contexto, a la vez que supone que el procesador pasa a ejecutar en modo privilegiado. Una vez que el sistema operativo ha terminado su trabajo, pondrá en ejecución al mismo o a otro proceso, por lo que se pasará al contexto de dicho proceso, lo que supone un nuevo cambio de contexto y que el procesador pase a ejecutar en modo usuario.

3.9.2. Núcleo con ejecución dentro de los procesos de usuario

Una solución alternativa, muy empleada en los sistemas operativos de propósito general actuales, es ejecutar prácticamente todo el *software* del sistema operativo en el contexto de los procesos de usuario. El sistema operativo se visualiza como una serie de funciones que se ejecutan en el contexto del proceso, pero habiendo pasado el proceso a ejecutar en modo privilegiado. Esto implica, cuando la interrupción a tratar no tiene nada que ver con el proceso en curso, que, dentro de su contexto, se están realizando funciones para otro proceso.

El sistema operativo mantiene, por tanto, un flujo de ejecución con su propia pila por cada proceso. En este caso no hay cambio de contexto cuando un proceso es interrumpido, simplemente se ejecutan las funciones del sistema operativo en el contexto del proceso, pero utilizando la pila que el sistema operativo mantiene para cada proceso, pila que está en memoria protegida. Sin embargo, sí hay un cambio de modo, puesto que se pasa de estar ejecutando en modo usuario a modo privilegiado, por lo que el espacio de memoria del sistema operativo es accesible.

El mapa de memoria del proceso, en este caso, incorpora una pila propia del sistema operativo más las regiones de datos y texto del sistema operativo, regiones que comparte con el resto de los procesos, como se puede observar en la figura 3.28.

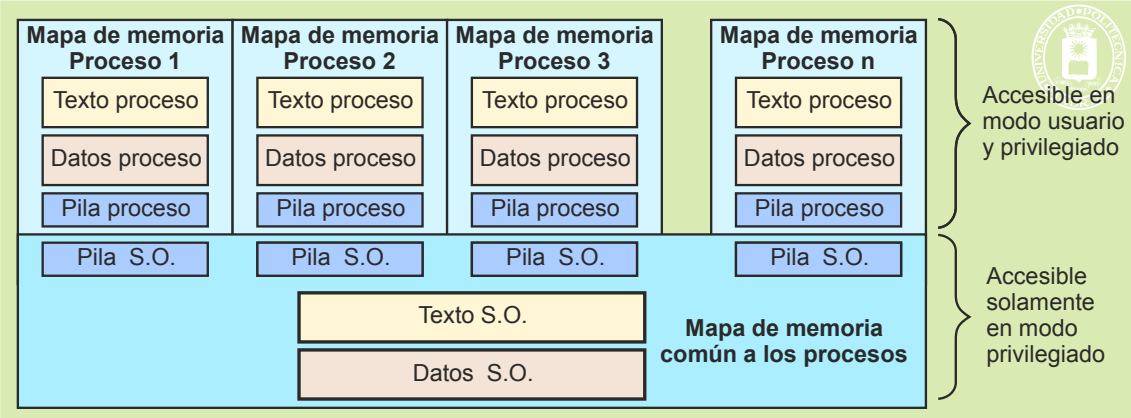


Figura 3.28 *Esquema de ejecución de un sistema operativo con ejecución en los procesos de usuario.*

Una vez que el sistema operativo ha realizado su trabajo, puede seguir ejecutando el mismo proceso, para lo cual simplemente pasa a modo usuario, o puede seleccionar otro proceso para ejecutar, lo que conlleva un cambio de contexto del proceso interrumpido al nuevo proceso.

Aunque el sistema operativo esté ejecutando dentro del contexto del proceso, no se viola la seguridad, puesto que el paso al sistema operativo se realiza por una interrupción, por lo que ejecuta en modo privilegiado, pero cuando vuelve a dar control al código de usuario del proceso pone el computador en modo usuario.

Diagrama de estados

Este tipo de sistema operativo presenta los cuatro estados de proceso representados en la figura 3.29. Dicha figura también presenta algunas de las posibles transiciones entre los estados.

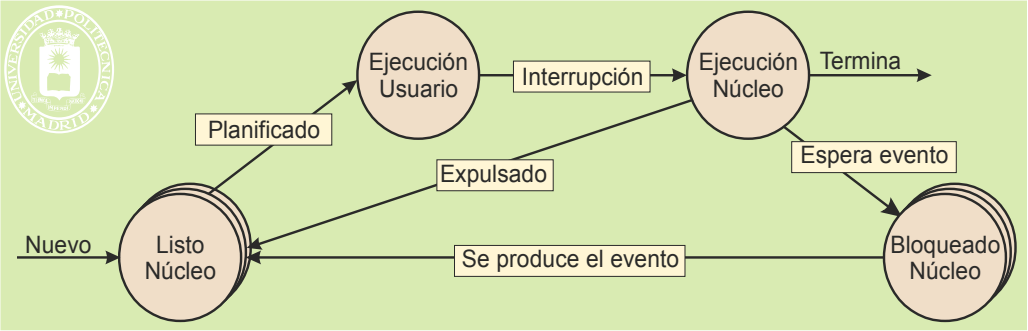


Figura 3.29 *Estados básicos del proceso en el caso de ejecución del núcleo dentro del proceso de usuario.*

Las diferencias con el diagrama de estados anterior son dos. Aparece el estado de ejecución en modo privilegiado y los estados del proceso son siempre en modo privilegiado menos para el estado de ejecución en modo usuario. Veamos con más detalle estos estados:

- **Ejecución usuario**. El proceso está ejecutando su propio código y el computador está en modo usuario.
- **Ejecución núcleo**. El proceso PA estaba en el estado de ejecución usuario y llega una interrupción (que puede ser excepción *hardware*, una interrupción externa o una instrucción TRAP). El sistema operativo entra a ejecutar dentro del contexto de PA pero con el procesador en modo privilegiado. No se produce cambio de contexto, pero se produce un cambio de modo, pasando a modo privilegiado. Observe que la interrupción a tratar puede provenir del propio proceso PA o puede ser ajena a él, pero en ambos casos el sistema operativo ejecuta en el contexto del proceso PA.
- **Bloqueado núcleo**. El proceso queda bloqueado igual que en el caso anterior, pero sigue en modo privilegiado.
- **Listo núcleo**. Cuando se produce el evento por el que se espera, el proceso pasa a listo, pero sigue en modo privilegiado. Cuando el proceso listo es planificado se pasa a modo de ejecución usuario.

Observemos que el proceso se inicia en modo privilegiado y termina en modo privilegiado.

Cambio de contexto

Cuando el núcleo ejecuta dentro del proceso de usuario, el paso del código de usuario al código del sistema operativo, no conlleva cambio de contexto, aunque sí conlleva un cambio de estado, por lo que no es necesario guardar el estado del procesador en el BCP. El sistema operativo ejecuta en el contexto del proceso PA interrumpido.

Finalizado el tratamiento de la o las interrupciones, si como resultado de alguna de ellas se activó el planificador, se ejecutará éste, que seleccionará el próximo proceso a ejecutar. Se produce, por tanto, un cambio de contexto.

En un cambio de contexto el proceso en ejecución puede transitar al estado de listo o al de bloqueado. Es importante distinguir entre estos dos tipos de cambio de contexto:

- **Cambio de contexto voluntario**: Se produce cuando el proceso en ejecución pasa al estado de bloqueado debido a que tiene que esperar por algún tipo de evento. Sólo pueden ocurrir dentro de una llamada al sistema o en el tratamiento de un fallo de página. No pueden darse nunca en una rutina de interrupción asíncrona, ya que ésta no está relacionada con el proceso que está actualmente ejecutando. Observe que el proceso deja el procesador puesto que no puede continuar ejecutando. Por tanto, el motivo de este cambio de contexto es un uso eficiente del procesador.
- **Cambio de contexto involuntario**: Se produce cuando el proceso en ejecución tiene que pasar al estado de listo ya que debe dejar el procesador por algún motivo (por ejemplo, debido a que se le ha acabado su rodaja o porción de tiempo o porque hay un proceso más urgente listo para ejecutar). Nótese que, en este caso, el proceso podría seguir ejecutando, pero se realiza el cambio de contexto para realizar un mejor reparto del tiempo del procesador.

El cambio de contexto es una operación relativamente costosa (exige la ejecución de varios miles de instrucciones máquina, puesto que hay que cambiar de tablas de páginas, hay que renovar la TLB, etc.) y no es realmente trabajo útil, sobre todo en el caso de que el proceso en ejecución pase al estado de listo (cambio de contexto involuntario), ya que en esa situación se ha incurrido en la sobrecarga del cambio de contexto, cuando se podría haber continuado con el proceso en ejecución. Por tanto, hay que intentar mantener la frecuencia de cambios de contexto dentro de unos límites razonables.

Cambio de modo

En el esquema de estados de la figura 3.29 se puede observar que se produce un cambio de modo en dos ocasiones: cuando el proceso en modo usuario es interrumpido, y cuando un proceso es activado (justo en las transiciones en las que, un núcleo de ejecución independiente, presenta cambio de contexto). El cambio de modo implica salvar el estado visible del procesador (lo que puede hacerse en la pila que tiene el sistema operativo para ese proceso), pero no implica salvar el estado no visible ni el mapa de memoria, por lo que es menos costoso que el cambio de contexto.

Otros estados

Cada sistema operativo define su modelo de estados de los procesos. Dichos modelos suelen ser más complejos que los esquemas generales presentados anteriormente. Por ejemplo, en UNIX encontramos que:

- Existe un estado denominado **zombi**, que se corresponde con un proceso que ha terminado su ejecución pero cuyo proceso padre todavía no ha ejecutado la llamada `wait`.
- Aparece el estado ***stopped***, al que un proceso transita cuando recibe la señal SIGSTOP (u otras señales que tienen ese mismo efecto) o cuando, estando bajo la supervisión de un depurador, recibe cualquier señal.
- El estado de bloqueo está dividido en dos estados independientes:
  - **Espera interrumpible**: además de por la ocurrencia del suceso que está esperando, un proceso puede salir de este estado por una señal dirigida al mismo. Considere, por ejemplo, un proceso que está bloqueado esperando porque le llegue información por la red o del terminal. Es impredecible cuándo puede llegar esta información y debe de ser posible hacer que este proceso salga de este estado para, por ejemplo, abortar su ejecución.
  - **Espera no interrumpible**: en este caso, sólo la ocurrencia del suceso puede hacer que este proceso salga de este estado. Debe de tratarse de un suceso que es seguro que ocurrirá y, además, en un tiempo relativamente acotado, como, por ejemplo una interrupción del disco.
- Normalmente, el campo del BCP que guarda el estado no distingue explícitamente entre el estado listo y en ejecución. Esa diferencia se deduce a partir de otras informaciones adicionales.

Si se trata de un sistema con *threads*, el diagrama de estados se aplicará a cada *thread*. Este es el caso de Windows, donde cada *thread* transita por un diagrama como el de la figura 3.29, junto con dos estados específicos:

- ***standby***: el proceso ha sido seleccionado como el próximo que ejecutará en un determinado procesador, pero debe esperar hasta que se cumplan las condiciones necesarias.
- ***transition***: el *thread* está listo para ejecutar pero su pila de sistema no está residente.

3.10. TRATAMIENTO DE INTERRUPCIONES

Primero se estudia la problemática que surge al ocurrir interrupciones en las diversas situaciones en las que se puede encontrar el sistema, introduciendo el concepto de sistema operativo expulsable. Seguidamente, se estudia el tratamiento de interrupciones justamente en el caso de un sistema operativo de tipo no expulsable y con ejecución dentro de los procesos de usuario.

3.10.1. Interrupciones y expulsión

De forma global podemos decir que en un sistema existen las tres categorías de código siguientes:

- Rutinas de tratamiento de interrupción.
- Servicios del sistema operativo (según lo visto, un servicio puede necesitar varias fases).
- Código de usuario.

En términos generales, las rutinas de tratamiento de interrupción tienen prioridad sobre el código de los servicios y éstos, a su vez, tienen prioridad sobre el código de usuario. Esto significa que, mientras esté pendiente una rutina de tratamiento de interrupción, no deberá ejecutar código de servicios y que, mientras se tenga pendiente código de algún servicio, no deberá ejecutarse código de usuario.

Por ello, cuando se produce una interrupción en medio de código de usuario o de servicio, se deja de ejecutar dicho código y se pasa a ejecutar la correspondiente rutina de tratamiento de interrupción. Con gran frecuencia la interrupción implica la activación de una fase de un servicio, por lo que se ejecutará dicho código antes de pasar a ejecutar el código de usuario.

Dado que una interrupción puede producirse en cualquier instante, puede ocurrir cuando se está ejecutando código de cualquier categoría. Analizaremos seguidamente cada uno de los posibles casos.

Interrupción durante rutina de tratamiento de interrupción. Anidamiento de interrupciones

La interrupción puede estar inhibida o enmascarada. En este caso el *hardware* está reteniendo la interrupción, por lo que no es tratada. Si se mantiene por un tiempo esta situación, puede llegar a perderse la interrupción, o bien porque se retire o bien porque se superponga otra interrupción del mismo tipo, lo que en general es inadmisible.

Para evitar este problema se diseñan las rutinas de tratamiento de interrupción de forma que puedan ser interrumpidas por otra interrupción (generalmente de nivel más prioritario). La consecuencia de esta alternativa de diseño es el **anidamiento de las interrupciones**. La figura 3.30 muestra esta situación: se está ejecutando un proceso y llega la interrupción 1, por lo que se empieza con la ejecución de la rutina de tratamiento 1. Antes de completarse esta rutina llega la interrupción 2, por lo que empieza a ejecutarse la rutina 2, pero llega la interrupción 3 antes de finalizar, por lo que se pasa a la rutina 3. Podemos decir que unas rutinas expulsan a otras del procesador.

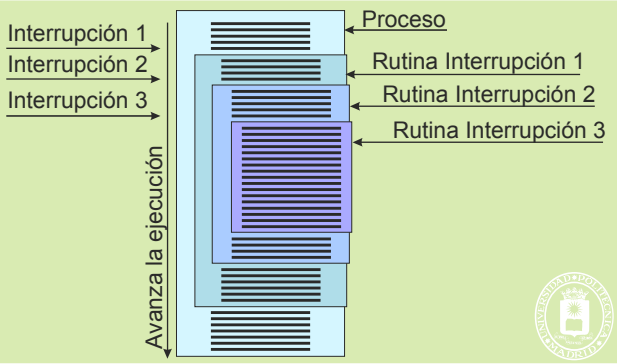


Figura 3.30 *Anidamiento de interrupciones.*

En la figura 3.30 las rutinas de tratamiento se terminan de ejecutar en orden LIFO, lo que es bastante frecuente, sobre todo si las rutinas de tratamiento sólo permiten interrupciones de niveles más prioritarios, pero no debe descartarse el que se completen en otro orden. Finalizado el tratamiento de todas las rutinas de interrupción se pasaría a ejecutar el código de servicio (si lo hay), el planificador (si se ha solicitado en alguna rutina de interrupción) y el código interrumpido u otro código de usuario.

Para reducir al máximo el anidamiento y minimizar el tiempo que se tarda en atender una interrupción la rutina de tratamiento se divide en dos partes: la que llamaremos **rutina no aplazable** incluye el tratamiento que es necesario realizar urgentemente y debe ser lo más breve posible. Y otras que se aplaza, tal y como se detalla en la sección “3.10.2 Detalle del tratamiento de interrupciones”, página 98.

Un aspecto muy importante en el diseño del tratamiento de las interrupciones es que la rutina no aplazable no puede quedarse "a medias" durante un tiempo indefinido, ya que el dispositivo periférico podría quedarse en estado incoherente y podrían perderse datos. No hay que olvidar que los dispositivos tienen su vida propia y que necesitan ser atendidos dentro de determinadas ventanas de tiempo. Entre los mecanismos que no puede utilizar una rutina de interrupción no aplazable destacaremos los siguientes:

- No puede realizar un cambio de contexto.
- No puede usar un semáforo o *mutex* para sincronizarse.
- No puede generar un fallo de página.
- No puede acceder al espacio de memoria del usuario, puesto que podría generar un fallo de página y porque no sabe en qué proceso está ejecutando.

Interrupción durante código de servicio. Núcleo expulsable y no expulsable

Mientras se ejecuta código de servicio, las interrupciones estarán permitidas. Al aceptarse una interrupción se pasará a ejecutar su rutina de tratamiento. Ahora bien, la interrupción suele acarrear la ejecución de una fase de servicio. Nos encontramos, por tanto, con que es necesario ejecutar dos códigos de servicio, el que fue interrumpido y el que es requerido por la interrupción.

En los sistemas con **núcleo no expulsable** se completa siempre la fase de servicio en curso, antes de empezar el tratamiento de una nueva fase. Esta solución presenta un problema cuando la nueva fase es más prioritaria que la

interrumpida, puesto que hay que esperar a que complete una ejecución menos prioritaria antes de empezar con la más prioritaria. Nótese que la prioridad de un servicio está directamente relacionada con la prioridad del proceso que requiere dicho servicio más que con el tipo de servicio.

Este problema se evita en los sistemas con **núcleo expulsable**, puesto que permiten dejar a medio ejecutar una fase de un servicio para ejecutar otra más prioritaria. Evidentemente, esta solución es más flexible que la anterior, pero es más difícil de implementar, puesto que se presentan problemas de concurrencia en el código del sistema operativo cuando ambos servicios comparten estructuras de información.

Veamos con el ejemplo de la figura 3.31 la secuencia de ejecución para el caso de núcleo expulsable y no expulsable. Supongamos que se trata de dos procesos A y B, siendo A más prioritario que el B. En el instante t1 el proceso A pide leer del disco 1, por lo que el sistema operativo ejecuta la primera fase del servicio, mandando la orden al controlador del disco. En el instante t2, el proceso B solicita leer del disco 2. Mientras el sistema operativo está ejecutando la primera fase de dicho servicio, en el instante t3, se produce la interrupción del controlador del disco 1, indicando que ha terminado la lectura.

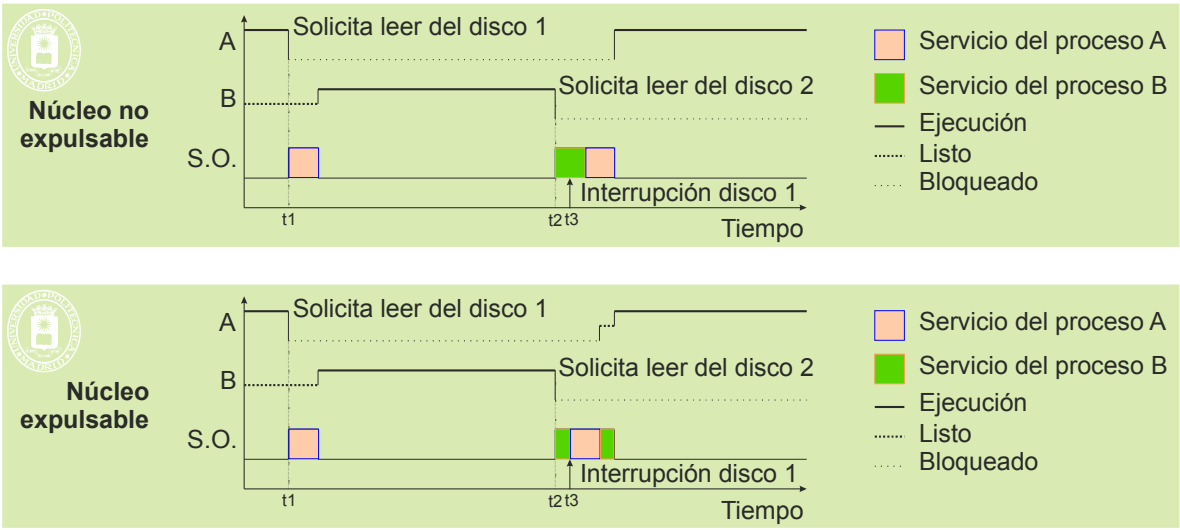


Figura 3.31 *Núcleo no expulsable y expulsable.*

En el caso de núcleo no expulsable se ejecuta la rutina de tratamiento de dicha interrupción y se aplaza la ejecución de la segunda fase del servicio del proceso A hasta completar la fase en curso del servicio del proceso B.

Por el contrario, en el caso de núcleo expulsable se ejecuta la rutina de tratamiento de dicha interrupción y se ejecuta a continuación la segunda fase del servicio del proceso A, dado que es más prioritario que el B. Seguidamente se completa la fase en curso del servicio del proceso B.

De igual forma que las rutinas de interrupción suelen permitir el anidamiento, el código del sistema operativo expulsable permite su anidamiento.

Interrupción durante código de usuario. Núcleo expulsivo y no expulsivo

En un sistema operativo no expulsivo un proceso permanecerá en ejecución hasta que él mismo solicite un servicio del sistema operativo. Evidentemente, durante su ejecución pueden aparecer interrupciones que serán tratadas por el sistema operativo, pero, finalizado el tratamiento, se volverá al proceso interrumpido.

Por el contrario, en un sistema operativo expulsivo, un proceso puede ser expulsado del procesador por diversas razones ajenas al mismo, como puede ser que ha pasado a listo un proceso más prioritario o que ha completado el tiempo de procesador preasignado. Como se verá en la sección "3.12 Planificación del procesador", en un sistema multiusuario se suele asignar a los procesos lo que se llama una **rodaja de tiempo** de procesador**,** que no es más que el tiempo máximo de ejecución continuada permitido. Una vez consumida su rodaja, el proceso es expulsado, esto es, devuelto a la cola de listos para ejecutar.

No debemos confundir expulsable con expulsivo. El primer término se aplica al propio sistema operativo, mientras que el segundo se aplica a los procesos de usuario.

Tratamiento de eventos síncronos y asíncronos

El tratamiento de un evento por parte de un proceso en modo privilegiado va a ser significativamente diferente dependiendo de que el evento sea de tipo síncrono o asíncrono.

Un evento síncrono (TRAP o excepción *hardware* síncrona) está vinculado al proceso en ejecución. Se trata, al fin y al cabo, de una solicitud de servicio por parte del proceso en ejecución, por tanto, es razonable que desde la rutina de tratamiento del evento se realicen operaciones que afecten directamente al proceso en ejecución. Así, desde la rutina de tratamiento se puede acceder al mapa del proceso para leer o escribir información en él. Por ejemplo, una llamada al sistema de lectura de un fichero o dispositivo requiere que su rutina de tratamiento acceda al mapa de memoria del proceso en ejecución para depositar en él los datos solicitados.

Cuando se realiza el tratamiento de un evento asíncrono (interrupción externa o excepción *hardware* asíncrona), aunque se lleve a cabo en el contexto del proceso en ejecución (o sea, el proceso en ejecución ejecute en modo

privilegiado el tratamiento del evento), el proceso no está vinculado con dicho evento. Por tanto, no tiene sentido que se realicen, desde la rutina de tratamiento, operaciones que afecten directamente al proceso interrumpido, como, por ejemplo, acceder a su imagen de memoria.

Como se aprecia en la figura 3.32, esta distinción en el tratamiento de estos dos tipos de eventos, incluso marca una división, aunque sólo sea conceptual, del sistema operativo en dos capas:

- Capa superior: rutinas del sistema operativo vinculadas al tratamiento de los eventos síncronos. Se podría decir que esta capa está más "próxima" a los procesos.
- Capa inferior: rutinas del sistema operativo vinculadas al tratamiento de interrupciones de dispositivos. Se podría decir que esta capa está más "próxima" al *hardware*.

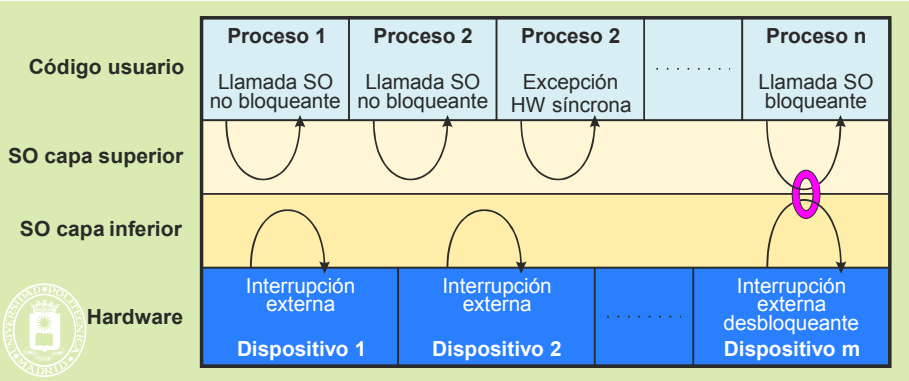


Figura 3.32 *Tratamiento de eventos síncronos y asíncronos.*

La dificultad de implementación estriba en la manipulación de las estructuras de datos que permiten «comunicar» ambas capas.

3.10.2. Detalle del tratamiento de interrupciones

En esta sección se analiza en detalle el tratamiento de las interrupciones, dado que es uno de los aspectos más importantes del núcleo del sistema operativo. Este tratamiento debe ser seguro y eficiente. Como existen distintos tipos de interrupción, que exigen un tratamiento distinto, se tratarán de forma individualizada los siguientes casos:

- Excepciones *hardware* síncronas.
- Excepciones *hardware* asíncronas.
- Interrupciones externas.
- Llamadas al sistema, que se analizan en la sección "3.10.3 Llamadas al sistema operativo".

Algunos aspectos del tratamiento de las interrupciones difieren de un tipo de sistema operativo a otro. Para el desarrollo de esta sección consideraremos la siguiente situación:

- Sistema operativo **no expulsable**. No se admite anidamiento en el código del sistema operativo.
- Sistema operativo **expulsivo**. Pueden existir cambios de contexto involuntarios.
- Máquina monoprocesador.
- Ejecución dentro de los procesos de usuario.

En los casos de núcleo expulsable y de máquina multiprocesadora aparecen en el código del sistema operativo problemas de concurrencia.

Interrupciones compartidas

Es muy frecuente que el controlador de interrupciones del procesador no disponga de suficientes líneas de interrupción como para dedicar una a cada dispositivo. Esto exige que se comparta una línea de interrupción entre varios dispositivos.

El *hardware* suele estar previsto para que se puedan compartir las interrupciones externas, por lo que no existen problemas de conexionado. Sin embargo, el *software* de tratamiento se complica, puesto que ha de descubrir qué dispositivo es el que realmente generó la solicitud de interrupción.

En el caso de las interrupciones internas puede ocurrir lo mismo. Es frecuente que exista una única interrupción para todas las excepciones *hardware* y una única interrupción para todas las llamadas al sistema.

Consideraciones generales al tratamiento de las interrupciones

Cuando se acepta una interrupción, el núcleo del sistema operativo arranca una nueva secuencia de ejecución propia para esa interrupción, que se ejecuta dentro del contexto de ejecución interrumpido, pero en modo privilegiado. Si el proceso ya estaba en modo privilegiado tratando una interrupción anterior, se inicia una nueva secuencia, dando lugar al anidamiento de las interrupciones.

Como ya se indicó en la sección "3.10 Tratamiento de interrupciones" , página 95, no todas las acciones requeridas para tratar una interrupción tienen la misma urgencia, dividiéndose en aplazables y no aplazables.

- Las **acciones no aplazables** a su vez se descomponen en críticas y no críticas y se engloban en el manejador de la interrupción, que debe ejecutarse de forma inmediata.
  - Las **acciones críticas** son las que hay que realizar bajo estrictas restricciones de tiempo, luego con todas las interrupciones desactivadas. Son acciones tales como reprogramar el controlador que ha interrumpido o actualizar determinadas estructuras de datos que utiliza dicho controlador. Algunas de estas acciones pueden realizarse en la rutina genérica analizada anteriormente.
  - Las **acciones no críticas** se llevan a cabo una vez completadas las acciones críticas y con las interrupciones ya permitidas (generalmente se activarán las de mayor nivel que la que se está tratando) y se refieren a la actualización de estructuras de información a las que sólo accede el sistema operativo (p. ej. almacenar en un *buffer* la tecla pulsada). Dado que los dispositivos *hardware* generalmente disponen de muy poca memoria hay que hacer las lecturas y escrituras en el controlador dentro de una pequeña ventana de tiempo. En caso contrario se sobrescribirían registros y se producirían errores e inconsistencias.
- Las **acciones aplazables** se pueden realizar algo más tarde y desacopladas del controlador del dispositivo que interrumpió. Son acciones tales como copiar la línea tecleada al *buffer* del usuario, tratar una trama, despertar un proceso o lanzar una operación de E/S.

En cada entrada de la tabla IDT de interrupciones se instala la dirección de una pequeña rutina que invoca una rutina genérica. Además, el sistema operativo asocia a cada tipo de interrupción una estructura de datos que, entre otras cosas, contiene la identificación de la o las rutinas no aplazables asociadas a ese tipo de interrupción.

La secuencia de eventos sería la siguiente:

- Cuando se produce la interrupción, el procesador realiza su labor y pasa el control a la rutina instalada en la correspondiente entrada IDT. Esta rutina suele ser una rutina genérica que realiza las siguientes funciones:
  - Salva en la pila de sistema del proceso interrumpido, puesto que es la que está activa, los registros visibles en modo usuario que no se hayan salvado de forma automática.
  - Cuando la interrupción es compartida realiza una búsqueda para determinar dicho origen y conocer el tipo de interrupción a tratar (en caso de una interrupción externa la búsqueda se basará en una encuesta o *polling*).
  - Dependiendo del procesador esta rutina se puede encargar de desinhibir las interrupciones, dejando activo el nivel de interrupción adecuado, o puede dejar esta labor a la rutina asociada.
  - Finalmente invoca a la rutina no aplazable asociada al tipo de interrupción.
- La rutina no aplazable realiza las operaciones urgentes del tratamiento de la interrupción y retorna a la rutina genérica.
- La rutina genérica restaura los registros almacenados anteriormente en la pila y ejecuta la instrucción de retorno de interrupción RETI.

Interrupciones *software*

Los sistemas operativos utilizan este mecanismo para múltiples labores, todas ellas relacionadas con la ejecución diferida de una operación, siendo las dos principales la ejecución de las acciones aplazables vinculadas con una interrupción y la realización de labores de planificación.

Siguiendo esta estrategia, en la rutina no aplazable sólo se realizan las acciones imprescindibles, activando una interrupción *software* de manera que, cuando haya terminado la rutina no aplazable, así como otras que estuvieran anidadas, se lleven a cabo las rutinas aplazables. Nótese que, dado que varias interrupciones han podido anidarse y aplazar la ejecución de sus operaciones no críticas, es necesario gestionar una estructura de datos que mantenga las operaciones pendientes de realizarse. Normalmente, se organiza como una cola donde cada rutina no aplazable almacena la identificación de una función que llevará a cabo sus operaciones aplazables, así como un dato que se le pasará como parámetro a la función cuando sea invocada. La rutina de tratamiento de la interrupción *software* vaciará la cola de trabajos invocando cada una de las funciones encoladas.

En algunos sistemas se permite establecer una prioridad a la hora de realizar la inserción en la cola de operaciones pendientes. Esta prioridad determina en qué posición de la cola queda colocada la petición y, por tanto, en qué orden será servida.

Es importante resaltar que, dado el carácter asíncrono de las interrupciones *software*, desde una rutina de interrupción *software* no se debe acceder al mapa del proceso actualmente en ejecución, ni causar un fallo de página, ni bloquear al proceso en ejecución.

Implementación de las interrupciones *software*

Algunos procesadores disponen de una instrucción privilegiada que produce una interrupción de prioridad mínima y que se utiliza para generar las interrupciones *software*. Si no se dispone de dicha instrucción se puede proceder de la siguiente forma: se refleja la activación de la interrupción *software* en una variable H. Además, cada vez que termina una rutina de tratamiento de interrupción se comprueba si se va a retornar a modo usuario. En el caso positivo se analiza la variable H. Si está activada se ejecuta una interrupción *software*, antes de retornar a modo usuario.

Tratamiento de las excepciones *hardware* síncronas

Con excepción del fallo de página, el tratamiento de excepciones *hardware* síncronas es relativamente sencillo. Por un lado, el sistema operativo no debe producir nunca excepciones *hardware* síncronas, puesto que indican una anomalía o que se está depurando el programa. Por otro lado, el tratamiento es simple y no requiere acciones aplazadas.

La secuencia de tratamiento queda esquematizada en la figura 3.33, y consta de los siguientes pasos:

- La unidad del computador que detecta la situación (la unidad aritmética, la unidad de control o la MMU) produce la interrupción (flecha 1).
- El procesador acepta la interrupción y, a través de la tabla de interrupciones (flecha 2), ejecuta la rutina genérica y salta a ejecutar la correspondiente rutina no aplazable (flecha 3), que realiza las operaciones detalladas en la figura.
- Puede ser necesario acceder a la unidad que ha generado la excepción *hardware* (flecha 4). Estas suelen ser acciones críticas que deben ser ejecutadas sin permitir otras interrupciones. Finalmente se permiten las interrupciones.
- Si el programa no está siendo depurado, se notifica al proceso la excepción *hardware* generada. Esto se realiza en UNIX enviando una señal, mientras que en Windows se realiza mediante una excepción. Si la señal o excepción no está tratada, el proceso terminará.
- Si el programa está siendo depurado, se detiene su ejecución y se notifica al proceso depurador la ocurrencia de la excepción. El depurador la tratará como considere oportuno, facilitando al programador la depuración del programa erróneo.
- Se retorna a la rutina genérica que restituye los registros y ejecuta un RETI.

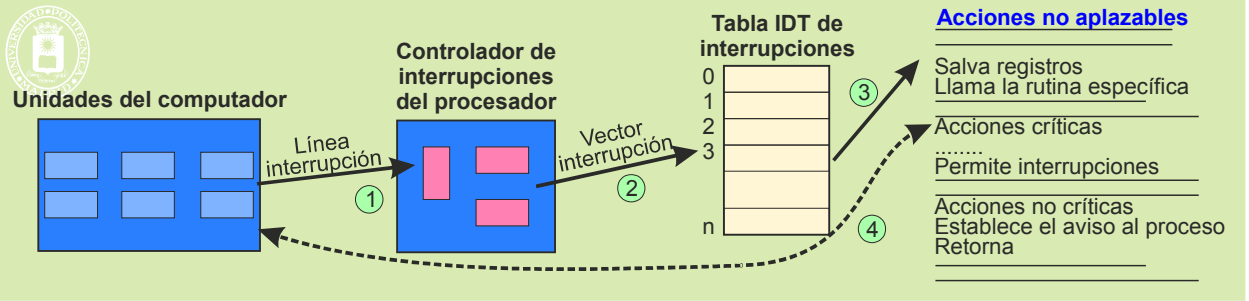


Figura 3.33 *Tratamiento de las excepciones hardware síncronas (que no sea fallo de página).*

El fallo de página es un caso peculiar, puesto que, por un lado, no se notifica al proceso y, por otro lado, exige que el sistema operativo realice la correspondiente migración de la página afectada. Como acción no aplazable se tiene la lectura de la dirección que ha causado el fallo de página y del tipo de acceso (flecha 4 de la figura 3.33), y como acciones aplazables la determinación del marco a utilizar, la generación de la orden al disco de paginación y el tratamiento de la finalización de la operación de migración.

En gran parte del código del sistema operativo se permiten los fallos de páginas, pero en algunas zonas están prohibidos, como, por ejemplo, en el código que trata un fallo de página, puesto que se produciría un bloqueo del sistema.

Tratamiento de las excepciones *hardware* asíncronas

Estas interrupciones indican situaciones de error en el *hardware* que, por lo general, impiden que el sistema siga ejecutando, como pueden ser el fallo de la alimentación o un error en la unidad de memoria.

La secuencia es similar a la reflejada en la figura 3.33. Dependiendo del tipo de error generado se puede hacer un reintento para ver si desaparece. Si el error persiste, la acción que se lleva a cabo es avisar al usuario del problema y cerrar el sistema lo más ordenadamente que se pueda.

Tratamiento de las interrupciones externas

Hay que destacar que, en este caso, la rutina no aplazable ejecuta bajo una situación desconocida: puede que se interrumpiese código del sistema operativo y que éste quede en un estado inconsistente (por ejemplo, a medio rellenar una entrada de una tabla interna), por lo que no se podrá utilizar la funcionalidad general del sistema operativo en dicha rutina. Además, ha de ejecutar muy rápidamente, por lo que no puede realizar operaciones que supongan un bloqueo o espera, como puede ser un fallo de página. Esto impone serias restricciones a la hora de codificar las rutinas no aplazables, puesto que sólo puede hacer uso de una funcionalidad reducida del sistema operativo y toda la información debe permanecer en memoria principal.

La secuencia de tratamiento de las interrupciones externas se muestra en la figura 3.34 y consiste en los siguientes pasos:

- El controlador del dispositivo genera la petición de interrupción (flecha 1).
- El procesador acepta la interrupción y, a través de la tabla de interrupciones (flecha 2), ejecuta la rutina genérica y salta a ejecutar la correspondiente rutina no aplazable (flecha 3). Si la interrupción es compartida, debe existir una fase de detección del dispositivo que generó la interrupción, lo que se puede hacer, por ejemplo, mediante una encuesta entre todos los dispositivos que comparten la interrupción. Esta acción puede ser crítica o no, dependiendo del *hardware* del sistema.

- Se completan las acciones críticas, si es que existen, y se permiten las interrupciones. Dichas acciones suelen exigir un diálogo con el controlador del dispositivo (flecha 4).
- Inscribe en la tabla de operaciones pendientes del SO a su rutina aplazable (flecha 6), para su posterior ejecución. En algunos casos puede no existir rutina aplazable o, por el contrario, pueden existir más de una, inscribiéndose todas ellas.
- Se activa la interrupción *software*.
- Se retorna a la rutina genérica, que restituye los registros y ejecuta un RETI.

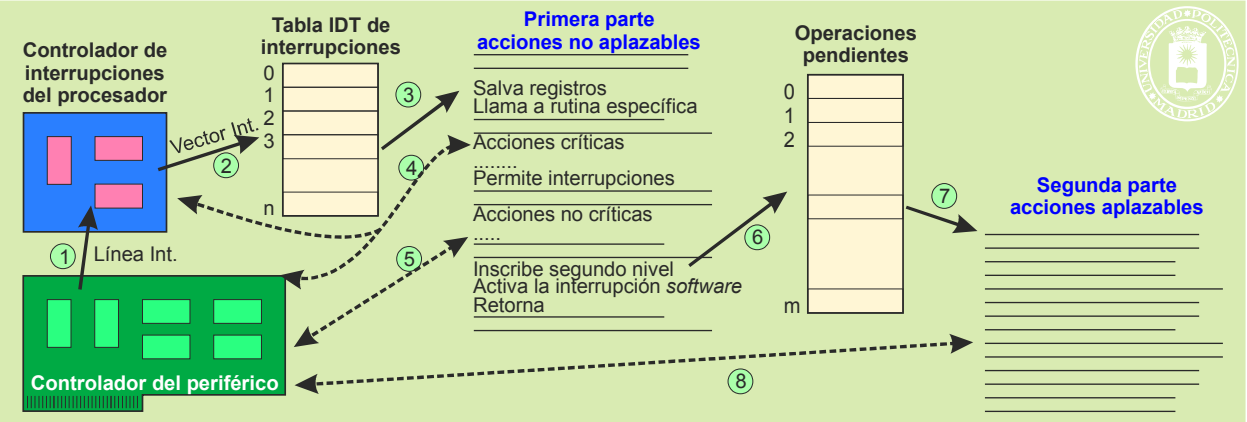


Figura 3.34 *Pasos existentes en el tratamiento de una interrupción externa.*

Cuando hayan retornado todas las interrupciones se ejecuta la interrupción *software*, que se encarga de ejecutar todas las rutinas anotadas en la tabla de operaciones pendientes (flecha 7). En los sistemas operativos no expulsables se cumple que, al ejecutar las rutinas aplazables, el sistema sí está en una situación conocida y consistente, por lo que el código de las mismas puede hacer uso de la funcionalidad general del sistema operativo. Además, ya no existe la premura de la primera parte, por lo que se pueden admitir operaciones que supongan una cierta espera. Según el caso, esta parte también puede acceder al controlador del periférico (flecha 8) y puede, a su vez, inscribir tareas en la tabla de operaciones pendientes para su posterior ejecución.

La figura 3.35 muestra un ejemplo de ejecución formado por una secuencia de 5 interrupciones, de las cuales la 1, la 3 y la 5 tienen operaciones aplazables. Se puede observar el anidamiento de las rutinas de interrupción 1, 2 y 3, y que, completadas la 3 y la 2, se anida la 4 sobre la 1. Supondremos que la interrupción 4 es de reloj y que se comprueba que el proceso A ha consumido su rodaja de tiempo, por lo que debe activarse la planificación. Una vez completadas todas las rutinas anteriores se ejecuta la interrupción *software* que busca en la tabla de operaciones pendientes las rutinas que han inscrito las interrupciones 1 y 3. Sin embargo, mientras se está ejecutando la rutina aplazada de la interrupción 3 aparece la interrupción 5, por lo que se trata antes de completar dicha rutina aplazada. Una vez completadas las rutinas aplazadas la interrupción *software* llama al planificador, que fue solicitado por la interrupción 4 y se produce un cambio de contexto, pasándose a ejecutar el proceso B.

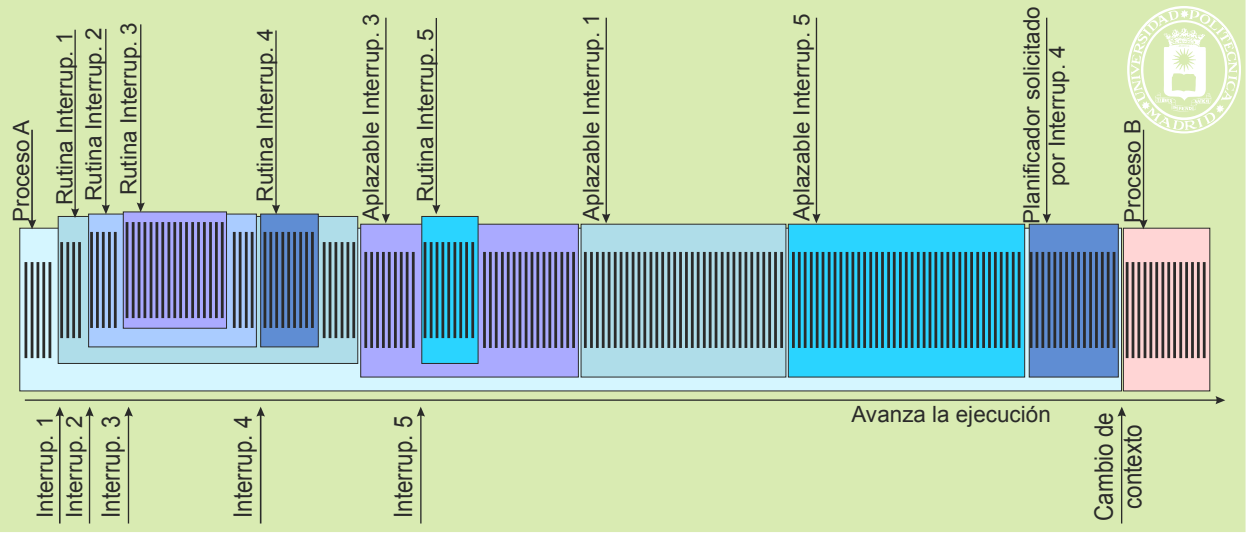


Figura 3.35 *Ejecución de las rutinas de interrupción y de las rutinas aplazadas.*

3.10.3. Llamadas al sistema operativo

En un sistema no expulsable no hay anidamiento de servicios, por lo que el sistema operativo está en un estado consistente y el código del servicio puede hacer uso de la funcionalidad general del sistema operativo.

Suele existir una única interrupción de solicitud de servicio, esto significa que todos los servicios empiezan ejecutando el mismo código. Se dice que hay una única puerta de entrada para los servicios. En procesadores Intel x86, Linux utiliza el vector de interrupción 0x80 como ventana de servicios, mientras que Windows utiliza el 0x2E.

El servicio puede requerir una espera (que bloqueará al proceso), como cuando se lee de disco, o no requerir espera, como cuando se cierra un fichero. Trataremos los dos casos por separado.

Cuando no hay espera, como se representa en la figura 3.36, se producen los siguientes pasos:

- La instrucción TRAP genera la interrupción de petición de servicio (flecha 1).
- El procesador acepta la interrupción, por lo que el proceso pasa de modo usuario a modo privilegiado.
- A través de la tabla de interrupciones (flecha 2) se ejecuta la rutina genérica que salva los registros visibles en la pila de sistema del proceso interrumpido. Seguidamente utiliza el identificador del servicio (almacenado en un registro) para entrar en la tabla de servicios (flecha 3) y determinar el punto de acceso del servicio solicitado.
- Llama al servicio (flecha 4) y ejecuta el correspondiente código.
- Se retorna a la rutina genérica que restituye los registros y ejecuta un RETI, con lo que se vuelve a la instrucción siguiente al TRAP (flecha 5).

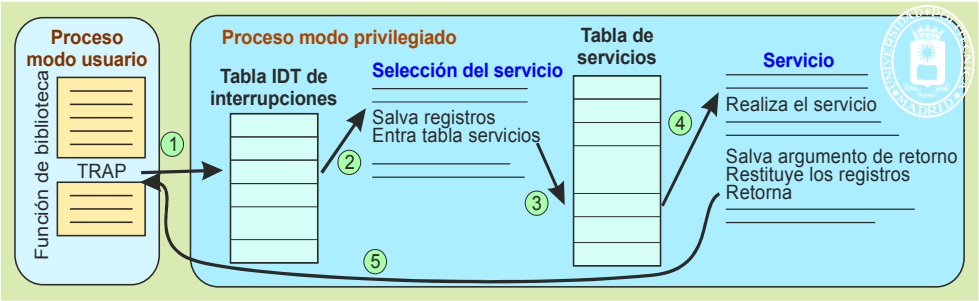


Figura 3.36 *Ejecución de un servicio que no presenta espera en un sistema operativo con ejecución dentro de los procesos de usuario.*

Según se desprende de la figura 3.36, cuando se completa el servicio se prosigue con el proceso que solicitó el servicio, ya en modo usuario. Esto no siempre es así, puesto que, durante la ejecución del servicio, pudo llegar una interrupción cuyo resultado fue poner en listo para ejecutar a un proceso, activando una interrupción *software* para ejecutar el planificador. Si el planificador selecciona otro proceso se haría un cambio de contexto y se pasaría a ejecutar dicho proceso.

Cuando el servicio contiene una espera, el tratamiento se divide en dos fases: una que inicia el servicio y otra que lo termina. La figura 3.37 muestra esta situación.

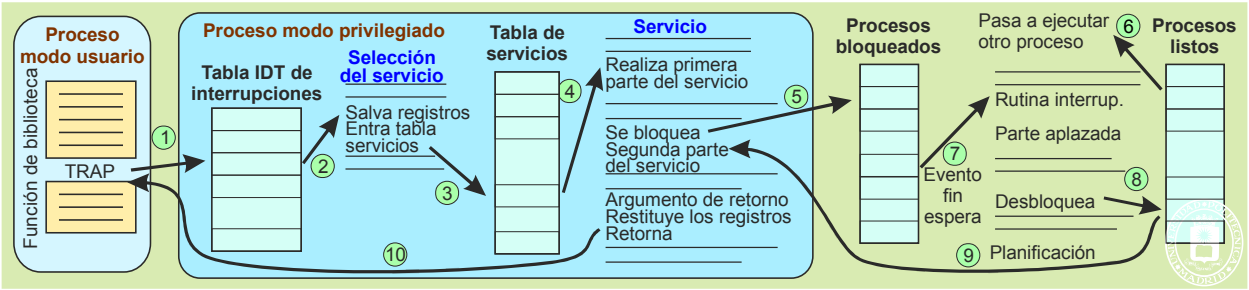


Figura 3.37 *Ejecución de un servicio que presenta espera en un sistema operativo con ejecución dentro de los procesos de usuario.*

La puesta en ejecución de la primera fase es igual que en el caso anterior, por lo que no se repetirá. El resto de la secuencia es como sigue:

- La primera fase inicia el servicio, por ejemplo, lanza la orden de lectura al disco. Seguidamente se ejecuta el planificador, el proceso queda bloqueado (flecha 5), y se pone en ejecución el proceso seleccionado (flecha 6), por lo que se produce un cambio de contexto.
- Más adelante, un evento indica el fin de la espera. Por ejemplo, el controlador del disco completa la lectura pedida y genera una interrupción. Esta interrupción ejecutará en el contexto de otro proceso (flecha 7) y podrá tener una parte aplazada.
- Si la operación se completó con éxito, el proceso pasa de bloqueado a listo (flecha 8).
- Cuando el planificador seleccione otra vez este proceso, seguirá su ejecución completando la segunda fase del servicio (flecha 9), por ejemplo, copiando al *buffer* del proceso la información leída del disco.
- Finalmente se genera el argumento de retorno del servicio, se restituyen los registros visibles y se retorna al proceso que sigue su ejecución en modo usuario (flecha 10).

3.10.4. Cambios de contexto voluntario e involuntario

Cambio de contexto voluntario

Con respecto a los cambios de contexto voluntarios, el programador del sistema operativo va a realizar una programación que podríamos considerar normal, pero va a incluir operaciones de cambio de contexto cuando así se requiera, debido a que, bajo ciertas condiciones, el proceso no pueda continuar su ejecución hasta que no se produzca un cierto evento.

Nótese que el código del bloqueo, que llamaremos `bloquear` debe elevar al máximo el nivel de interrupción del procesador puesto que está modificando la lista de procesos listos y es prácticamente seguro que todas las rutinas de interrupción manipulan esta lista, puesto que normalmente desbloquean algún proceso.

Téngase en cuenta que este tipo de cambios de contexto sólo pueden darse dentro de una llamada (o en el tratamiento de la excepción de fallo de página), por lo que sólo se podrá llamar a `bloquear` desde el código de una llamada. Asimismo, no se debe invocar desde una rutina de interrupción ya que el proceso en ejecución no está relacionado con la interrupción. Una determinada llamada puede incluir varias llamadas a `bloquear`, puesto que podría tener varias condiciones de bloqueo. Cuando el proceso por fin vuelva a ejecutar lo hará justo después de la llamada al cambio de contexto dentro de la rutina `bloquear`.

Cuando se produzca el evento que estaba esperando el proceso, éste se desbloqueará, pasando al estado de listo para ejecutar. Nótese que es un cambio de estado, pero no un cambio de contexto.

El código `desbloquear` podría ser invocado tanto desde una llamada al sistema como desde una interrupción y que, dado que manipula la lista de procesos listos, debería ejecutarse gran parte de la misma con el nivel de interrupción al máximo.

Cambio de contexto involuntario

Una llamada al sistema o una rutina de interrupción pueden desbloquear un proceso más importante o pueden indicar que el proceso actual ha terminado su turno de ejecución. Esta situación puede ocurrir dentro de una ejecución anidada de rutinas de interrupción.

Para asegurar el buen funcionamiento del sistema, hay que diferir el cambio de contexto involuntario hasta que terminen todas las rutinas de interrupción anidadas. La cuestión es cómo implementar este esquema de cambio de contexto retardado. La solución más elegante es utilizar el ya conocido mecanismo de interrupción *software* descrito anteriormente.

Con la interrupción *software* el cambio de contexto involuntario es casi trivial: cuando dentro del código de una llamada o una interrupción se detecta que hay que realizar, por el motivo que sea, un cambio de contexto involuntario, se activa la interrupción *software*. Dado que se trata de una interrupción del nivel mínimo, su rutina de tratamiento no se activará hasta que el nivel de interrupción del procesador sea el adecuado. Si había un anidamiento de rutinas de interrupción, todas ellas habrán terminado antes de activarse la rutina de la interrupción *software*. Esta rutina de tratamiento se encargará de realizar el cambio de contexto involuntario, que se producirá justo cuando se pretendía: en el momento en que han terminado las rutinas de interrupción.

Surge en este punto una decisión de diseño importante: en el caso de que dentro de este anidamiento en el tratamiento de eventos se incluya una llamada al sistema (que, evidentemente, tendrá que estar en el nivel más externo del anidamiento), ¿se difiere el cambio de contexto involuntario hasta que termine también la llamada o sólo hasta que termine la ejecución de las rutinas de tratamiento de interrupción? Dicho de otro modo, ¿el código de las llamadas al sistema se ejecuta con el procesador en un nivel de interrupción que inhabilita las interrupciones *software* o que las permite, respectivamente? Esta decisión tiene mucha trascendencia, dando lugar a dos tipos de sistemas operativos:

- Núcleo expulsable: si el cambio de contexto se difiere sólo hasta que terminen las rutinas de interrupción.
- Núcleo no expulsable: si el cambio de contexto se difiere hasta que termine también la llamada al sistema.

3.11. TABLAS DEL SISTEMA OPERATIVO

Como se muestra en la figura 3.7, página 76, el sistema operativo mantiene una serie de estructuras de información necesarias para gestionar los procesos, la memoria, los dispositivos de entrada/salida, los ficheros abiertos, etc. Estas tablas se irán viendo a lo largo del libro, pero podemos destacar las siguientes:

- **Procesos**. Tabla de procesos y colas de procesos.
- **Memoria**. Para la gestión de la memoria es necesario mantener:
  - Tablas de páginas, en los sistemas con memoria virtual.
  - Tabla con las zonas o marcos de memoria libre.
- **E/S**. El sistema operativo mantendrá una cola por cada dispositivo, en la que se almacenarán las operaciones pendientes de ejecución, así como la operación en curso de ejecución.
- **Ficheros**. Con información sobre los ficheros en uso.

Nos centraremos en esta sección en la información específica de los procesos.

Colas de procesos

Una de las estructuras de datos que más utiliza un sistema operativo para la gestión de procesos es la cola, o lista, de procesos. El sistema operativo enlaza en una lista todos los procesos que están en el mismo estado. Estas listas, evidentemente, estarán basadas en el tipo de lista genérico del sistema operativo correspondiente.

En todos los sistemas existe una lista de procesos listos que, normalmente, incluye el proceso en ejecución (o los procesos en ejecución, en el caso de un multiprocesador), aunque en teoría sean dos estados distintos. Normalmente, existe una variable global que hace referencia al proceso (o procesos) en ejecución.

En cuanto a los procesos bloqueados existen distintas alternativas a la hora de organizarlos en colas:

- Puede haber una única cola que incluya todos los procesos bloqueados, con independencia de cuál sea el motivo del bloqueo. Se trata de una solución poco eficiente, ya que desbloquear a un proceso esperando por un evento requiere recorrer toda la cola.
- Usar una cola por cada posible condición de bloqueo, por ejemplo, procesos bloqueados esperando datos de un determinado terminal o esperando que se libere un recurso. Esta es la solución usada, por ejemplo, en Linux. El único inconveniente, aunque tolerable, es el gasto de datos de control requeridos por cada cola.
- Utilizar una solución intermedia de manera que se hagan corresponder múltiples eventos a una misma cola.

Dado que cada proceso sólo puede estar en un estado en cada momento, basta con tener un puntero en el BCP (o dos, en el caso de que las colas tengan doble enlace) para insertar el proceso en la cola asociada al estado correspondiente.

Hay que resaltar que estas colas de procesos son una de las estructuras de datos más importantes y más utilizadas del sistema operativo. Cada cambio de estado requiere la transferencia del BCP de la cola que representa el estado previo a la cola correspondiente al nuevo estado.

Tabla de procesos

El sistema operativo asigna un BCP a cada proceso en la tabla de procesos. Sin embargo, no toda la información asociada a un proceso se encuentra en su BCP. La decisión de incluir o no una información en el BCP se toma según los dos argumentos de: eficiencia y necesidad de compartir información.

Eficiencia

Por razones de eficiencia, es decir, para acelerar los accesos, la tabla de procesos se construye en algunos casos como una estructura estática, formada por un número determinado de BCP del mismo tamaño. En este sentido, aquellas informaciones que pueden tener un tamaño variable no deben incluirse en el BCP. De incluirlas habría que reservar en cada BCP el espacio necesario para almacenar el mayor tamaño que puedan tener estas informaciones. Este espacio estaría presente en todos los BCP, pero estaría muy desaprovechado en la mayoría de ellos.

Un ejemplo de este tipo de información es la tabla de páginas, puesto que su tamaño depende de las necesidades de memoria de los procesos, valor que es muy variable de unos a otros. En este sentido, el BCP incluirá el RIED (Registro Identificador de Espacio de Direccionamiento) y una descripción de cada región (por ejemplo, incluirá la dirección virtual donde comienza la región, su tamaño, sus privilegios, la zona reservada para su crecimiento y el puntero a la subtabla de páginas, pero no la subtabla en sí).

Compartir la información

Cuando la información ha de ser compartida por varios procesos, no ha de residir en el BCP, cuyo acceso está restringido al proceso que lo ocupa. A lo sumo, el BCP contendrá un apuntador que permita alcanzar esa información.

Un ejemplo de esta situación lo presentan los procesos en relación con los punteros de posición de los ficheros que tienen abiertos. Dos procesos pueden tener simultáneamente abierto el mismo fichero por dos razones: el fichero se heredó abierto de un proceso ancestro o el fichero se abrió de forma independiente por los dos procesos. En el primer caso se trata de procesos diseñados para compartir el fichero, por lo que deben compartir el puntero de posición (PP). En el modelo UNIX existe una tabla única llamada intermedia que mantiene los punteros de posición de todos los ficheros abiertos por todos los procesos.

Finalmente diremos que otra razón que obliga a que las tablas de páginas sean externas al BCP es para permitir que se pueda compartir memoria. En este caso, como se analizará en detalle en el capítulo “4 Gestión de memoria”, dos o más procesos comparten parte de sus tablas de páginas.

3.12. PLANIFICACIÓN DEL PROCESADOR

La planificación de recursos es necesaria cuando múltiples usuarios necesitan usar de forma exclusiva un determinado recurso, que puede constar de uno o más ejemplares. La planificación determina en cada instante qué ejemplar se le asigna a cada uno de los usuarios que requiera utilizar el recurso en ese momento. Cuando existe más de un recurso la planificación lleva a cabo una **multiplexación espacial** de los recursos, puesto que determina qué recurso específico se asigna a cada usuario. Además, la planificación lleva a cabo una **multiplexación temporal** al determinar en qué instante se le asigna un recurso a un usuario.

Como muestra la figura 3.38, las solicitudes de los usuarios se pueden acumular, formándose una cola o varias colas de espera.

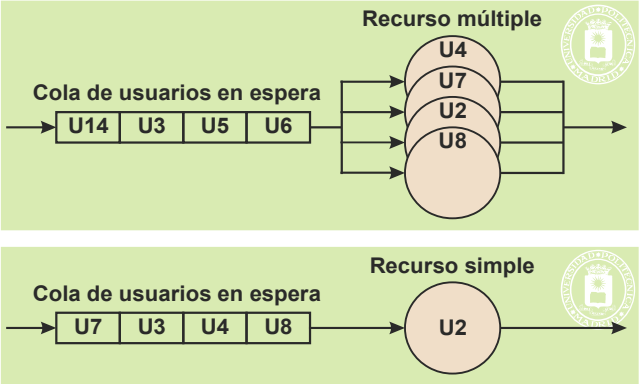


Figura 3.38 *Planificación en un sistema con múltiples ejemplares de un recurso y con recurso simple.*

Se presentan varias alternativas a la hora de realizar la planificación:

- Planificación **no expulsiva**. Una vez asignado un recurso a un usuario, éste lo mantendrá hasta que termine de usarlo. Este esquema de planificación sólo se activa cuando un ejemplar queda libre y existen usuarios en espera.
- Planificación **expulsiva**. Esta alternativa, se le puede quitar el recurso a un usuario para asignárselo a otro. Se dice que el recurso es **expropiable**. Este esquema de planificación, además de activarse cuando queda un recurso libre, también se activará en otras circunstancias, tales como cuando llegada una nueva petición o cuando el tiempo de uso de un recurso por parte de un usuario llega a un cierto plazo máximo.
- Planificación con **afinidad**. En ocasiones, un usuario puede restringir (afinidad **estricta**) los ejemplares de un recurso que puede utilizar, o puede preferir (afinidad **natural**) los ejemplares de un recurso que pueden utilizarse para satisfacer sus peticiones. En la afinidad estricta aunque exista un recurso libre, si no pertenece al subconjunto restringido por el usuario, no se le asigna.

En el caso del computador, los usuarios son los procesos o *threads* y, en cuanto a los recursos a planificar, existe una gran variedad, entre la que se pueden destacar los siguientes recursos:

- **El procesador**. Es el recurso básico del computador. Nótese que la cola de procesos listos se corresponde con la cola de espera de este recurso, que es de tipo expropiable (basta con salvar y restaurar los registros en el BCP) y en el que se presenta la propiedad de la afinidad, como se analizará en la sección "3.12 Planificación del procesador".
- **La memoria**. En los sistemas con memoria virtual, tal como se analizará en la sección "4.10 Aspectos de diseño de la memoria virtual", se produce una multiplexación espacial y temporal de los marcos de página.
- **El disco**. Los procesos realizan peticiones de acceso al disco que deben planificarse para determinar en qué orden se van sirviendo. En la sección "5.5.2 Almacenamiento secundario" se estudiarán los diversos algoritmos de planificación del disco.
- **Los mecanismos de sincronización**. Cuando un proceso deja libre un *mutex* que tenía cerrado, como se analizará en el capítulo "6 Comunicación y sincronización de procesos" , o cuando libera un cerrojo asociado a un fichero, como se verá en el mismo capítulo, hay que planificar a cuál de los procesos que están esperando usar este recurso, en caso de que haya alguno, se le asigna dicho recurso.

En las siguientes secciones trataremos la planificación del procesador.

3.12.1. Objetivos de la planificación

El objetivo de la planificación es optimizar el comportamiento del sistema informático. Ahora bien, este comportamiento es muy complejo, por tanto, el objetivo de la planificación se deberá centrar en la faceta del comportamiento en la que se esté interesado en cada situación.

Parámetros de evaluación del planificador

Para caracterizar el comportamiento de los algoritmos de planificación se suelen definir dos tipos de parámetros: de usuario (ya sea de proceso o de *thread*) y de sistema. Los primeros se refieren al comportamiento del sistema tal y como lo perciben los usuarios o los procesos. Los segundos se centran principalmente en el uso eficiente del procesador. Un proceso o *thread* se puede caracterizar con tres parámetros principales:

- **Tiempo de ejecución** (Te): Tiempo que tarda en ejecutarse un proceso o *thread* desde que se crea hasta que termina totalmente. Incluye todo el tiempo en que el proceso está listo para ejecutar, en ejecución y en estado bloqueado (por sincronización o entrada/salida).

- **Tiempo de espera** (Tw): Este parámetro define el tiempo que pasa un proceso en la cola de procesos listos para ejecutar. Si el proceso no se bloquea nunca, es el tiempo que espera el proceso o *thread* en estado listo para ejecutar antes de que pase al estado de ejecución.
- **Tiempo de respuesta** (Ta): Tiempo que pasa entre el momento en que se crea el proceso y se pone listo para ejecutar y la primera vez que el proceso responde al usuario. Es fundamental en los sistemas interactivos, ya que un sistema de planificación se puede configurar para responder muy rápido al usuario, aunque luego el proceso o *thread* tenga un tiempo de ejecución largo.

Desde el punto de vista de la planificación, un sistema se caracteriza con dos parámetros principales:

- **Uso del procesador** (C): Expresa en porcentajes el tiempo útil de uso del procesador partido por el tiempo total (Tu / T). Este parámetro varía mucho dependiendo de los sistemas. Por ejemplo, un procesador de sobremesa suele usarse menos de un 15%. Sin embargo, un servidor muy cargado puede usarse al 95%. Este parámetro es importante en sistemas de propósito general, pero es mucho más importante en sistemas de tiempo real y con calidad de servicio.
- **Tasa de trabajos completados** (P): Este parámetro indica el número de procesos o *threads* ejecutados completamente por unidad de tiempo. Se puede calcular como la inversa del tiempo medio de ejecución (1 / Media(Te)).

Entre los objetivos que se persiguen están los siguientes:

- Optimizar el uso del procesador para conseguir más eficiencia: max(C).
- Minimizar el tiempo de ejecución medio de un proceso o *thread*: min(Te).
- Minimizar el tiempo de respuesta medio en uso interactivo: min(Ta).
- Minimizar el tiempo de espera medio: min(Tw).
- Maximizar el número de trabajos por unidad de tiempo: max(P).

Aunque también se pueden perseguir objetivos más complejos como:

- Imparcialidad.
- Política justa. Realizar un reparto equitativo del procesador.
- Eficiencia: mantener el procesador ocupado el mayor tiempo posible con procesos de usuario.
- Predictibilidad en la ejecución. Cumplir los plazos de ejecución de un sistema de tiempo real
- Equilibrio en el uso de los recursos.
- Minimizar la varianza del tiempo de respuesta.
- Reducir el tiempo de cambio entre procesos o *threads*.
- Etcétera.

En general, en los sistemas operativos de propósito general se persiguen los objetivos enunciados arriba, es decir, maximizar el uso del procesador (C) y el número de procesos ejecutados por unidad de tiempo (P), al tiempo que se quiere minimizar el tiempo de ejecución de un proceso, su tiempo de respuesta y de espera (Te, Tw, Ta). En otros sistemas, como en los servidores interactivos, se suele primar la reducción del tiempo de respuesta (Tw), de forma que los clientes tengan la sensación de que el sistema les atiende rápidamente y no abandonen el servicio. En los sistemas de trabajo por lotes, se suele tratar de maximizar C y P, para explotar al máximo el procesador, como veremos más adelante.

3.12.2. Niveles de planificación de procesos

Los sistemas pueden incluir varios niveles de planificación de procesos. La figura 3.39 muestra el caso de tres niveles: corto, medio y largo plazo.

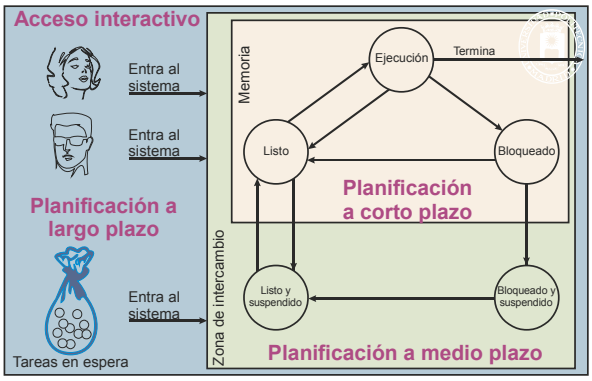


Figura 3.39 *Tipos de planificación.*

La planificación a **largo plazo** tiene por objetivo añadir nuevos procesos al sistema, tomándolos de la lista de espera. Estos procesos son procesos de tipo *batch*, en los que no importa el instante preciso en el que se ejecuten (siempre que se cumplan ciertos límites de espera).

La planificación a **medio plazo** trata la suspensión de procesos. Es la que decide qué procesos pasan a suspendido y cuáles dejan de estar suspendidos. Añade o elimina procesos de memoria principal modificando, por tanto, el grado de multiprogramación.

La planificación a **corto plazo** se encarga de seleccionar el proceso en estado de listo que pasa a estado de ejecución. Es, por tanto, la que asigna el procesador. Debido a la frecuencia con que se activa (alrededor de un centenar de milisegundos en un sistema de turno rotatorio), el planificador a corto plazo debe ser rápido y generar poca carga para el procesador.

3.12.3. Puntos de activación del planificador

Un aspecto importante de la planificación radica en los puntos del sistema operativo desde los que se invoca dicho algoritmo. Dichos puntos son los siguientes (donde pone proceso, entender proceso o *thread)*:

- Cambio de contexto voluntario
  - Cuando el proceso en ejecución termina su ejecución, ya sea voluntaria o involuntariamente.
  - Si el proceso en ejecución realiza una llamada bloqueante.
  - Si el proceso en ejecución causa una excepción que lo bloquea (por ejemplo, un fallo de página).
  - En caso de que el proceso en ejecución realice una llamada que ceda el uso del procesador, volviendo al final la cola de listos. Algunos sistemas operativos ofrecen llamadas de esta índole. En UNIX se corresponde con la llamada `pthread_yield`.
- Cambio de contexto involuntario
  - Si el proceso en ejecución realiza una llamada que desbloquea a un proceso más “importante” (el concepto de importancia dependerá de cada algoritmo, como veremos más adelante) que el mismo.
  - Cuando se produce una interrupción que desbloquea a un proceso más “importante” que el actual.
  - Si el proceso en ejecución realiza una llamada que disminuye su grado de “importancia” y hay un proceso que pasa a ser más “importante” que el primero.
  - En caso de que el proceso en ejecución cree un proceso más “importante” que el mismo.
  - Si se produce una interrupción de reloj que indica que el proceso en ejecución ha completado su turno y debe ceder el procesador.

Según se ha visto anteriormente, en el caso de los cambios de contexto involuntarios, una vez determinada la necesidad del cambio de contexto, el momento exacto en el que se realiza depende de qué tipo de sistema operativo se trate:

- **Núcleo no expulsable**. El cambio de contexto involuntario, y la invocación del planificador que le precede, se diferirá hasta justo el momento cuando se va a retornar a modo usuario. Por tanto, si la necesidad de cambio de contexto se ha producido dentro de una rutina de interrupción de máxima prioridad que está anidada con otras rutinas de interrupción de menor prioridad y con una llamada al sistema, se esperará a que se completen todas las rutinas de interrupción anidadas, así como la llamada al sistema en curso antes de realizarlo.
- **Núcleo expulsable**. El cambio de contexto involuntario se diferirá sólo hasta que se completen todas las rutinas de interrupción anidadas, en caso de que las haya. Por tanto, si se estaba ejecutando una llamada al sistema, ésta queda interrumpida.

Esta diferencia de comportamiento causa que la latencia de la activación de un proceso, y, por consiguiente, su tiempo de respuesta, sea mayor en los núcleos no expulsivos, puesto que, si hay una llamada al sistema en curso, hay que esperar que se complete, pudiendo ser considerablemente larga.

3.12.4. Algoritmos de planificación

Analizaremos los algoritmos más relevantes, destacando que los planificadores suelen emplear una combinación de ellos con extensiones heurísticas.

Primero en llegar primero en ejecutar FCFS

El algoritmo FCFS (*First Come First Served*) selecciona al usuario que lleva más tiempo esperando en la cola de listos.

Es un algoritmo sencillo, que introduce muy poca sobrecarga en el sistema, lo que permite obtener el máximo rendimiento del procesador. Como no es expulsivo los usuarios mantienen el recurso hasta que dejan de necesitarlo. Por ejemplo, un proceso que recibe el procesador la mantiene hasta que requiere una operación de entrada/salida o de sincronización. Por ello, este algoritmo beneficia a los procesos intensivos en procesamiento frente a los intensivos en entrada/salida. Puede producir largos tiempos de espera, por lo no es adecuado para los sistemas interactivos, estando limitado su posible uso a sistemas de lotes. Sin embargo, todo proceso llega a ejecutar, por lo que no produce inanición.

El trabajo más corto (SJF)

El algoritmo SJF (*Shortest Job First*) busca el trabajo con la ráfaga de procesamiento más corta de la cola de procesos listos. Es necesario, por tanto, disponer de la duración de las ráfagas de procesamiento, lo que es imposible, a memos que se trate de trabajos por lotes repetitivos, en los que se conoce la duración de las ráfagas de ejecución de los mismos. Una alternativa es estimar estas ráfagas extrapolando los valores de ráfaga anteriores de cada proceso.

Minimiza el tiempo de espera medio Tw, en base a penalizar los trabajos largos. Presenta el riesgo de inanición de los usuarios de larga duración.

Puede incurrir en sobrecarga del sistema, puesto que hay que recorrer la cola de procesos listos, para buscar el más corto, o bien tener dicha cola ordenada, además, de la estimación de las ráfagas, en su caso.

Existe una versión expulsiva de este algoritmo denominado Primero el de menor tiempo restante o SRTF (*Shortest Remaining Time First*)

Planificación basada en prioridades

Este planificador selecciona al usuario con la mayor prioridad. Si existen varios usuarios (por ejemplo, procesos listos) con igual prioridad se utiliza otro de los algoritmos, por ejemplo, el FCFS, para seleccionar al agraciado. Tiene la ventaja de proporcionar grados de urgencia.

El sistema mantiene una cola de usuarios por prioridad, como se muestra en la figura 3.40.

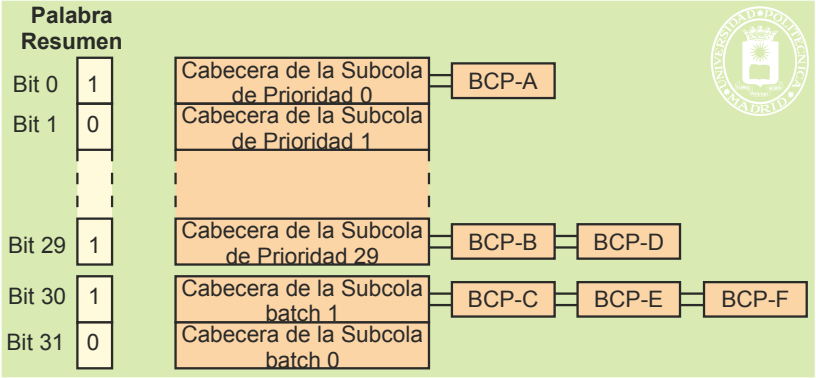


Figura 3.40 *Ejemplo de colas de procesos organizadas por prioridad. La palabra resumen permite acelerar la búsqueda, puesto que solamente hay que analizar las niveles que tienen el resumen a 1.*

Este algoritmo puede producir inanición en los usuarios de baja prioridad. Para evitar este problema se utilizan prioridades dinámicas y mecanismos de envejecimiento, que se encargan de aumentar la prioridad a los usuarios que lleven un determinado tiempo esperando a ser atendidos.

Existe una versión expulsiva de este algoritmo, de forma que si llega un usuario más prioritario que el que está ejecutando, se expulsa a éste para dar el recurso al más prioritario.

La tabla 3.2 muestra las clases de planificación y las prioridades que se asignan a cada una de ellas tanto en el núcleo de Linux como en el de Windows

Tabla 3.2 *Niveles de prioridad en el núcleo de Linux y Windows.*

Parámetro	Linux	Windows
Clases de planificación	3	2
1. Prioridades normales (dinámicas)	40; de -20 a 19	15; de 1 a 15
2. Prioridades de tiempo real FCFS (fijas)	100; de 0 a 99	16; de 16 a 31
3. Prioridades de tiempo real con rodaja (fijas)	100; de 0 a 99	—
Orden de importancia de la prioridad	Baja→Alta	Alta→Baja

Turno rotatorio (round robin)

Este algoritmo se utiliza para repartir de forma equitativa el procesador entre los procesos listos, proporcionando un tiempo respuesta (Ta) acotado. Tiene la ventaja de ofrecer reparto equitativo.

Es una variación del algoritmo FCFS, puesto que se concede el procesador por un tiempo máximo acotado, denominado **rodaja** o **cuanto**. Si se cumple la rodaja sin que el proceso abandone voluntariamente el procesador, éste es expulsado y puesto al final de la cola, como se puede observar en la figura 3.41. Igualmente, cuando un proceso pasa de bloqueado a listo se pone la final de la cola.

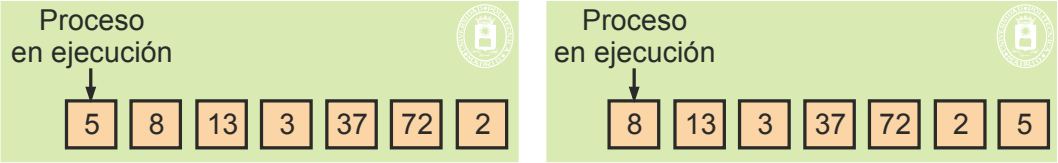


Figura 3.41 *En el algoritmo round robin cuando un proceso deja el procesador por haber consumido su rodaja pasa al último lugar de la cola.*

Es un algoritmo expulsivo, que solamente expulsa el proceso si ha consumido enteramente su rodaja.

Un aspecto importante es el dimensionado de la rodaja. Si es muy grande el algoritmo tiende a ser igual que el FCFS. Si, por el contrario, es muy pequeña introduce mucha sobrecarga en el sistema. En el diseño de la rodaja existen varias alternativas, como las siguientes:

- Rodaja igual para todos los procesos.
- Rodaja distinta según el tipo de proceso.
- Rodaja dinámica, cuyo valor se ajusta según sea el comportamiento del proceso o el comportamiento global del sistema.

Colas multinivel

Los procesos se organizan en distintas colas y se aplica un algoritmo de planificación distinto a cada cola. Por ejemplo, en la figura 3.40 se puede observar que hay 30 colas de prioridad y dos colas batch. En las colas de prioridad se puede aplicar un round robin entre los procesos de cada una, mientras que en las colas *batch* se puede aplicar un SJF con expulsión si aparece un proceso con prioridad.

El esquema de colas multinivel se caracteriza por los siguientes parámetros:

- El número de niveles existentes, es decir, el número de clases de procesos que se distinguen.
- El algoritmo de planificación de cada nivel.
- El esquema de planificación que se usa para repartir el procesador entre los distintos niveles.

Para evitar la inanición se puede añadir un mecanismo de envejecimiento, de forma que pasado un cierto tiempo esperando en una cola se pase al nivel siguiente.

En la figura 3.42 se muestra un ejemplo con tres niveles sin realimentación, tal que en cada nivel se usa un algoritmo de turno rotatorio con distinto tamaño de la rodaja. En cuanto al mecanismo de planificación entre niveles, se usa un esquema de prioridad tal que sólo se ejecutarán procesos de un determinado nivel si en los niveles de mayor prioridad no hay ningún proceso listo.

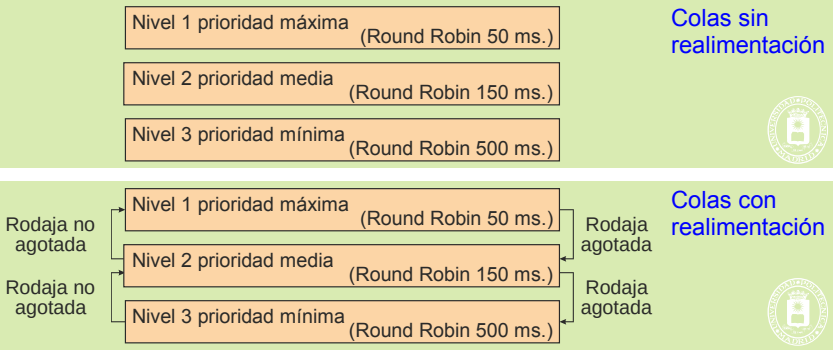


Figura 3.42 *Colas multinivel con y sin realimentación*

En el modelo de colas con realimentación, el sistema operativo puede cambiar de nivel a un proceso dependiendo de su evolución. En este nuevo modelo existe un parámetro adicional: la política de cambio de nivel, que establece en qué circunstancias un proceso incluido en un determinado nivel pasa a formar parte de otro nivel. En el caso de la figura 3.42 se puede establecer que un proceso que no agota su rodaja se le sube de nivel, mientras que un proceso que agota su rodaja se baja de nivel. Con esta estrategia, los procesos con un uso intensivo de la entrada/salida, estarán en el nivel 1, otorgándoles la mayor prioridad, mientras que los intensivos en el uso del procesador se situarán en el nivel 3, con la menor prioridad, quedando en el nivel intermedio los procesos con un perfil mixto.

3.12.5. Planificación en multiprocesadores

La planificación en multiprocesadores se puede hacer en base a mantener una única bolsa de procesos listos para todo el sistema o mantener una bolsa de procesos listos por cada procesador.

Un aspecto importante en la planificación en multiprocesadores es la **afinidad**, puesto que cuando un proceso ejecuta en un determinado procesador, en la jerarquía de memorias caches se va almacenando la información de su conjunto de trabajo. Si el proceso vuelve a ejecutar en el mismo procesador, habrá cierta probabilidad de que encuentre en la cache de ese procesador información suya, disminuyendo, por tanto, los fallos de cache.

Planificación basada en una cola única

En este caso se mantiene una única estructura de datos en el sistema que incluye todos los procesos listos, ya que, como se vio previamente, muchos algoritmos organizan los procesos listos en varias colas para hacer más eficiente la gestión. Este sistema presenta equilibrado automático de la carga, no permitiendo que un procesador esté libre si hay un proceso listo para ejecutar.

Esta solución presenta un problema importante: dado que todas las decisiones del planificador requieren consultar la cola de listos y que, por tanto, se requiere un cerrojo para evitar los problemas de coherencia durante su ma-

nipulación, los accesos a esta estructura se convierten en un cuello de botella que limita severamente la capacidad de crecimiento del sistema.

Planificación basada en una cola por procesador

Con este esquema cada procesador se planifica de manera independiente, como si fuera un sistema uniprocesador, según muestra la figura 3.43. Cuando se desbloquea un proceso asociado a un procesador, se incorpora a la cola de listos del mismo. Asimismo, cuando se produce un cambio de contexto voluntario en un procesador, sólo se busca en la cola de procesos listos de ese procesador. Con este esquema, por tanto, no hay congestión en el acceso a las estructuras de datos del planificador, puesto que cada procesador consulta las suyas. Además, asegura directamente un buen aprovechamiento de la afinidad al mantener un proceso en el mismo procesador.

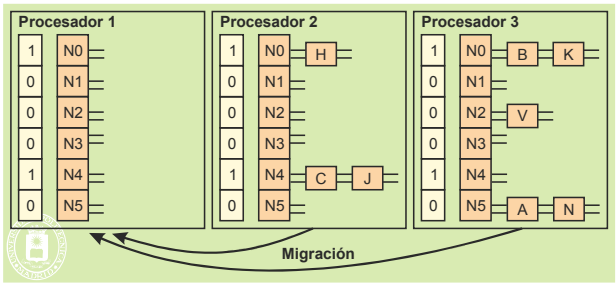


Figura 3.43 *Planificación basada en una cola por procesador.*

Este sistema no presenta equilibrado automático de la carga, por lo que hay que añadir mecanismos explícitos que realicen esta labor, analizando la carga de todos los procesadores y migrando procesos listos a los procesadores menos cargados.

3.13. SERVICIOS

Esta sección describe los principales servicios que ofrecen UNIX y Windows para la gestión de procesos, *threads* y planificación. También se presentan los servicios que permiten trabajar con señales (UNIX), excepciones (Windows) y temporizadores.

3.13.1. Servicios UNIX para la gestión de procesos

En esta sección se describen los principales servicios que ofrece UNIX para la gestión de procesos. Estos servicios se han agrupado según las siguientes categorías:

- Identificación de procesos.
- El entorno de un proceso.
- Creación de procesos.
- Terminación de procesos.

En las siguientes secciones se presentan los servicios incluidos en cada una de estas categorías, utilizando sus prototipos en lenguaje C.

Identificación de procesos

UNIX identifica cada proceso por medio de un entero único denominado identificador de proceso de tipo `pid_t`. Los servicios relativos a la identificación de los procesos son los siguientes:

- pid_t **getpid**(void);

Este servicio devuelve el identificador del proceso que lo solicita.

- pid_t **getppid**(void);

Devuelve el identificador del proceso padre.

El programa 3.1 muestra un ejemplo de utilización de ambos servicios.

Programa 3.1 Programa que imprime su identificador de proceso y el identificador de su proceso padre.

```
#include <sys/types.h>
#include <stdio.h>

int main(void)
{
    pid_t id_proceso;
```

```
    pid_t id_padre;

    id_proceso = getpid();
    id_padre = getppid();

    printf("Identificador de proceso: %d\n", id_proceso);
    printf("Identificador del proceso padre: %d\n", id_padre);
    return 0;
}
```

Cada usuario en el sistema tiene un identificador único denominado identificador de usuario, de tipo `uid_t`. Cada proceso lleva asociado un usuario que se denomina propietario o usuario real. El proceso tiene también un identificador de usuario efectivo, que determina los privilegios de ejecución que tiene el proceso. Generalmente el usuario real es el mismo que el efectivo. El sistema incluye también grupos de usuarios, cada usuario debe ser miembro al menos de un grupo. Al igual que con los usuarios, cada proceso lleva asociado el identificador de grupo real al que pertenece y el identificador de grupo efectivo. Los servicios que permiten obtener estos identificadores son los siguientes:

- uid_t **getuid**(void);

Este servicio devuelve el identificador de usuario real del proceso que lo solicita.

- uid_t **geteuid**(void);

Devuelve el identificador de usuario efectivo.

- gid_t **getgid**(void);

Este servicio permite obtener el identificador de grupo real.

- gid_t **getegid**(void);

Devuelve el identificador de grupo efectivo.

El programa 3.2 muestra un ejemplo de utilización de estos cuatro servicios.

Programa 3.2 Programa que imprime la información de identificación de un proceso.

```
#include <sys/types.h>
#include <stdio.h>
int main(void)
{
    printf("Identificador de usuario: %d\n", getuid());
    printf("Identificador de usuario efectivo: %d\n",     geteuid());
    printf("Identificador de grupo: %d\n", getgid());
    printf("Identificador de grupo efectivo: %d\n", getegid());
    return 0;
}
```

- int **setuid**(uid_t uid);

Si el proceso que lo ejecuta tiene UID efectiva de root se cambian el usuario real y el efectivo por el valor `uid`. Si no es privilegiado es igual al **`seteuid`**.

Este servicio lo ejecuta el proceso login, que inicialmente es root, una vez autenticado un usuario para dejarle un *shell* con la UID

- int **seteuid**(uid_t euid);

Establece el usuario efectivo. Si el proceso que lo ejecuta no tiene UID efectiva de root, solamente puede poner como efectivo su real, por ejemplo:

```
seteuid (getuid);
```

Si el proceso tiene UID efectiva de root cambia el UID efectivo.

Si un programa que tienen identidades real y efectiva de root desea asumir temporalmente la identidad de un usuario sin privilegios y luego recuperar sus privilegios de root, lo puede hacer mediante este servicio, al no perder su identidad real de root.

El entorno de un proceso

El entorno de un proceso viene definido por una lista de variables que se pasan en el momento de comenzar su ejecución. Estas variables se denominan variables de entorno y se puede acceder a ellas a través de la variable global `environ`, declarada de la siguiente forma:

```
extern char *environ[];
```

El entorno es un vector de cadenas de caracteres de la forma `nombre=valor`, donde `nombre` hace referencia al nombre de una variable de entorno y `valor` al contenido de la misma.

Este mismo entorno lo recibe el `main` como tercer parámetro. Basta con declararlo de la siguiente forma:

int main(int argc, char *argv[], char *envp[])

El programa 3.3 imprime las variables de entorno de un proceso.

Programa 3.3 Programa que imprime el entorno del proceso dos veces, usando `environ` y `envp`.

```
#include <stdio.h>
#include <stdlib.h>

extern char **environ;

int main(int argc, char *argv[], char *envp[])
{
   int i;

   printf("Lista de variables de entorno usando environ de %s\n", argv[0]);
   for(i=0; environ[i] != NULL; i++)
      printf("environ[%d] = %s\n", i, environ[i];

   printf("Lista de variables de entorno usando envp de %s\n", argv[0]);
   for(i=0; envp[i] != NULL; i++)
      printf("envp[%d] = %s\n", i, envp[i];
   return 0;
}
```

Cada aplicación interpreta la lista de variables de entorno de forma específica. UNIX establece el significado de determinadas variables de entorno. Las más comunes son:

- HOME, directorio de trabajo inicial del usuario.
- LOGNAME, nombre del usuario asociado a un proceso.
- PATH, prefijo de directorios para encontrar ejecutables.
- TERM, tipo de terminal.
- TZ, información de la zona horaria.

◙ char ***getenv**(const char *name);

El servicio `getenv` permite obtener el valor de una variable de entorno. Devuelve un puntero al valor de la variable de entorno de nombre `name`, o `NULL` si la variable de entorno no se encuentra definida.

El programa 3.4 utiliza el servicio `getenv` para imprimir el valor de la variable de entorno HOME.

Programa 3.4 Programa que imprime el valor de la variable HOME.

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char *home;

   home = getenv("HOME");
   if (home == NULL)
      printf("HOME no se encuentra definida\n");
   else
      printf("El valor de HOME es %s\n", home);
   return 0;
}
```

◙ int **putenv**(const char *string);

El servicio `putenv` permite añadir o cambiar el valor de una variable de entorno. Devuelve un 0 en caso de éxito y un -1 en caso de erro.

El argumento `string` tiene el formato nombre=valor. Si el nombre no existe, se añade `string` al entorno. Si el nombre existe se le cambia el valor.

La cadena apuntada por `string` se convierte en parte del entorno, por tanto, si se cambia su contenido se cambia el entorno.

Creación de procesos

◙ pid_t **fork**();

La forma de crear un proceso en un sistema operativo que ofrezca la interfaz UNIX es invocando el servicio `fork`. El sistema operativo trata este servicio realizando una clonación del proceso que lo solicite. El proceso que solicita el servicio se convierte en el proceso padre del nuevo proceso, que es, a su vez, el proceso hijo.

La figura 3.44 muestra que la clonación del proceso padre se realiza copiando la imagen de memoria y el BCP. Observe que el proceso hijo es una copia del proceso padre en el instante en que éste solicita el servicio `fork`. Esto significa que los datos y la pila del proceso hijo son los que tiene el padre en ese instante de ejecución. Es más, dado que al entrar el sistema operativo a tratar el servicio, lo primero que hace es salvar los registros en el BCP del padre, al copiarse el BCP se copian los valores salvados de los registros, por lo que el hijo tiene los mismos valores que el padre. Esto significa, en especial, que el contador de programa de los dos procesos tiene el mismo valor, por lo que van a ejecutar la misma instrucción máquina. No hay que caer en el error de pensar que el proceso hijo empieza la ejecución del código en su punto de inicio; repetimos: el hijo empieza a ejecutar, al igual que el padre, en la sentencia que esté después del `fork`.

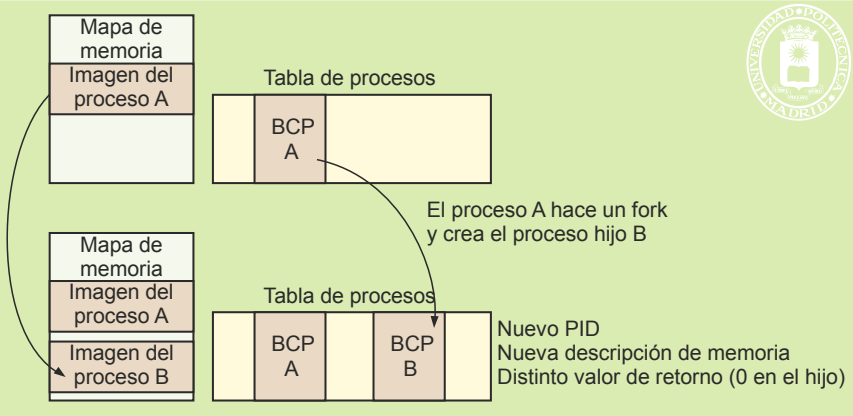


Figura 3.44 *Creación de un proceso mediante el servicio fork.*

El servicio `fork` es invocado una sola vez por el padre, pero retorna dos veces, una en el padre y otra en el hijo.

En realidad el proceso hijo no es totalmente idéntico al padre, puesto que algunos de los valores del BCP han de ser distintos. Las diferencias más importantes son las siguientes:

- El proceso hijo tiene su propio identificador de proceso, único en el sistema.
- El proceso hijo tiene una nueva descripción de la memoria. Aunque el hijo tenga las mismas regiones con el mismo contenido, no tienen por qué estar en la misma zona de memoria (esto es especialmente cierto en el caso de sistemas sin memoria virtual).
- El tiempo de ejecución del proceso hijo se iguala a cero.
- Todas las alarmas pendientes se desactivan en el proceso hijo.
- El conjunto de señales pendientes se pone a vacío.
- El valor que retorna el sistema operativo como resultado del `fork` es distinto:
  - El hijo recibe un «0».
  - El padre recibe el identificador de proceso del hijo.

Este valor de retorno se puede utilizar mediante una cláusula de condición para que el padre y el hijo sigan flujos de ejecución distintos, como se muestra en la figura 3.45, donde el hijo ejecuta un exec.

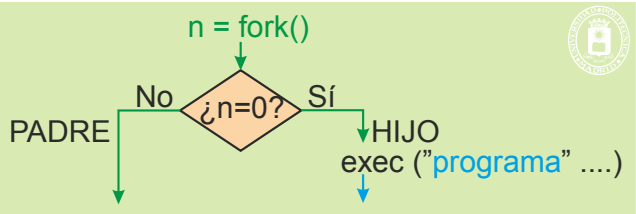


Figura 3.45 *Uso frecuente del servicio fork.*

La cláusula de condición puede estar basada en un if o en un switch, como muestra la figura 3.46.

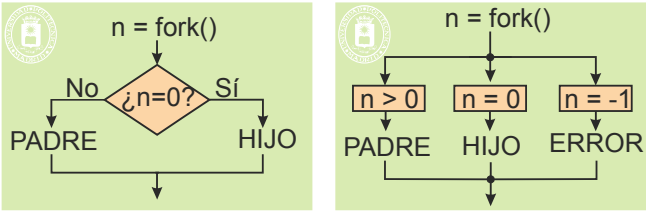


Figura 3.46 *Ejemplos de flujo de control después de un fork, mediante if y mediante switch.*

Observe que las modificaciones que realice el proceso padre sobre sus registros e imagen de memoria después del `fork` no afectan al hijo y, viceversa, las del hijo no afectan al padre. Sin embargo, el proceso hijo tiene su propia copia de los descriptores del proceso padre. Esto hace que el hijo tenga acceso a los ficheros abiertos por el proceso padre. Además, padre e hijo comparten los punteros de posición de los ficheros abiertos hasta ese momento. Esta es la única forma por la cual se pueden compartir punteros de ficheros.

El programa 3.5 muestra un ejemplo de utilización del servicio `fork`. Este programa hace uso de la función de biblioteca `perror` que imprime un mensaje describiendo el error del último servicio ejecutado. Después del servicio `fork`, los procesos padre e hijo imprimirán sus identificadores de proceso utilizando el servicio `getpid`, y los identificadores de sus procesos padre, por medio del servicio `getppid`. Observe que los identificadores del proceso padre son distintos en cada uno de los dos procesos.

Programa 3.5 Programa que crea un proceso.

```
#include <sys/types.h>
#include <stdio.h>

int main(void)
{
   pid_t pid;

   pid = fork();
   switch(pid){
      case -1: /* error del fork() */
         perror("fork");
         break;
      case 0:   /* proceso hijo */
         printf("Proceso %d; padre = %d\n", getpid(), getppid());
         break;
      default: /* padre */
         printf("Proceso %d; padre = %d\n", getpid(), getppid());
   }
   return 0;
}
```

El código del programa 3.6 crea una cadena de *n* procesos como se muestra en la figura 3.47.

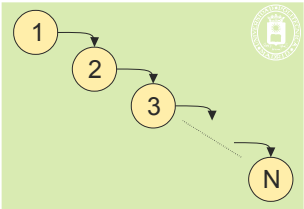


Figura 3.47 *Cadena de procesos*

Programa 3.6 Programa que crea la cadena de procesos de la figura 3.47.

```
#include <sys/types.h>
#include <stdio.h>

int main(void)
{
   pid_t pid;
   int i;
   int n = 10;

   for (i = 0; i > n; i++){
      pid = fork();
```

```
        if (pid != 0)
            break;
    }
    printf("El padre del proceso %d es %d\n", getpid(), getppid());
    return 0;
}
```

En cada ejecución del bucle se crea un proceso. El proceso padre obtiene el identificador del proceso hijo, que será distinto de cero y saldrá del bucle utilizando la sentencia `break` de C. El proceso hijo continuará la ejecución con la siguiente iteración del bucle. Esta recursión se repetirá hasta que se llegue al final del bucle.

El programa 3.7 crea un conjunto de procesos cuya estructura se muestra en la figura 3.48.

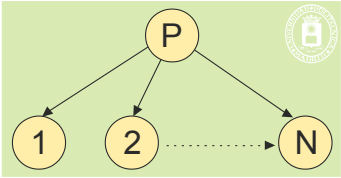


Figura 3.48 *Creación de n procesos hijos.*

Programa 3.7 Programa que crea la estructura de procesos de la figura 3.48.

```
#include <stdio.h>
#include <sys/types.h>

int main(void)
{
    pid_t pid;
    int i;
    int n = 10;

    for (i = 0; i > n; i++){
        pid = fork();
        if (pid == 0)
            break;
    }
    printf("El padre del proceso %d es %d\n", getpid(), getppid());
    return 0;
}
```

En este programa, a diferencia del anterior, es el proceso hijo el que sale del bucle ejecutando la sentencia `break`, siendo el padre el encargado de crear todos los procesos.

Cambiar el programa que ejecuta un proceso

El servicio `exec` de UNIX tiene por objetivo cambiar el programa que está ejecutando un proceso. Se puede considerar que el servicio tiene tres fases. En la primera se comprueba que el servicio se puede realizar sin problemas. En la segunda se vacía el proceso de casi todo su contenido. Una vez iniciada esta fase no hay marcha atrás. En la tercera se carga un nuevo programa. La figura 3.49 muestra estas dos fases.

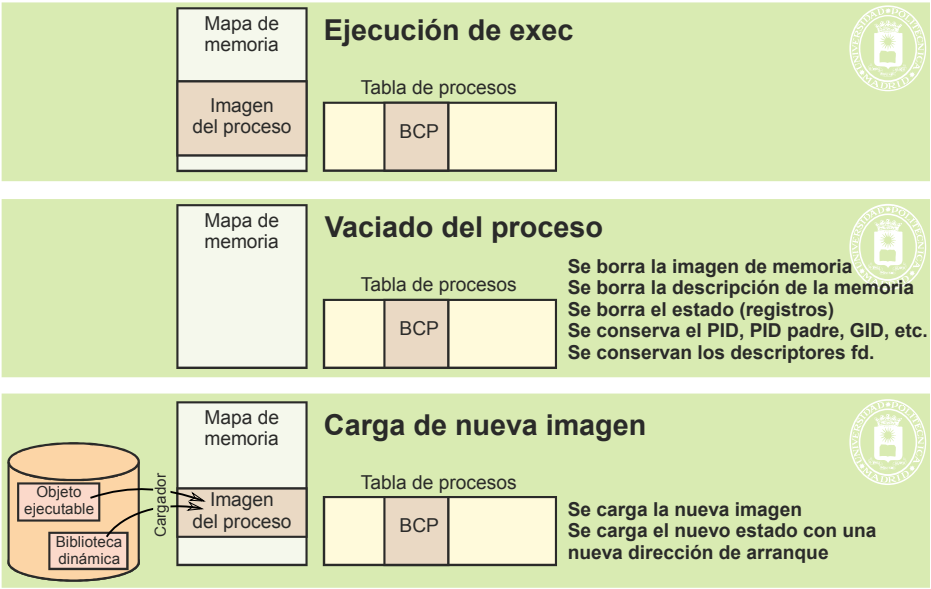


Figura 3.49 *Funcionamiento del servicio exec.*

En la fase de vaciado del proceso se conservan algunas informaciones, como:

- Entorno del proceso, que el sistema operativo incluye en la nueva pila del proceso.
- Algunas informaciones del BCP como: identificador de proceso, identificador del proceso padre, identificador de usuario y descriptores de ficheros abiertos.

En la fase de carga hay que realizar las siguientes operaciones:

- Asignar al proceso un nuevo espacio de memoria.
- Cargar el texto y los datos iniciales en las regiones correspondientes.
- Crear la pila inicial del proceso con el entorno y los argumentos que se pasan al programa.
- Rellenar el BCP con los valores iniciales de los registros y la descripción de las nuevas regiones de memoria.

Recuerde que el servicio `fork` crea un nuevo proceso, que ejecuta el mismo programa que el proceso padre, y que el servicio `exec` no crea un nuevo proceso, sino que permite que un proceso pase a ejecutar un programa distinto.

En UNIX existe una familia de funciones `exec`, cuyos prototipos se muestran a continuación:

- int **execl**(char *path, char *arg, ...);
- int **execv**(char *path, char *argv[]);
- int **execle**(char *path, char *arg, ...);
- int **execve**(char *path, char *argv[], char *envp[]);
- int **execlp**(char *file, const char *arg, ...);
- int **execvp**(char *file, char *argv[]);

La familia de funciones `exec` reemplaza la imagen del proceso por una nueva imagen. Esta nueva imagen se construye a partir de un fichero ejecutable. Si el servicio se ejecuta con éxito, éste no retorna, puesto que la imagen del proceso habrá sido reemplazada, en caso contrario devuelve -1.

La función `main` del nuevo programa llamado tendrá la forma:

int main(int argc, char *argv[])

donde `argc` representa el número de argumentos que se pasan al programa, incluido el propio nombre del programa, y `argv` es un vector de cadenas de caracteres, conteniendo cada elemento de este vector un argumento pasado al programa. El primer componente de este vector (`argv[0]`) representa el nombre del propio programa.

El argumento `path` apunta al nombre del fichero ejecutable donde reside la nueva imagen del proceso. El argumento `file` se utiliza para construir el nombre del fichero ejecutable. Si el argumento `file` contiene el carácter «`/`», entonces el argumento `file` constituye el nombre del fichero ejecutable. En caso contrario, el prefijo del nombre para el fichero se construye por medio de la búsqueda en los directorios pasados en la variable de entorno PATH.

El argumento `argv` contiene los argumentos pasados al programa y debería acabar con un puntero `NULL`. El argumento `envp` apunta al entorno que se pasará al proceso y se puede obtener de la variable externa `environ`.

Los descriptores de los ficheros abiertos previamente por el proceso que solicita el servicio `exec` permanecen abiertos en la nueva imagen del proceso, excepto aquellos con el *flag* `FD_CLOEXEC`. Los directorios abiertos en el proceso que solicita el servicio serán cerrados en la nueva imagen del proceso.

Las señales con la acción por defecto seguirán por defecto. Las señales ignoradas seguirán ignoradas por el proceso y las señales con manejadores activados tomarán la acción por defecto.

Si el fichero ejecutable tiene activo el bit de modo `set-user-ID` (aclaración 3.7), el identificador efectivo del proceso pasará a ser el identificador del propietario del fichero ejecutable.

Aclaración 3.7. Cuando un usuario ejecuta un fichero ejecutable con el bit de modo *set-user-ID* activo, el nuevo proceso creado tendrá como identificador de usuario efectivo el identificador de usuario del propietario del fichero ejecutable. Este identificador es el que se utiliza para comprobar los permisos en los accesos a determinados recursos como son los ficheros. Por tanto, cuando se ejecuta un programa con este bit activo, el proceso ejecuta con la identidad del propietario del fichero ejecutable no con la suya.

Después del servicio `exec` el proceso mantiene los siguientes atributos:

- Identificador de proceso.
- Identificador del proceso padre.
- Identificador del grupo del proceso.
- Identificador de usuario real.
- Identificador de grupo real.
- Directorio actual de trabajo.
- Directorio raíz.
- Máscara de creación de ficheros.
- Máscara de señales del proceso.
- Señales pendientes.

En el programa 3.8 se muestra un ejemplo de utilización del servicio `execlp`. Este programa crea un proceso hijo, que ejecuta el mandato `ls -l`, para listar el contenido del directorio actual de trabajo.

Programa 3.8 Programa que ejecuta el mandato `ls -l`.

```
#include <sys/types.h>
#include <stdio.h>

int main(void)
{
   pid_t pid;
   int status;

   pid = fork();
   switch(pid){
      case -1: /* error del fork() */
         return 1;
      case 0:   /* proceso hijo */
         execlp("ls","ls","-l",NULL);
         perror("exec");
         return 2;
      default: /* padre */
         printf("Proceso padre\n");
   }
   return 0;
}
```

Igualmente, el programa 3.9 ejecuta el mandato `ls -l` haciendo uso del servicio `execvp`.

Programa 3.9 Programa que ejecuta el mandato `ls -l` mediante el servicio `execvp`.

```
#include <sys/types.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
   pid_t pid;
   char *argumentos[3];

   argumentos[0] = "ls";
   argumentos[1] = "-l";
   argumentos[2] = NULL;

   pid = fork();
   switch(pid)    {
      case -1: /* error del fork() */
```

```
            return 1;
        case 0:   /* proceso hijo */
            execvp(argumentos[0], argumentos);
            perror("exec");
            return 2;
        default: /* padre */
            printf("Proceso padre\n");
    }
    return 0;
}
```

El programa 3.10 crea un proceso que ejecuta un mandato recibido en la línea de argumentos.

Programa 3.10 Programa que ejecuta el mandato pasado en la línea de mandatos.

```
#include <sys/types.h>
#include <stdio.h>

main(int argc, char *argv[])
{
    pid_t pid;

    pid = fork();
    switch(pid)    {
        case -1: /* error del fork() */
            return 1;
        case 0:   /* proceso hijo */
            execvp(argv[1], &argv[1]);
            perror("exec");
            return 2;
        default: /* padre */
            printf("Proceso padre\n");
    }

    return 0;
}
```

Terminación de procesos

Un proceso puede terminar su ejecución de forma normal o anormal. La terminación normal se puede realizar de cualquiera de las tres formas siguientes:

- Ejecutando una sentencia `return` en la función `main`.
- Ejecutando la función `exit`.
- Mediante el servicio `_exit`.

◙ void **_exit**(int status);

El servicio `_exit` tiene por misión finalizar la ejecución de un proceso. Recibe como argumento un valor entre 0 y 255, que sirve para que el proceso dé una indicación de cómo ha terminado. Como se verá más adelante, esta información la puede recuperar el proceso padre que, de esta forma, puede conocer cómo ha terminado el hijo. La finalización de un proceso tiene las siguientes consecuencias:

- Se cierran todos los descriptores de ficheros.
- La terminación del proceso no finaliza de forma directa la ejecución de sus procesos hijos.
- Si el proceso padre del proceso que solicita el servicio se encuentra ejecutando un servicio `wait` o `waitpid` (su descripción se realizará a continuación), se le notifica la terminación del proceso.
- Si el proceso padre no se encuentra ejecutando un servicio `wait` o `waitpid`, el código de finalización del `exit` se salva hasta que el proceso padre ejecute la llamada `wait` o `waitpid`.
- Si la implementación soporta la señal SIGCHLD, ésta se envía al proceso padre.
- El sistema operativo libera todos los recursos utilizados por el proceso.

◙ void **exit**(int status);

En realidad, `exit` es una función de biblioteca que llama al servicio `_exit` después de preparar la terminación ordenada del proceso. Cuando un programa ejecuta la sentencia `return valor;` dentro de la función `main`, el efecto es idéntico al de `exit(valor)`. El `exit` puede usarse desde cualquier parte del programa.

En general, se recomienda utilizar la función `exit` en vez del servicio `_exit`, puesto que es más portable y permite volcar a disco los datos no actualizados. La función `exit` realiza los siguientes pasos:

- Todas las funciones registradas con la función estándar de C `atexit`, son llamadas en orden inverso a su registro. Si cualquiera de estas funciones llama a `exit`, los resultados no serán portables. La sintaxis de esta función es: `int atexit(void (*func)(void));`
- Vacía los almacenamientos intermedios asociados a las funciones de entrada/salida del lenguaje.
- Se llama al servicio `_exit`.

Un proceso también puede terminar su ejecución de forma anormal por la recepción de una señal que provoca su finalización o llamando a la función `abort`. La señal puede estar causada por un evento externo (p. ej.: se pulsa `CTLR-C`), por una señal enviada por otro proceso, o por un error de ejecución, como, por ejemplo, la ejecución de una instrucción ilegal o un acceso ilegal a una posición de memoria.

Cuando un proceso termina de forma anormal, generalmente, se produce un fichero denominado *core* que incluye la imagen de memoria del proceso en el momento de producirse su terminación y que puede ser utilizado para depurar el programa.

El programa 3.11 finaliza su ejecución utilizando la función `exit`. Cuando se llama a esta función se ejecuta la función `fin`, registrada previamente por medio de la función `atexit`.

Programa 3.11 Ejemplo de utilización de `exit` y `atexit`.

```
#include <stdio.h>
#include <stdlib.h>

void fin(void)
{
   printf("Fin de la ejecución del proceso %d\n", getpid());
}

void main(void)
{
   if (atexit(fin) != 0)  {
      perror("atexit");
      exit(1);
   }

   exit(0); /* provoca la ejecución de la función fin */
   return 0;
}
```

- pid_t **wait**(int *status);
- pid_t **waitpid**(pid_t pid, int *status, int options);

Ambos servicios permiten a un proceso padre quedar bloqueado hasta que termine la ejecución de un proceso hijo, obteniendo información sobre el estado de terminación del mismo.

El servicio `wait` suspende la ejecución del proceso hasta que finaliza la ejecución de uno de sus procesos hijos y devuelve el identificador del proceso hijo cuya ejecución ha finalizado. Si `status` es distinto de `NULL`, entonces en esta variable se almacena información relativa al proceso que ha terminado. Si el hijo retornó un valor desde la función `main`, utilizando la sentencia `return`, o pasó un valor como argumento a `exit`, éste valor se puede obtener utilizando las macros definidas en el fichero de cabecera `sys/wait.h`. Como macros del lenguaje C recuerde que valen 0 para indicar falso y cualquier otro valor para verdadero.

- `WIFEXITED(status)`: devuelve un valor verdadero si el hijo terminó normalmente.
- `WEXITSTATUS(status)`: permite obtener el valor devuelto por el proceso hijo en el servicio `exit` o el valor devuelto en la función `main`, utilizando la sentencia `return`. Esta macro sólo puede ser utilizada cuando `WIFEXITED` devuelve un valor verdadero.
- `WIFSIGNALED(status)`: devuelve un valor verdadero si el proceso hijo finalizó su ejecución como consecuencia de la recepción de una señal para la cual no se había programado manejador.
- `WTERMSIG(status)`: devuelve el número de la señal que provocó la finalización del proceso hijo. Esta macro sólo puede utilizarse si `WIFSIGNALED` devuelve un valor verdadero.
- `WIFSTOPPED(status)`: devuelve un valor verdadero si el estado fue devuelto por un proceso hijo actualmente suspendido. Este valor sólo puede obtenerse en el servicio `waitpid` con la opción `WUNTRACED`, como se verá a continuación.
- `WSTOPSIG(status)`: devuelve el número de la señal que provocó al proceso hijo suspenderse. Esta macro sólo puede emplearse si `WIFSTOPPED` devuelve un valor distinto de cero.

El servicio `waitpid` tiene el mismo funcionamiento si el argumento `pid` es -1 y el argumento `options` es cero. Su funcionamiento en general es el siguiente:

- Si `pid` es -1, se espera la finalización de cualquier proceso.
- Si `pid` es mayor que cero, se espera la finalización del proceso hijo con identificador `pid`.
- Si `pid` es cero, se espera la finalización de cualquier proceso hijo con el mismo identificador de grupo de proceso que el del proceso que solicita servicio.
- Si `pid` es menor que -1, se espera la finalización de cualquier proceso hijo cuyo identificador de grupo de proceso sea igual al valor absoluto del valor de `pid`.
- El argumento `options` se construye mediante el OR binario de cero o más valores definidos en el fichero de cabecera `sys/wait.h`. De especial interés son los siguientes:
  - `WNOHANG`. El servicio `waitpid` no suspenderá el proceso que lo solicita si el estado del proceso hijo especificado por `pid` no se encuentra disponible.
  - `WUNTRACED`. Permite que el estado de cualquier proceso hijo especificado por `pid`, que esté suspendido, sea devuelto al programa que solicita el servicio `waitpid`.

El programa 3.12 ejecuta un mandato recibido en la línea de argumentos por la función `main`. En este programa, el proceso padre solicita el servicio `wait` para esperar la finalización del mandato. Una vez concluida la ejecución, el proceso padre imprime información sobre el estado de terminación del proceso hijo.

Programa 3.12 Programa que imprime información sobre el estado de terminación de un proceso hijo.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
   pid  t pid;
   int valor;

   pid = fork();
   switch(pid)    {
      case -1: /* error del fork() */
         return 1;
      case 0:   /* proceso hijo */
         execvp(argv[1], &argv[1]);
         perror("exec");
         return 2;
      default: /* padre */
         while (wait(&valor) != pid) continue;

         if (valor == 0)
            printf("El mandato se ejecutó de forma normal\n");
         else {
            if (WIFEXITED(valor))
               printf("El hijo terminó normalmente y su valor devuelto
                           fue %d\n", WEXITEDSTATUS(valor));

            if (WIFSIGNALED(valor))
               printf("El hijo terminó al recibir la señal %d\n",
      WTERMSIG(valor));
         }
    }
   return 0;
}
```

La utilización de los servicios `fork`, `exec`, `wait` y `exit` hacen variar la jerarquía de procesos (véase figura 3.50). Con el servicio `fork` aparecen nuevos procesos y con el `exit` desaparecen. Sin embargo, existen algunas situaciones particulares que convienen analizar.

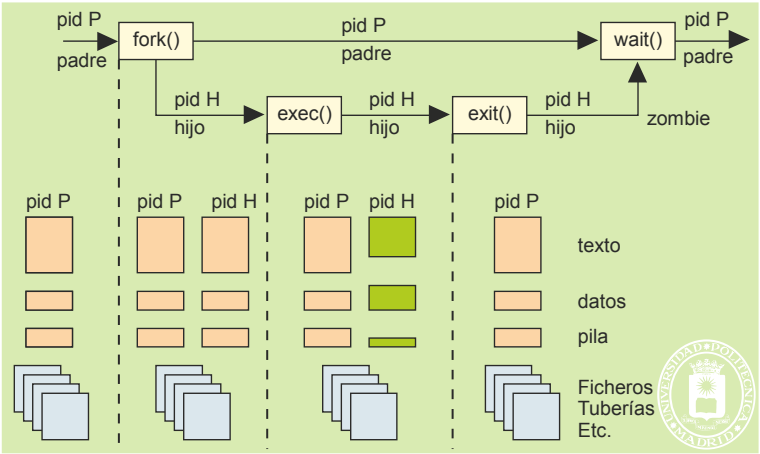


Figura 3.50 *Uso de los servicios fork, exec, wait y exit.*

La primera situación se refleja en la figura 3.51. Cuando un proceso termina y se encuentra con que el padre no está bloqueado en un servicio `wait`, se presenta el problema de dónde almacenar el estado de terminación que el hijo retorna al padre. Esta información no se puede almacenar en el BCP del padre, puesto que puede haber un número indeterminado de hijos que hayan terminado. Por ello, la información se deja en el BCP del hijo, hasta que el padre la adquiera mediante el oportuno `wait`. Un proceso muerto que se encuentra esperando el `wait` del padre se dice que está en estado **zombi**. El proceso ha devuelto todos sus recursos con excepción del BCP, que contiene el pid del padre y de su estado de terminación.

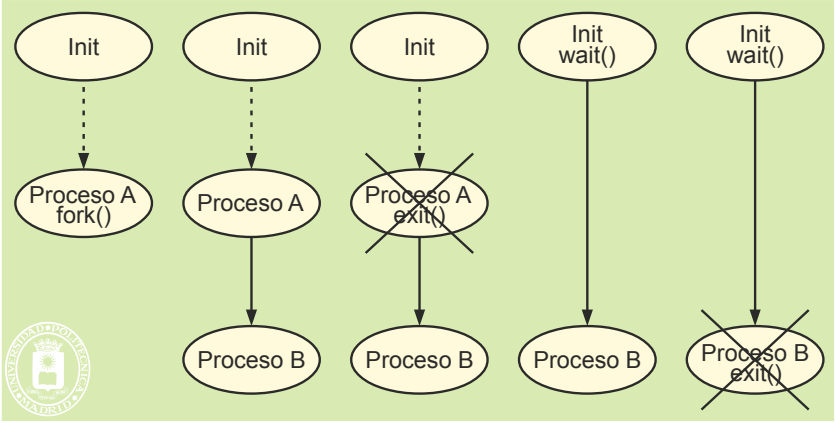


Figura 3.51 *En UNIX el proceso init hereda los procesos hijos que se quedan sin padre.*

La segunda situación se presenta en la figura 3.52 y se refiere al caso de que un proceso con hijos termine antes que estos. De no tomar alguna acción correctora, estos procesos contendrían en su BCP una información de pid del padre obsoleta, puesto que ese proceso ya no existe. La solución adoptada en UNIX es que estos procesos «huérfanos» los toma a su cargo el proceso *init*. En concreto, el proceso B de la mencionada figura pasará a tener como padre al *init*.

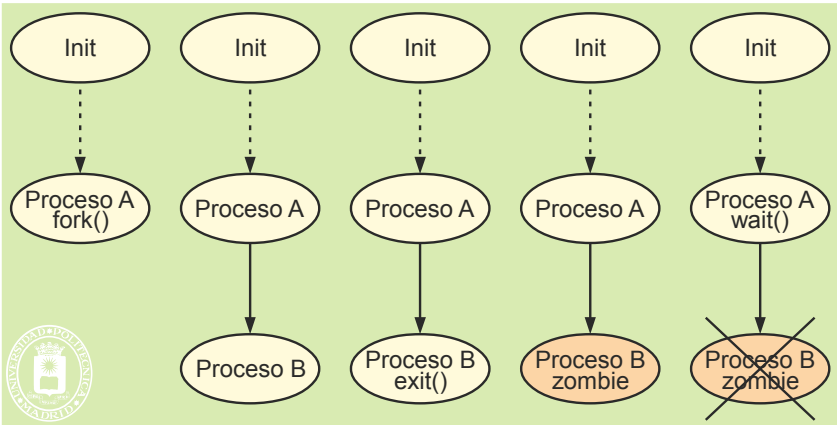


Figura 3.52 *Proceso zombi.*

Proceso zombi.

Para que los procesos heredados por *init* acaben correctamente, y no se conviertan en zombis, el proceso *init* está en un bucle infinito de `wait`.

La mayoría de los intérpretes de mandatos (*shells*) permiten ejecutar mandatos en *background*, finalizando la línea de mandatos con el carácter &. Así, cuando se ejecuta el siguiente mandato:

```
ls -l &
```

el *shell* ejecutará el mandato `ls -l` sin esperar la terminación del proceso que lo ejecutó. Esto permitiría al intérprete de mandatos estar listo para ejecutar otro mandato, el cual puede ejecutarse de forma concurrente a `ls -l`. Los procesos en *background*, además, se caracterizan porque no se pueden interrumpir con `CTLR-C`.

El programa 3.13 crea un proceso que ejecuta en *background* el mandato pasado en la línea de argumentos.

Programa 3.13 Ejecución de un proceso en background.

```
#include <sys/types.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    pid_t pid;

    pid = fork();
    switch(pid){
        case -1: /* error del fork() */
            return 1;
        case 0:   /* proceso hijo */
            execvp(argv[1], &argv[1]);
            perror("exec");
            return 2;
        default: /* padre */
    }
    return 0;
}
```

3.13.2. Servicios UNIX de gestión de *threads*

Existen *threads* de distintos fabricantes de UNIX. En esta sección se describen los principales servicios del estándar UNIX relativos a la gestión de *threads*. Estos servicios se han agrupado de acuerdo a las siguientes categorías:

- Atributos de un *thread*.
- Creación e identificación de *threads*.
- Terminación de *threads*.

Atributos de un *thread*

Cada *thread* en UNIX tiene asociado una serie de atributos que representan sus propiedades. Los valores de los diferentes atributos se almacenan en un objeto atributo de tipo `pthread_attr_t`. Existen una serie de servicios que se aplican sobre el tipo anterior y que permiten modificar los valores asociados a un objeto de tipo atributo. A continuación, se describen las principales funciones relacionadas con los atributos de los *threads*.

- int **pthread_attr_init**(pthread_attr_t *attr);

Este servicio permite iniciar un objeto atributo que se puede utilizar para crear nuevos *threads*.

- int **pthread_attr_destroy**(pthread_attr_t *attr);

Destruye el objeto de tipo atributo pasado como argumento a la misma.

- int **pthread_attr_setstacksize**(pthread_attr_t *attr, int stacksize);

Cada *thread* tiene una pila. Este servicio permite definir el tamaño de la pila.

- int **pthread_attr_getstacksize**(pthread_attr_t *attr, int *stacksize);

Permite obtener el tamaño de la pila de un *thread*.

- int **pthread_attr_setdetachstate**(pthread_attr_t *attr, int detachstate);

Si el valor del argumento `detachstate` es `PTHREAD_CREATE_DETACHED`, el *thread* que se cree con esos atributos se considerará como independiente y liberará sus recursos cuando finalice su ejecución. Si el valor del argumento `detachstate` es `PTHREAD_CREATE_JOINABLE`, el *thread* se crea como no independiente y no liberará sus recursos cuando finalice su ejecución. En este caso es necesario que otro *thread* espere por su finalización. Esta espera se consigue mediante el servicio `pthread_join`, que se describirá más adelante.

- int **pthread_attr_getdetachstate**(pthread_attr_t *attr, int *detachstate);

Permite conocer si es `DETACHED` o `JOINABLE`.

Creación e identificación de *threads*

Los servicios relacionados con la creación e identificación de *threads* son los siguientes:

- int **pthread_create**(pthread_t *thread, pthread_attr_r *attr,
  void * (*start_routine) (void *), void *arg);

Este servicio permite crear un nuevo *thread* que ejecuta una determinada función. El primer argumento de la función apunta al identificador del *thread* que se crea, este identificador viene determinado por el tipo `pthread_t`. El segundo argumento especifica los atributos de ejecución asociados al nuevo *thread*. Si el valor de este segundo argumento es NULL, se utilizarán los atributos por defecto, que incluyen la creación del proceso como no independiente. El tercer argumento indica el nombre de la función a ejecutar cuando el *thread* comienza su ejecución. Esta función requiere un solo parámetro que se especifica con el cuarto argumento, `arg`.

- pthread_t **pthread_self**(void)

Un *thread* puede averiguar su identificador invocando este servicio.

Terminación de *threads*

Los servicios relacionados con la terminación de *threads* son los siguientes:

- int **pthread_join**(pthread thid, void **value);

Este servicio es similar al `wait`, pero a diferencia de éste, es necesario especificar el *thread* por el que se quiere esperar, que no tiene por qué ser un *thread* hijo. Suspende la ejecución del *thread* llamante hasta que el *thread* con identificador `thid` finalice su ejecución. El servicio devuelve en el segundo argumento el valor que pasa el *thread* que finaliza su ejecución en el servicio `pthread_exit`, que se verá a continuación. Únicamente se puede solicitar el servicio `pthread_join` sobre *threads* creados como no independientes.

- int **pthread_exit**(void *value)

Es análogo al servicio `exit` sobre procesos. Incluye un puntero a una estructura que es devuelta al *thread* que ha ejecutado el correspondiente servicio `pthread_join`, lo que es mucho más genérico que el argumento que permite el servicio `wait`.

La figura 3.53 muestra una jerarquía de *threads*. Se supone que el *thread* A es el primario, por lo que corresponde a la ejecución del `main`. Los procesos B, C y D se han creado mediante `pthread_create` y ejecutan respectivamente los procedimientos b(), c() y d(). El *thread* D se ha creado como «no independiente», por lo que otro *thread* puede hacer una operación *join* sobre él. La figura muestra que el *thread* C hace una operación *join* sobre el D, por lo que se queda bloqueado hasta que termine.

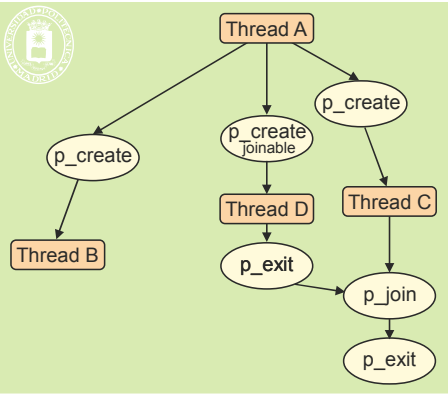


Figura 3.53 *Ejemplo de jerarquía de threads.*

El programa 3.14 crea dos *threads* que ejecutan la función `func`. Una vez creados se espera su finalización con el servicio `pthread_join`.

Programa 3.14 Programa que crea dos *threads* no independientes.

```
#include <pthread.h>
#include <stdio.h>

void func(void)
{

    printf("Thread %d\n", pthread_self());
    pthread_exit(0);
}

int main(void)
```

```
{
    pthread_t th1, th2;

    /* se crean dos threads con atributos por defecto */
    pthread_create(&th1, NULL, func, NULL);
    pthread_create(&th2, NULL, func, NULL);

    printf("El thread principal continúa ejecutando\n");

    /* se espera su terminación */
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);

    return 0;
}
```

El programa 3.15 crea diez *threads* independientes, que liberan sus recursos cuando finalizan (se han creado con el atributo `PTHREAD_CREATE_DETACHED`). En este caso, no se puede esperar la terminación de los *threads*, por lo que el *thread* principal que ejecuta el código de la función `main` debe continuar su ejecución en paralelo con ellos. Para evitar que el *thread* principal finalice la ejecución de la función `main`, lo que supone la ejecución del servicio `exit` y, por tanto, la finalización de todo el proceso (aclaración 3.8), junto con todos los *threads*, el *thread* principal suspende su ejecución durante cinco segundos para dar tiempo a la creación y destrucción de los *threads* que se han creado.

Aclaración 3.8. La ejecución de `exit` supone la finalización del proceso que lo solicita. Esto supone, por tanto, la finalización de todos sus *threads*, ya que éstos sólo tienen sentido dentro del contexto de un proceso.

Programa 3.15 Programa que crea 10 *threads* independientes.

```
#include <pthread.h>
#include <stdio.h>

#define MAX_THREADS 10

void func(void)
{
    printf("Thread %d\n", pthread_self());
    pthread_exit(0);
}

int main(void)
{
    int j;
    pthread_attr_t attr;
    pthread_t thid[MAX_THREADS];

    /* Se inician los atributos y se marcan como independientes */
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
    for(j = 0; j < MAX_THREADS; j++)
        pthread_create(&thid[j], &attr, func, NULL);

    /* El thread principal debe esperar la finalización de los */
    /* threads que ha creado para lo cual se suspende durante */
    /* un cierto tiempo, esperando su finalización */
    sleep(5);
    return 0;
}
```

El programa 3.16 crea un *thread* por cada número que se introduce. Cada *thread* ejecuta el código de la función `imprimir`.

Programa 3.16 Programa que crea un *thread* por cada número introducido.

```
#include <pthread.h>
#include <stdio.h>

#define MAX_THREADS 10

void imprimir(int *n)
{
   printf("Thread %d %d\n", pthread_self(), *n);
   pthread_exit(0);
}

int main(void)
{
   pthread_attr_t attr;
   pthread_t thid;
   int num;

   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

   while(1) {
      scanf("%d", &num); /* espera */
      pthread_create(&thid, &attr, imprimimr, &num);
   }
   return 0;
}
```

Este programa, tal y como se ha planteado, presenta un problema ya que éste falla si el número que se pasa a un *thread* es sobrescrito por el *thread* principal en la siguiente iteración del bucle `while`, antes de que el *thread* que se ha creado lo haya utilizado. Este problema requiere que los *threads* se sincronicen en el acceso a este número. La forma de sincronizar procesos e *threads* se tratará en el capítulo "6 Comunicación y sincronización de procesos".

3.13.3. Servicios UNIX para gestión de señales y temporizadores

En esta sección se presentan los principales servicios que ofrece UNIX para la gestión de señales y temporizadores.

El fichero de cabecera `signal.h` declara la lista de señales posibles que se pueden enviar a los procesos en un sistema. A continuación se presenta una lista de las señales que deben incluir todas las implementaciones. La acción por defecto para todas estas señales es la terminación anormal de proceso.

- SIGABRT: terminación anormal.
- SIGALRM: señal de fin de temporización.
- SIGFPE: operación aritmética errónea.
- SIGHUP: desconexión del terminal de control.
- SIGILL: instrucción máquina inválida.
- SIGINT: señal de atención interactiva.
- SIGKILL: señal que mata al proceso (no se puede ignorar ni armar).
- SIGPIPE: escritura en una tubería sin lectores.
- SIGQUIT: señal de terminación interactiva.
- SIGSEGV: referencia a memoria inválida.
- SIGTERM: señal de terminación.
- SIGUSR1: señal definida por la aplicación.
- SIGUSR2: señal definida por la aplicación.

Si una implementación soporta control de trabajos, entonces también debe dar soporte, entre otras, a las siguientes señales:

- SIGCHLD: indica la terminación del proceso hijo. La acción por defecto es ignorarla.
- SICONT: continuar si está detenido el proceso.
- SIGSTOP: señal de bloqueo (no se puede armar ni ignorar).

Los nombres de las señales que hemos utilizado se refieren a macros C que producen el número identificador de cada una de ellas.

A continuación, se describen los principales servicios UNIX relativos a las señales. Estos servicios se han agrupado de acuerdo a las siguientes categorías:

- Conjuntos de señales.
- Envío de señales.

- Armado de una señal.
- Bloqueo de señales.
- Espera de señales.
- Servicios de temporización.

Conjuntos de señales

Como se ha indicado anteriormente, existe una serie de señales que un proceso puede recibir durante su ejecución. Un proceso puede realizar operaciones sobre grupos o conjuntos de señales. Estas operaciones sobre grupos de señales utilizan conjuntos de señal de tipo `sigset_t` y son las que se detallan seguidamente. Todas ellas menos `sigismember` devuelven 0 si tienen éxito o -1 si hay un error.

◙ int **sigemptyset**(sigset_t *set);

Inicia un conjunto de señales de modo que no contenga ninguna señal.

◙ int **sigfillset**(sigset_t *set);

Inicia un conjunto de señales con todas las señales disponibles en el sistema.

◙ int **sigaddset**(sigset_t *set, int signo);

Añade al conjunto `set`, la señal con número `signo`.

◙ int **sigdelset**(sigset_t *set, int signo);

Elimina del conjunto `set` la señal con número `signo`.

◙ int **sigismember**(sigset_t *set, int signo);

Permite determinar si una señal pertenece a un conjunto de señales, devolviendo 1 si la señal `signo` se encuentra en el conjunto de señales `set`. En caso contrario devuelve 0.

Envío de señales

Algunas señales como SIGSEGV o SIGiBUS las genera el sistema operativo cuando ocurren ciertos errores. Otras señales se envían de unos procesos a otros utilizando el servicio `kill`.

◙ int **kill**(pid_t pid, int sig);

Envía la señal `sig` al proceso o grupo de procesos especificado por `pid`. Para que un proceso pueda enviar una señal a otro proceso designado por `pid`, el identificador de usuario efectivo o real del proceso que envía la señal debe coincidir con el identificador real o efectivo del proceso que la recibe, a no ser que el proceso que envía la señal tenga los privilegios adecuados, por ejemplo, es un proceso ejecutado por el superusuario.

Si `pid` es mayor que cero, la señal se enviará al proceso con identificador de proceso igual a `pid`. Si `pid` es cero, la señal se enviará a todos los procesos cuyo identificador de grupo sea igual al identificador de grupo del proceso que envía la señal. Si `pid` es negativo, pero distinto de -1, la señal será enviada a todos los procesos cuyo identificador de grupo sea igual al valor absoluto de `pid`. Para `pid` igual a -1, Linux especifica que la señal se envía a todos los procesos para los cuales tiene permiso el proceso que solicita el servicio, menos al proceso 1 (init).

Armado de una señal

◙ int **sigaction**(int sig, struct sigaction *act, struct sigaction *oact);

Este servicio tiene tres argumentos: el número de señal para la que se quiere establecer el manejador, un puntero a una estructura de tipo `struct sigaction`, para establecer el nuevo manejador, y un puntero a una estructura del mismo tipo, que almacena información sobre el manejador establecido anteriormente.

La estructura `sigaction`, definida en el fichero de cabecera `signal.h`, está formada por los siguientes campos:

```
struct sigaction {
    void (*sa_handler)();  /* Manejador para la señal */
    sigset_t sa_mask;      /* Señales bloqueadas durante la ejecución del manejador */
    int sa_flags;          /* opciones especiales */
};
```

El primer argumento indica la acción a ejecutar cuando se reciba la señal. Su valor puede ser:

- `SIG_DFL`: indica que se lleve a cabo la acción por defecto que, para la mayoría de las señales (pero no para todas), consiste en matar al proceso.
- `SIG_IGN`: especifica que la señal deberá ser ignorada cuando se reciba.
- Una función que devuelve un valor de tipo `void` y que acepta como argumento un número entero. Cuando el sistema operativo envía una señal a un proceso, coloca como argumento del manejador el número de la señal que se ha enviado. Esto permite asociar un mismo manejador para diferentes señales, de tal manera que el manejador realizará una u otra acción en función del valor pasado al mismo y que identifica a la señal que ha sido generada.

Si el valor del tercer argumento es distinto de NULL, la acción previamente asociada con la señal será almacenada en la posición apuntada por oact. El servicio devuelve 0 en caso de éxito o -1 si hubo algún error.

El siguiente fragmento de código hace que el proceso que lo ejecute ignore la señal SIGINT que se genera cuando se pulsa CTLR-C.

```
struct sigaction act;

act.sa_handler = SIG_IGN;      /* ignorar la señal */
act.sa_flags = 0;              /* ninguna acción especial */
/* Se inicia el conjunto de señales a bloquear cuando se reciba la señal */
sigemptyset(&act.sa_mask);
sigaction(SIGINT, &act, NULL);
```

Máscara de señales

La máscara de señales de un proceso define un conjunto de señales que serán bloqueadas. Bloquear una señal es distinto a ignorarla. Cuando un proceso bloquea una señal, ésta no será enviada al proceso hasta que se desbloquee o se ignore. Si el proceso ignora la señal, ésta simplemente se deshecha. Los servicios asociados con la máscara de señales de un proceso se describen a continuación.

◙ int **sigprocmask**(int how, sigset_t *set, sigset_t *oset);

Permite modificar o examinar la máscara de señales de un proceso. Con este servicio se puede bloquear un conjunto de señales de tal manera que su envío será congelado hasta que se desbloqueen o se ignoren.

El valor del argumento how indica el tipo de cambio sobre el conjunto de señales set. Los posibles valores de este argumento se encuentran definidos en el fichero de cabecera signal.h y son los siguientes:

- SIG_BLK: añade un conjunto de señales a la máscara de señales del proceso.
- SIG_UNBLOCK: elimina de la máscara de señales de un proceso las señales que se encuentran en el conjunto set.
- SIG_SETMASK: crea la nueva máscara de señales de un proceso con el conjunto indicado.

Si el segundo argumento del servicio es NULL, el tercero proporcionará la máscara de señales del proceso que se está utilizando sin ninguna modificación. Si el valor del tercer argumento es distinto de NULL, la máscara anterior se almacenará en oset.

El siguiente fragmento de código bloquea la recepción de todas las señales.

```
sigset_t mask;


sigfillset(&mask);
sigprocmask(SIG_SETMASK, &mask, NULL);
```

Si se quiere, a continuación, desbloquear la señal SIGSEGV, se deberá ejecutar el siguiente fragmento de código:

```
sigset_t mask;


sigemptyset(&mask);
sigaddset(&mask, SIGSEGV);
sigprocmask(SIG_UNBLOCK, &mask, NULL);
```

◙ int **sigpending**(sigset_t *set);

Devuelve el conjunto de señales bloqueadas que se encuentran pendientes de entrega al proceso. El servicio almacena en set el conjunto de señales bloqueadas pendientes de entrega.

Espera de señales

Cuando se quiere esperar la recepción de alguna señal, se utiliza el servicio pause.

◙ int **pause**(void);

Este servicio bloquea al proceso que lo invoca hasta que llegue una señal. No permite especificar el tipo de señal por la que se espera. Dicho de otro modo, sólo la llegada de cualquier señal no ignorada ni enmascarada sacará al proceso del estado de bloqueo.

Servicios de temporización

En esta sección se describen los servicios relativos a los temporizadores.

◙ unsigned int **alarm**(unsigned int seconds);

Para activar un temporizador se debe utilizar el servicio `alarm`, que envía al proceso la señal `SIGALRM` después de pasados el número de segundos especificados en el argumento `seconds`. Si `seconds` es igual a cero se cancelará cualquier petición realizada anteriormente.

El programa 3.17 ejecuta la función `tratar_alarma` cada tres segundos. Para ello, arma un manejador para la señal `SIGALRM` mediante el servicio `sigaction`. A continuación, entra en un bucle infinito en el que activa un temporizador, especificando 3 segundos como argumento del servicio `alarm`. Seguidamente, suspende su ejecución, mediante el servicio `pause`, hasta que se reciba una señal, en concreto la señal `SIGALRM`.

Durante la ejecución de la función `tratar_alarma`, se bloquea la recepción de la señal `SIGINT`, que se genera cuando se teclea `CTRL-C`.

Programa 3.17 Programa que imprime un mensaje cada 3 segundos.

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void tratar_alarma(int)
{
   printf("Activada\n");
}

int main(void)
{
   struct sigaction act;
   sigset_t mask;

   /* estable el manejador */
   act.sa_handler = tratar_alarma;    /* función a ejecutar */
   act.sa_flags = 0;                  /* ninguna acción específica*/

   /* Se bloquea la señal SIGINT cuando se ejecute la función
      tratar_alarma */
   sigemptyset(&act.sa_mask);
   sigaddset(&act.sa_mask, SIGINT);

   sigaction(SIGALRM, &act, NULL);

   for(;;)
   {
      alarm(3);                       /* se arma el temporizador */
      pause();                        /* se suspende el proceso hasta que se reciba una
                        señal */
   }
   return 0;
}
```

El programa 3.18 muestra un ejemplo en el que un proceso temporiza la ejecución de un proceso hijo. El programa crea un proceso hijo que ejecuta un mandato recibido en la línea de mandatos y espera su finalización. Si el proceso hijo no termina antes de que haya transcurrido una determinada cantidad de tiempo, el padre mata al proceso enviándole una señal mediante el servicio `kill`. La señal que se envía es `SIGKILL`, señal que no se puede ignorar ni capturar.

Programa 3.18 Programa que temporiza la ejecución de un proceso hijo.

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

pid_t pid;

void matar_proceso(int)
{
```

```
    kill(pid, SIGKILL);          /* se envía la señal al hijo */
}

int main(int argc, char *argv[])
{
    int status;
    char **argumentos;
    struct sigaction act;

    argumentos = &argv[1];
    /* Se crea el proceso hijo */
    pid = fork();
    switch(pid)
    {
        case -1:                 /* error del fork() */
            exit(1);
        case 0:                  /* proceso hijo */
            /* El proceso hijo ejecuta el mandato recibido */
            execvp(argumentos[0], argumentos);
            perror("exec");
            exit(1);
        default:                 /* padre */
            /* establece el manejador */
            act.sa_handler = matar_proceso;  /*función a ejecutar*/
            act.sa_flags = 0; /                 * ninguna acción específica */
            sigemptyset(&act.sa_mask);
            sigaction(SIGALRM, &act, NULL);
            alarm(5);

            /* Espera al proceso hijo */
            wait(&status);
    }

    return 0;
}
```

En el programa anterior, el proceso padre, una vez que ha armado el temporizador, se bloquea esperando la finalización del proceso hijo, mediante un servicio `wait`. Si la señal SIGALRM se recibe antes de que el proceso haya finalizado, el padre ejecutará la acción asociada a la recepción de esta señal. Esta acción se corresponde con la función `matar_proceso`, que es la que se encarga de enviar al proceso hijo la señal `SIGKILL`. Cuando el proceso hijo recibe esta señal, se finaliza su ejecución.

- int **sleep**(unsigned int seconds)

El proceso se suspende durante un número de segundos pasado como argumento. El proceso despierta y retorna cuando ha transcurrido el tiempo establecido o cuando se recibe una señal.

3.13.4. Servicios UNIX de planificación

Para la modificación de la prioridad de un proceso, se usa la llamada al sistema `nice`.

- int **nice**(int inc);

Este servicio permite a un proceso cambiar su prioridad base. El parámetro `inc` es interpretado como un valor que se suma a la prioridad base del proceso. Dado que el grado de prioridad es inversamente proporcional al valor de la prioridad, un valor positivo en este parámetro implica una disminución de la prioridad: tanto menor será la prioridad resultante como mayor sea ese valor. Un valor negativo en ese parámetro conlleva un aumento de la prioridad, pero esa operación sólo la puede realizar un súper-usuario. De esta restricción surge el curioso nombre de esta llamada: cuando los usuarios normales utilizan este servicio (o el mandato del mismo nombre) es para disminuir la prioridad de sus procesos, mostrándose, por tanto, *agradables* con respecto a los otros usuarios. Como podrá imaginar el lector, se trata de una llamada que no se usa con mucha frecuencia por parte de los usuarios normales. El servicio `setpriority`, presente en algunas versiones de UNIX, tiene un comportamiento similar.

A continuación, se presenta el servicio que permite que un *thread* pueda ceder voluntariamente el uso del procesador, sin bloquearse, pasando simplemente a la cola de listos.

- int **pthread_yield**(void);

Este servicio causa que el *thread* que lo invoque ceda el uso del procesador. En Linux existe la llamada `sched_yield`, que permite que un proceso realice esa misma operación.

Por último, revisaremos las funciones que tienen que ver con el control de la afinidad estricta de un proceso. Dado que este tipo de funciones no está estandarizado dentro del mundo de UNIX, se presentan las funciones específicas de Linux.

- int **sched_setaffinity**(pid_t pid, unsigned int longitud, cpu_set_t *máscara);

Este servicio permite establecer la afinidad estricta del proceso identificado por `pid`, es decir, el conjunto de procesadores en los que podrá ejecutar. Este conjunto se especifica mediante el parámetro `máscara`, cuyo tamaño se indica en el parámetro `longitud`. El servicio `sched_getaffinity` permite obtener la afinidad estricta actual de un proceso. Existe un conjunto de macros que permiten manipular esa máscara, como se verá en un ejemplo posterior.

A continuación, se presenta el programa 3.19 que muestra el uso del servicio `nice` para cambiar la prioridad de un proceso. Se trata de un programa en el que un proceso padre y un hijo escriben un número muy elevado de mensajes en la salida estándar. Dado que escriben el mismo número de mensajes, deberían terminar en un tiempo relativamente cercano. El usuario puede controlar la prioridad de ejecución del proceso hijo mediante el argumento que recibe el programa. Probando con distintos valores, se puede ver el efecto del servicio `nice`. Con un valor de cero, el proceso hijo mantiene su prioridad y termina, aproximadamente, al mismo tiempo que el padre. Según van usándose valores positivos mayores (recuerde que sólo podrá probar con valores negativos si es el súper-usuario), el hijo irá alargando progresivamente su tiempo de ejecución, al tener menor prioridad.

Programa 3.19 Programa que muestra el uso del servicio `nice`.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

/* bucle de trabajo de los procesos */
void funcion(char *proceso) {
    int i;
    static int const num_iter=1000000;

    for (i=0; i<num_iter; i++)
        printf("proceso %s iteración %d\n", proceso, i);
}

int main(int argc, char *argv[]) {
    int prio_hijo;

    if (argc!=2) {
        fprintf(stderr, "Uso: %s prioridad_hijo\n", argv[0]);
        return 1;
    }
    /* elimina el buffering en stdout */
    setbuf(stdout, NULL);

    /* crea un proceso hijo */
    if (fork() == 0) {
        prio_hijo=atoi(argv[1]);

        /* le cambia la prioridad */
        if (nice(prio_hijo) < 0) {
            perror("Error cambiando prioridad");
            return 1;
        }
        funcion("hijo");
    }
    else {
        funcion("padre");
        wait(NULL);
    }
    return 0;
}
```

El segundo ejemplo, que se corresponde con el programa 3.20, muestra el uso de los servicios `sched_getaffinity` y `sched_setaffinity` de Linux. El programa lanza un proceso hijo por cada uno de los procesadores existentes, fijando la afinidad de cada proceso hijo a un único procesador. En la salida generada por este programa cuando se ejecuta en un multiprocesador se puede apreciar que cada proceso ejecuta siempre en el mismo procesador. Observe el uso de las macros que permiten manejar la máscara que define un conjunto de procesadores. Así, por ejemplo, para incluir un determinado procesador en una máscara, se usa la macro `CPU_SET`.

Programa 3.20 Programa que modifica la afinidad estricta de los procesos.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
#define    USE_GNU
#include <sched.h>
#include <stdlib.h>

/* bucle de trabajo de los procesos */
void funcion(int procesador) {
   int i;
   static int const num_iter=10000000;

   for (i=0; i<num_iter; i++)
      printf("proceso %d ejecutando en procesador %d\n",
         getpid(), procesador);
}

int main(int argc, char *argv[]) {
   int i;
   int num_procesadores=0;
   cpu_set_t procesadores;
   cpu_set_t mi_procesador;

   /* elimina el buffering en stdout */
   setbuf(stdout, NULL);

   /* obtiene qué procesadores están disponibles para este proceso */
   sched_getaffinity(0, sizeof(procesadores), &procesadores);

   /* Bucle que crea un hijo por cada procesador disponible */
   for (i=0; i<CPU_SETSIZE; i++)

      /* ¿existe el procesador i? */
      if (CPU_ISSET(i, &procesadores)) {
         num_procesadores++;
         /* crea proceso hijo */
         if (fork() == 0) {
            CPU_ZERO(&mi_procesador);
            CPU_SET(i, &mi_procesador);
            /* asigna el proceso hijo al procesador i */
         if (sched_setaffinity(0, sizeof(mi_procesador),
               &mi_procesador) == -1) {
            perror("Error en sched_setaffinity");
            return 1;
         }
         else {
            /* el proceso se pone a trabajar */
            funcion(i);
            return 0;
         }
         }
      }

/* espera a que terminen todos los hijos */
   for (i=0; i<num_procesadores; i++)
      wait(NULL);
```

```
    return 0;
}
```

3.13.5. Servicios Windows para la gestión de procesos

En Windows cada proceso contiene uno o más *threads*. Como se indicó anteriormente, en Windows el *thread* es la unidad básica de ejecución. Los procesos en Windows se diferencian de los de UNIX, en que Windows no mantiene ninguna relación padre-hijo. Por conveniencia, sin embargo, en el texto se asumirá que un proceso padre crea a un proceso hijo. Los servicios que ofrece Windows se han agrupado, al igual que en UNIX, en las siguientes categorías:

- Identificación de procesos.
- El entorno de un proceso.
- Creación de procesos.
- Terminación de procesos.

Identificación de procesos

En Windows, los procesos se identifican mediante identificadores de procesos y manejadores. Un identificador de proceso es un objeto de tipo entero que identifica de forma única a un proceso en el sistema. El manejador se utiliza para identificar al proceso en todas las funciones que realizan operaciones sobre el proceso.

- HANDLE GetCurrentProcess(VOID);
- DWORD GetCurrentProcessId(VOID);

Estos dos servicios permiten obtener la identificación del propio proceso. La primera devuelve el manejador del proceso que solicita el servicio y la segunda su identificador de proceso.

- HANDLE **OpenProcess**(DWORD fdwAccess, BOOL fInherit, DWORD IdProcess);

Permite obtener el manejador de un proceso conocido su identificador. El primer argumento especifica el modo de acceso al objeto que identifica al proceso con identificador `IdProcess`. Algunos de los posibles valores para este argumento son:

- `PROCESS_ALL_ACCESS`: especifica todos los modos de acceso al objeto.
- `SYNCHRONIZE`: permite que el proceso que obtiene el manejador pueda esperar la terminación del proceso con identificador `IdProcess`.
- `PROCESS_TERMINATE`: permite al proceso que obtiene el manejador finalizar la ejecución del proceso con identificador `IdProcess`.
- `PROCESS_QUERY_INFORMATION`: el manejador se puede utilizar para obtener información sobre el proceso.

El argumento `fInherit` especifica si el manejador devuelto por el servicio puede ser heredado por los nuevos procesos creados por el que ejecuta el servicio. Si su valor es TRUE, el manejador se puede heredar.

El servicio devuelve el manejador del proceso en caso de éxito o NULL si se produjo algún error.

El entorno de un proceso

Un proceso recibe su entorno en su creación (mediante el servicio `CreateProcess` descrito en la siguiente sección).

- DWORD **GetEnvironmentVariable**(LPCTSTR lpszName, LPTSTR lpszValue, DWORD cchValue);

Este servicio obtiene en `lpszValue` el valor de la variable de entorno con nombre `lpszName`. El argumento `cchValue` especifica la longitud del *buffer* en memoria al que apunta `lpszValue`. El servicio devuelve la longitud de la cadena en la que se almacena el valor de la variable (`lpszValue`) o 0 si hubo algún error.

- BOOL **SetEnvironmentVariable**(LPCTSTR lpszName, LPTSTR lpzsValue);

`SetEnvironmentVariable` permite modificador el valor de una variable de entorno. Devuelve TRUE si se ejecutó con éxito.

- LPVOID GetEnvironmentStrings(VOID);

Permite obtener un puntero al comienzo del bloque en el que se almacenan las variables de entorno. El programa 3.21 ilustra el uso de este servicio para imprimir la lista de variables de entorno de un proceso.

Programa 3.21 Programa que lista las variables de entorno de un proceso en Windows.

```
#include <windows.h>
#include <stdio.h>

int main(void)
```

```
{
    char *lpszVar;
    void *lpvEnv;

    lpvEnv = GetEnvironmentStrings();
    if (lpvEnv == NULL) {
        printf("Error al acceder al entorno\n");
        exit(1);
    }

    /* las variables de entorno se encuentran separadas por un NULL */
    /* el bloque de variables de entorno termina también en NULL */
    for (lpszVar = (char *) lpvEnv; lpszVar != NULL; lpszVar++) {
        while (*lpszVar)
            putchar(*lpszVar++);
        putchar("\n");
    }
    return 0;
}
```

Creación de procesos

En Windows, los procesos se crean mediante el servicio `CreateProcess`, que es similar a la combinación `fork`-`exec` de UNIX. Windows no permite a un proceso cambiar su imagen de memoria y ejecutar otro programa distinto.

```
BOOL CreateProcess (
    LPCTSTR lpszImageName,
    LPTSTR lpszCommandLine,
    LPSECURITY_ATTRIBUTES lpsaProcess,
    LPSECURITY_ATTRIBUTES lpsaThread,
    BOOL fInheritHandles,
    DWORD fdwCreate,
    LPVOID lpvEnvironment,
    LPCTSTR lpszCurdir,
    LPSTARTUPINFO lpsiStartInfo,
    LPPROCESS_INFORMATION lppiProcInfo);
```

El servicio crea un nuevo proceso y su *thread* principal. El nuevo proceso ejecuta el fichero ejecutable especificado en `lpszImageName`. Esta cadena puede especificar el nombre de un fichero con camino absoluto o relativo, pero el servicio no utilizará el camino de búsqueda. Si `lpszImageName` es NULL, se utilizará como nombre de fichero ejecutable la primera cadena delimitada por blancos del argumento `lpszCommandLine`.

El argumento `lpszCommandLine` especifica la línea de mandatos a ejecutar, incluyendo el nombre del programa a ejecutar. Si su valor es NULL, el servicio utilizará la cadena apuntada por `lpszImageName` como línea de mandatos. El nuevo proceso puede acceder a la línea de mandatos utilizando los argumentos `argc` y `argv` de la función `main` del lenguaje C.

El argumento `lpsaProcess` determina si el manejador asociado al proceso creado y devuelto por el servicio puede ser heredado por otros procesos hijos. Si es NULL, el manejador no puede heredarse. Lo mismo se aplica para el argumento `lpsaThread`, pero relativo al manejador del *thread* principal devuelto por el servicio.

El argumento `fInheritHandles` indica si el nuevo proceso hereda los manejadores que mantiene el proceso que solicita el servicio. Si su valor es TRUE el nuevo proceso hereda todos los manejadores que tenga abiertos el proceso padre y con los mismos privilegios de acceso.

El argumento `fdwCreate` puede combinar varios valores que determinan la prioridad y la creación del nuevo proceso. Algunos de estos valores son:

- `CREATE_SUSPEND`: el *thread* principal del proceso se crea en estado suspendido y sólo se ejecutará cuando se llame al servicio `ResumeThread` descrito más adelante.
- `DETACHED_PROCESS`: para procesos con consola, indica que el nuevo proceso no tenga acceso a la consola del proceso padre. Este valor no puede utilizarse con el siguiente.
- `CREATE_NEW_CONSOLE`: el nuevo proceso tendrá una nueva consola asociada y no heredará la del padre. Este valor no puede utilizarse con el anterior.
- `NORMAL_PRIORITY_CLASS`: el proceso creado no tiene necesidades especiales de planificación.
- `HIGH_PRIORITY_CLASS`: el proceso creado tiene una alta prioridad de planificación.
- `IDLE_PRIORITY_CLASS`: especifica que los *threads* del proceso sólo ejecuten cuando no haya ningún otro proceso ejecutando en el sistema.
- `REALTIME_PRIORITY_CLASS`: el proceso creado tiene la mayor prioridad posible. Los *threads* del nuevo proceso serán capaces de expulsar a cualquier otro proceso, incluyendo los procesos del sistema operativo.

El argumento `lpEnvironment` apunta al bloque del entorno del nuevo proceso. Si el valor es NULL, el nuevo proceso obtiene el entorno del proceso que solicita el servicio.

El argumento `lpszCurdir` apunta a una cadena de caracteres que indica el directorio de trabajo para el nuevo proceso. Si el valor de este argumento es NULL, el proceso creado tendrá el mismo directorio que el padre.

El argumento `lpStartupInfo` apunta a una estructura de tipo `STARTPINFO`, que especifica la apariencia de la ventana asociada al nuevo proceso. Para especificar los manejadores para la entrada, salida y error estándar deben utilizarse los campos `hStdIn`, `hStdOut` y `hStdErr`. En este caso, el campo `dwFlags` de esta estructura debe contener el valor `STARTF_USESTDHANDLES`.

Por último, en el argumento `lpProcessInformation`, puntero a una estructura de tipo `PROCESS_INFORMATION`, se almacenará información sobre el nuevo proceso creado. Esta estructura tiene la siguiente definición:

```
typedef struct PROCESS_INFORMATION {
    HANDLE hProcess;
    HANDLE hThread;
    DWORD dwProcessId;
    DWORD dwThreadId;
} PROCESS_INFORMATION;
```

En los campos `hProcess` y `hThread` se almacenan los manejadores del nuevo proceso y de su *thread* principal. En `dwProcessId` se almacena el identificador del nuevo proceso y en `dwThreadId` el identificador del *thread* principal del proceso creado.

El programa 3.22 crea un proceso que ejecuta un mandato pasado en la línea de argumentos. El proceso padre no espera a que finalice, es decir, es nuevo proceso se ejecuta en *background*.

Programa 3.22 Programa que crea un proceso que ejecuta la línea de mandatos pasada como argumento.

```
#include <windows.h>
#include <stdio.h>

int main(int argc, LPTSTR argv [])
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    if (!CreateProcess(
            NULL,           /* utiliza la línea de mandatos */
            argv[1],        /* línea de mandatos pasada como argumentos */
            NULL,           /* manejador del proceso no heredable*/
            NULL,           /* manejador del thread no heredable */
            FALSE,          /* no hereda manejadores */
            0,              /* sin flags de creación */
            NULL,           /* utiliza el entorno del proceso */
            NULL,           /* utiliza el directorio de trabajo del padre */
            &si,
            &pi))
    {
        printf ("Error al crear el proceso. Error: %x\n",  GetLastError ());
        ExitProcess(1);
    }

    /* el proceso acaba */
    return 0;
}
```

Terminación de procesos

Un proceso puede finalizar su ejecución de forma voluntaria de tres modos:

- Ejecutando dentro de la función `main` la sentencia `return`.
- Ejecutando el servicio `ExitProcess`.
- Ejecutando la función de la biblioteca de C `exit`, función similar a `ExitProcess`.

◙ VOID **ExitProcess**(UINT nExitCode);

Cierra todos los manejadores abiertos del proceso, especifica el código de salida del proceso y lo termina.

◙ BOOL **GetExitCodeProcess**(HANDLE hProcess, LPDWORD lpdwExitCode);

Devuelve el código de terminación del proceso con manejador `hProcess`. El proceso especificado por `hProcess` debe tener el acceso `PROCESS_QUERY_INFORMATION` (véase `OpenProcess`). Si el proceso todavía no ha terminado, el servicio devuelve en `lpdwExitCode` el valor `STILL_ALIVE`, en caso contrario almacenará en este valor el código de terminación.

- BOOL **TerminateProcess**(HANDLE hProcess, UINT uExitCode);

Este servicio aborta la ejecución del proceso con manejador `hProcess`. El código de terminación para el proceso vendrá dado por el argumento `uExitCode`. El servicio devuelve TRUE si se ejecuta con éxito.

Esperar por la finalización de un proceso

En Windows un proceso puede esperar la terminación de cualquier otro proceso siempre que tenga permisos para ello y disponga del manejador correspondiente. Para ello, se utilizan las funciones de espera de propósito general, las cuales también se tratarán en el capítulo "6 Comunicación y sincronización de procesos". Estas funciones son:

- DWORD **WaitForSingleObject**(HANDLE hObject, DWORD dwTimeOut);

Bloquea al proceso hasta que el proceso con manejador `hObject` finalice su ejecución. El argumento `dwTimeOut` especifica el tiempo máximo de bloqueo expresado en milisegundos. Un valor de 0 hace que el servicio vuelva inmediatamente después de comprobar si el proceso finalizó la ejecución. Si el valor es `INFINITE`, el servicio bloquea al proceso hasta que el proceso acabe su ejecución.

- DWORD **WaitForMultipleObjects**(DWORD cObjects, LPHANDLE lphObjects,
  BOOL fWaitAll. DWORD dwTimeOut);

Permite esperar la terminación de varios procesos. El argumento `cObjects` especifica el número de procesos (el tamaño del vector `lphObjects`) por los que se desea esperar. El argumento `lphObjects` es un vector con los manejadores de los procesos sobre los que se quiere esperar. Si el argumento `fWaitAll` es TRUE, entonces el servicio debe esperar por todos los procesos, en caso contrario el servicio devuelve tan pronto como un proceso haya acabado. El argumento `dwTimeOut` tiene el significado descrito anteriormente.

Estas funciones, aplicadas a procesos, pueden devolver los siguientes valores:

- `WAIT_OBJECT_0`: indica que el proceso terminó en el caso del servicio `WaitForSingleObject`, o todos los procesos terminaron si en `WaitForMultipleObjects` el argumento `fWaitAll` es TRUE.
- `WAIT_OBJECT_0+n`, donde 0 <= n <= `cObjects`. Restando este valor de `WAIT_OBJECT_0` se puede determinar el número de procesos que han acabado.
- `WAIT_TIMEOUT`: indica que el tiempo de espera expiró antes de que algún proceso acabara.

Las funciones devuelven 0XFFFFFFFF en caso de error.

El programa 3.23 crea un proceso que ejecuta el programa pasado en la línea de mandatos. El programa espera a continuación a que el proceso hijo termine imprimiendo su código de salida.

Programa 3.23 Programa que crea un proceso que ejecuta la línea de mandatos pasada como argumento y espera por él.

```
#include <windows.h>
#include <stdio.h>

int main(int argc, LPTSTR argv [])
{
   STARTUPINFO si;
   PROCESS_INFORMATION pi;
   DWORD code;

   if (!CreateProcess(
         NULL,        /* utiliza la línea de mandatos */
         argv[1],     /* línea de mandatos pasada como argumentos */
         NULL,        /* manejador del proceso no heredable*/
         NULL,        /* manejador del thread no heredable */
         FALSE,       /* no hereda manejadores */
         0,           /* sin flags de creación */
         NULL,        /* utiliza el entorno del proceso */
         NULL,        /* utiliza el directorio de trabajo del padre */
         &si,
         &pi))
   {
      printf ("Error al crear el proceso. Error: %x\n", GetLastError ());
      ExitProcess(1);
   }
```

```
    /* Espera a que el proceso creado acabe */
    WaitForSingleObject(pi.hProcess, INFINITE);
    /* Imprime el código de salida del proceso hijo */
    if (!GetExitCodeProcess(pi.hProcess, &code) {
        printf ("Error al acceder al código. Error: %x\n", GetLastError ());
        ExitProcess(2);
    }
    printf("codigo de salida del proceso %d\n", code);
    ExitProcess(0);
}
```

3.13.6. Servicios Windows para la gestión de *threads*

Los *threads* son la unidad básica de ejecución en Windows. Los servicios de Windows para la gestión de *threads* pueden agruparse en las siguientes categorías:

- Identificación de *threads*.
- Creación de *threads*.
- Terminación de *threads*.

Identificación de *threads*

En Windows, los *threads*, al igual que los procesos, se identifican mediante identificadores de *threads* y manejadores. Estos presentan las mismas características que los identificadores y manejadores para procesos, ya descritos en la sección "3.13.5 Servicios Windows para la gestión de procesos".

◙ HANDLE GetCurrentThread(VOID);

Devuelve el manejador del *thread* que ejecuta el servicio.

◙ DWORD GetCurrentThreadId(VOID);

Devuelve el identificador de *thread*.

◙ HANDLE **OpenThread**(DWORD fdwAccess, BOOL fInherit, DWORD Idthread);

Permite obtener el manejador del *thread* dado su identificador. El primer argumento especifica el modo de acceso al objeto que identifica al *thread* con identificador `Idthread`. Algunos de los posibles valores para este argumento son:

- `THREAD_ALL_ACCESS`: especifica todos los modos de acceso al objeto.
- `SYNCHRONIZE`: permite que el proceso que obtiene el manejador pueda esperar la terminación del *thread* con identificador `IdThread`.
- `THREAD_TERMINATE`: permite al proceso que obtiene el manejador finalizar la ejecución del *thread* con identificador `IdThread`.
- `THREAD_QUERY_INFORMATION`: el manejador se puede utilizar para obtener información sobre el *thread*.

El argumento `fInherit` especifica si el manejador devuelto por el servicio puede ser heredado por los nuevos procesos creados por el que solicita el servicio. Si su valor es TRUE, el manejador se puede heredar.

El servicio devuelve el manejador del *thread* en caso éxito o NULL si se produjo algún error.

Creación de *threads*

En Windows, los *threads* se crean mediante el servicio `CreateThread`.

```
◙ BOOL CreateThread (
        LPSECURITY_ATTRIBUTES lpsa,
        DWORD cbStack,
        LPTHREAD_START_ROUTINE lpStartAddr;
        LPVOID lpvThreadParam,
        DWORD fdwCreate,
        LPDWORD lpIdThread);
```

Este servicio crea un nuevo *thread*. El argumento `lpsa` contiene la estructura con los atributos de seguridad asociados al nuevo *thread*. Su significado es el mismo que el utilizado en el servicio `CreateProcess`. El argumento `cbStack` especifica el tamaño de la pila asociada al *thread*. Un valor de 0 especifica el tamaño por defecto (1 MiB). `lpStartAddr` apunta a la función a ser ejecutada por el *thread*. Esta función debe ajustarse al siguiente prototipo: `DWORD WINAPI MiFuncion(LPVOID);` Esta función acepta un argumento puntero a tipo desconocido y devuelve un valor de 32 bits. El *thread* puede interpretar el argumento como un DWORD o un puntero. El argumento `lpvThreadParam` almacena el argumento pasado al *thread*. Si `fdwCreate` es 0, el *thread* ejecuta in-

mediatamente después de su creación. Si su valor es `CREATE_SUSPENDED`, el *thread* se crea en estado suspendido. En `lpIdThread` se almacena el identificador del nuevo *thread* creado.

El servicio `CreateThread` devuelve el manejador para el nuevo *thread* creado o bien NULL en caso de error.

◙ DWORD **SuspendThread**(HANDLE hThread);

Suspende la ejecución del *thread* `hThread`.

◙ DWORD **ResumeThread**(HANDLE hThread);

Un *thread* suspendido puede ponerse de nuevo en ejecución mediante `ResumeThread`.

Terminación de *threads*

Al igual que con los procesos en Windows, los servicios relacionados con la terminación de *threads* se agrupan en dos categorías: servicios para finalizar la ejecución de un *thread* y servicios para esperar la terminación de un *thread*. Estos últimos son los mismos que los empleados para esperar la terminación de procesos (`WaitForSingleObject` y `WaitForMultipleObjects`) y no se volverán a tratar.

Un *thread* puede finalizar su ejecución de forma voluntaria de dos formas:

- Ejecutando dentro de la función principal del *thread* la sentencia `return`.
- Ejecutando el servicio `ExitThread`.

◙ VOID **ExitThread**(DWORD dwExitCode);

Con este servicio un *thread* finaliza su ejecución especificando su código de salida mediante el argumento `dwExitCode`. Este código puede consultarse con el servicio `GetExitCodeThread`, similar al servicio `GetExitCodeProcess`.

◙ BOOL **TerminateThread**(HANDLE hThread, DWORD dwExitCode);

Un *thread* puede abortar la ejecución de otro *thread* mediante `TerminateThread`, similar al servicio `TerminateProcess`.

3.13.7. Servicios Windows para el manejo de excepciones

Como se vio en la sección "3.5.2 Excepciones", una excepción es un evento que ocurre durante la ejecución de un programa y que requiere la ejecución de un código situado fuera del flujo normal de ejecución. Windows ofrece un manejo de excepciones estructurado, que permite la gestión de excepciones *software* y *hardware*. Este manejo permite la especificación de un bloque de código o manejador de excepción a ser ejecutado cuando se produce la excepción.

El manejo de excepciones en Windows necesita del soporte del compilador para llevarla a cabo. El compilador de C desarrollado por Microsoft ofrece a los programadores dos palabras reservadas que pueden utilizarse para construir manejadores de excepción. Esta son: `__try` y `__except`. La palabra reservada `__try` identifica el bloque de código que se desea proteger de errores. La palabra `__except` identifica el manejador de excepciones.

En las siguientes secciones se van a describir los tipos y códigos de excepción y el uso de un manejador de excepción.

Tipos y códigos de excepción

◙ DWORD GetExceptionCode(VOID);

Permite obtener el código de excepción. Este servicio debe ejecutarse justo después de producirse una excepción. El servicio devuelve el valor asociado a la excepción. Existen muchos tipos de excepciones, algunos de ellos son:

- `EXCEPTION_ACCESS_VIOLATION`: se produce cuando se accede a una dirección de memoria inválida.
- `EXCEPTION_DATATYPE_MISALIGMENT`: se produce cuando se accede a datos no alineados.
- `EXCEPTION_INT_DIVIDE_BY_ZERO`: se genera cuando se divide por cero.
- `EXCEPTION_PRIV_INSTRUCTION`: ejecución de una instrucción ilegal.

Uso de un manejador de excepciones

La estructura básica de un manejador de excepciones es:

```
__try {
   /* bloque de código a proteger */
}
__except (expresión) {
   /* manejador de excepciones */
}
```

El bloque de código encerrado dentro de la sección `__try` representa el código del programa que se quiere proteger. Si ocurre una excepción mientras se está ejecutando este fragmento de código, el sistema operativo transfiere el control al manejador de excepciones. Este manejador se encuentra dentro de la sección `__except`. La expresión asociada a `__except` se evalúa inmediatamente después de producirse la excepción. La expresión debe devolver alguno de los siguientes valores:

- `EXCEPTION_EXECUTE_HANDLER`: el sistema ejecuta el bloque `__except`.
- `EXCEPTION_CONTINUE_SEARCH`: el sistema no hace caso al manejador de excepciones, continuando hasta que encuentra uno.
- `EXCEPTION_CONTINUE_EXECUTION`: el sistema devuelve inmediatamente el control al punto en el que ocurrió la excepción.

El programa 3.24 muestra una versión mejorada de la función de biblioteca `strcpy` del lenguaje C. Esta nueva función detecta la existencia de punteros inválidos devolviendo NULL en tal caso.

Programa 3.24 Versión mejora de `strcpy` utilizando un manejador de excepciones.

```
LPTSTR StrcpySeguro(LPTSTR s1, LPTSTR s2) {
    __try{
        return strcpy(s1, s2);
    }
    __except(GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION ?
    EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) {
        return NULL;
    }
}   LPTSTR StrcpySeguro(LPTSTR s1, LPTSTR s2) {
        try{
        return strcpy(s1, s2);
    }
    __except(GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION ?
    EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) {
        return NULL;
    }
}
```

Si se produce un error durante la ejecución de la función `strcpy`, se elevará una excepción y se pasará a evaluar la expresión asociada al bloque `__except`.

Una forma general de manejar las excepciones que se producen dentro de un bloque `__try`, utilizando el servicio `GetExcepcionCode`, es la que se muestra en Programa.

Cuando se produce una excepción en el Programa, se ejecuta el servicio `GetExceptionCode`, que devuelve el código de excepción producido. Este valor se convierte en el argumento para la función filtrar. La función filtrar analiza el código devuelto y devuelve el valor adecuado para la expresión de `__except`. En el programa 3.25 se muestra que, en el caso de que la excepción se hubiese producido por una división por cero, se devolvería el valor `EXCEPTION_EXECUTE_HANDLER` que indicaría la ejecución del bloque `__except`.

Programa 3.25 Manejo general de excepciones.

```
__try {
    /* bloque de código a proteger */
}
__except (Filtrar(GetExceptionCode())) {
    /* manejador de excepciones */
}

DWORD Filtrar (DWORD Code) {
    switch(Code) {
        ....
        case EXCEPTION_INT_DIVIDE_BY_ZERO :
            return EXCEPTION_EXECUTE_HANDLER;
        ...
    }
}
```

3.13.8. Servicios Windows de gestión de temporizadores

Esta sección describe los servicios de Windows relativos a los temporizadores.

Activación de temporizadores

◙ UINT **SetTimer**(HWND hWnd, UINT nIDEvent, UINT uElapse, TIMERPROC lpTimerFunc);

Permite crear un temporizador. El argumento `hWnd` representa la ventana asociada al temporizador. Su valor es NULL a no ser que el gestor de ventanas esté en uso. El segundo argumento es un identificador de evento distinto de cero. Su valor se ignora si es NULL. El argumento `uElapse` representa el valor del temporizador en milisegundos. Cuando venza el temporizador se ejecutará la función especificada en el cuarto argumento. El servicio devuelve el identificador del temporizador o cero si no puede crear el temporizador.

El prototipo de la función a invocar cuando vence el temporizador es el siguiente:

VOID CALLBACK **TimerFunc**(HWND hWnd, UINT uMsg, UINT idEvent, DWORD dwTime);

Los dos primeros argumentos pueden ignorarse cuando no se está utilizando un gestor de ventanas. El argumento `idEvent` es el mismo identificador de evento proporcionado en el servicio `SetTimer`. El argumento `dwTime` representa el tiempo del sistema en formato UTC (*Coordinated Universal Time*, tiempo universal coordinado).

El servicio `SetTimer` crea un temporizador periódico, es decir, si el valor de `uElapse` es 5 ms, la función `lpTimerFunc` se ejecutará cada 5 ms.

◙ BOOL **KillTimer**(HWND hWnd, UINT uIdEvent);

Permite desactivar un temporizador. El argumento `uidEvent` es el valor devuelto por el servicio `SetTimer`. Este servicio devuelve `TRUE` en caso de éxito, lo que significa que se desactiva el temporizador creado con `SetTimer`.

El programa 3.26 imprime un mensaje por la pantalla cada 10 ms.

Programa 3.26 Programa que imprime un mensaje cada 10 ms.

```
#include <windows.h>
#include <stdio.h>

void Mensaje(HWND hwnd, UINT ms, UINT ide, DWORD time)
{
    printf("Ha vencido el temporizador\n");
}
int main(void)
{
    UINT idEvent = 2;
    UINT intervalo = 10;
    UINT tid;

    tid = SetTimer(NULL, idEvent, intervalo, Mensaje);
    while (TRUE)
        ;
    return 0;
}
```

Suspender la ejecución de un *thread*

◙ VOID **Sleep**(DWORD dwMilliseconds);

El *thread* cede el tiempo de rodaja que le quede y se suspende, al menos, durante el número de milisegundos recibido como argumento. Si el argumento es «0» se pone inmediatamente en estado de listo.

3.13.9. Servicios Windows de planificación

En el caso de Windows, como se explicó previamente, la clase a la que pertenece un proceso viene dada implícitamente por su nivel de prioridad. Por tanto, se usa la misma función tanto para cambiar la prioridad como para cambiar de clase.

◙ BOOL **SetPriorityClass**(HANDLE hproceso, DWORD prioridad);

Este servicio permite cambiar la prioridad del proceso cuyo manejador es `hproceso` de acuerdo con el valor especificado en el parámetro `prioridad`, que puede tomar los valores:

- IDLE_PRIORITY_CLASS: prioridad igual a 4.
- BELOW_NORMAL_PRIORITY_CLASS: prioridad igual a 6.
- NORMAL_PRIORITY_CLASS: prioridad igual a 8.
- ABOVE_NORMAL_PRIORITY_CLASS: prioridad igual a 10.
- HIGH_PRIORITY_CLASS: prioridad igual a 13.
- REALTIME_PRIORITY_CLASS: prioridad igual a 24.

Si se especifica el último valor, se está definiendo implícitamente que el proceso es de la clase de tiempo real, para lo cual se necesitan ciertos privilegios, mientras que los valores restantes se corresponden con procesos normales.

Además de poder controlar la prioridad de los procesos, se puede especificar la prioridad de los *threads* de un proceso, que no se define de forma absoluta sino relativa a la prioridad del proceso.

◙ BOOL **SetThreadPriority**(HANDLE hThread, int prioridad);

Este servicio permite definir la prioridad del *thread* cuyo manejador es `hThread` con respecto a la del proceso al que pertenece, pudiendo especificar en el parámetro `prioridad` los siguientes valores:

- THREAD_PRIORITY_HIGHEST: 2 unidades más que la prioridad base del proceso.
- THREAD_PRIORITY_ABOVE_NORMAL: 1 unidad más que la prioridad base del proceso.
- THREAD_PRIORITY_NORMAL: Igual a la prioridad base del proceso.
- THREAD_PRIORITY_BELOW_NORMAL: 1 unidad menos que la prioridad base del proceso.
- THREAD_PRIORITY_LOWEST: 2 unidades menos que la prioridad base del proceso.
- THREAD_PRIORITY_TIME_CRITICAL: se trata de un valor absoluto igual a 31 si el proceso es de tiempo real o igual a 15 si es un proceso normal.
- THREAD_PRIORITY_IDLE: se trata de un valor absoluto igual a 16 si el proceso es de tiempo real o igual a 1 si es un proceso normal.

En cuanto al servicio para ceder el uso del procesador, la función `Sleep`, presentada en el capítulo "3 Procesos", permite realizar esta labor si se especifica un valor de 0 como argumento.

Por último, se presenta el servicio que permite controlar la afinidad estricta de un proceso.

◙ BOOL **SetProcessAffinityMask**(HANDLE hproceso, DWORD_PTR máscara);

Este servicio permite establecer la afinidad estricta del proceso identificado por el manejador `hProceso`, es decir, el conjunto de procesadores en los que podrán ejecutar sus *threads*. Este conjunto se especifica mediante el parámetro `máscara`. El servicio `GetProcessAffinityMask` permite obtener la afinidad estricta actual de un proceso, mientras que `SetThreadAffinityMask` controla la afinidad de un único *thread*.

A continuación, se presenta el programa 3.27, que es equivalente al programa 3.19, página 130, pero usando los servicios de Windows.

Programa 3.27 Programa que muestra cómo se modifican las prioridades de los *threads*.

```
#include <windows>
#include <process.h>
#include <stdio.h>

/* número de valores de la prioridad de un thread */
#define NUMPRIO 7
/* número de mensajes que escribe el thread y el programa principal */
int const num_iter=1000000;

/* posibles valores de la prioridad de un thread */
int priovec[]= {THREAD_PRIORITY_IDLE, THREAD_PRIORITY_LOWEST,
    THREAD_PRIORITY_BELOW_NORMAL, THREAD_PRIORITY_NORMAL,
    THREAD_PRIORITY_ABOVE_NORMAL, THREAD_PRIORITY_HIGHEST,
    THREAD_PRIORITY_TIME_CRITICAL};

/* prioridad del thread */
int prio_th;

/* bucle de trabajo del thread */
DWORD WINAPI funcion(LPVOID p) {
    HANDLE thread;
    int i;

    thread=GetCurrentThread();
    if (!SetThreadPriority(thread, prio_vec[prio_th])) {
        fprintf(stderr, "Error al cambiar la prioridad del thread\n");
        return 1;
```

```
    }
    for (i=0; i<num_iter; i++)
        printf("thread iteración %d\n", i);

    return 0;
}

int main(int argc, char *argv[]) {
    HANDLE thread;
    DWORD thid;
    int i;

    if (argc!=2) {
        fprintf(stderr, "Uso: %s prioridad_thread (0 mínima; 6 máxima)\n", argv[0]);
        return 1;
    }
    /* calcula la prioridad del thread */
    prio_hijo=atoi(argv[1])%NUMPRIO;

    /* crea un proceso thread */
    /* NOTA: Si tiene problemas por el uso concurrente de la biblioteca de C,
            use  beginthreadex en vez de CreateThread */
    thread = CreateThread(NULL, 0, funcion, NULL, 0, &thid);
    if (thread == NULL) {
        fprintf(stderr, "Error al crear el thread\n");
        return 1;
    }

    /* bucle de trabajo del main */
    for (i=0; i<num_iter; i++)
        printf("main iteración %d\n", i);

    /* espera a que termine el thread */
    WaitForSingleObject(thread, INFINITE);

    /* libera el manejador del thread */
    CloseHandle(thread);
    return 0;
}
```

3.14. LECTURAS RECOMENDADAS

Son muchos los libros de sistemas operativos que cubren los temas tratados en este capítulo. Algunos de ellos son [Crowley, 1997], [Milenkovic, 1992], [Silberchatz, 2005], [Stallings, 2001] y [Tanenbaum, 2006]. [Solomon, 1998] describe en detalle la gestión de procesos de Windows NT, y en [IEEE, 2004] y [IEEE, 1996] puede encontrarse una descripción completa de todos los servicios UNIX descritos en este capítulo.

3.15. EJERCICIOS

1. ¿Cuál de los siguientes mecanismos *hardware* no es un requisito para construir un sistema operativo multiprogramado con protección entre usuarios? Razone su respuesta.

e) Memoria virtual.
f) Protección de memoria.
g) Instrucciones de E/S que sólo pueden ejecutarse en modo kernel.
h) Dos modos de operación: privilegiado y usuario.

2. ¿Puede degradarse el rendimiento de la utilización del procesador en un sistema sin memoria virtual siempre que aumenta el grado de multiprogramación?

3. Indique cuál de estas operaciones no es ejecutada por el **activador**:

i) Restaurar los registros de usuario con los valores almacenados en la TLB del proceso.
j) Restaurar el contador de programa.
k) Restaurar el puntero que apunta a la tabla de páginas del proceso.
l) Restaurar la imagen de memoria de un proceso.

4. ¿Siempre se produce un cambio de proceso cuando se produce un cambio de contexto? Razone su respuesta.

5. ¿Cuál es la información que no comparten los *threads* de un mismo proceso?

6. ¿Cuál de las siguientes transiciones entre los estados de un proceso no se puede producir en un sistema con un algoritmo de planificación no expulsivo?

m)Bloqueado a listo.
n) Ejecutando a listo.
o) Ejecutando a bloqueado.
p) Listo a ejecución.

7. Sea un sistema que usa un algoritmo de planificación de procesos *round-robin* con una rodaja de tiempo de 100 ms. En este sistema ejecutan dos procesos. El primero no realiza operaciones de E/S y el segundo solicita una operación de E/S cada 50 ms. ¿Cuál será el porcentaje de uso de la UCP?

8. ¿Qué sucede cuando un proceso recibe una señal? ¿y cuando recibe una excepción?

9. ¿Cómo se hace en UNIX para que un proceso cree otro proceso que ejecute otro programa? ¿y en Windows?

10. ¿Qué información comparten un proceso y su hijo después de ejecutar el siguiente código?

```
if (fork()!=0)
    wait (&status);
else
    execve (B, argumentos, 0);
```

11. En un sistema operativo conforme a la norma UNIX, ¿cuándo pasa un proceso a estado *zombi*?

12. Tras la ejecución del siguiente código, ¿cuántos procesos se habrán creado?

```
for (i=0; i < n; i++)
    fork();
```

13. Cuando un proceso ejecuta un servicio `fork` y luego el proceso hijo un `exec`, ¿qué información comparten ambos procesos?

14. ¿Qué diferencia existe entre bloquear una señal e ignorarla en UNIX?

15. Escribir un programa en lenguaje C que active unos manejadores para las señales SIGINT, SIGQUIT y SIGILL. Las acciones a ejecutar por dichos manejadores serán:

16. Para SIGINT y SIGQUIT, abortar el proceso con un estado de error.

17. Para SIGILL, imprimir un mensaje de instrucción ilegal y terminar.

18. Dado el siguiente programa en C:

```
void main(int argc, char argv) {
    int i;
    for (i =1; i <= argc; i++)
        fork();
    ....
```

Se pide dibujar un esquema que muestre la jerarquía de procesos que se crea cuando se ejecuta el programa con `argc` igual a 3. ¿Cuántos procesos se crean si `argc` vale n?

19. Responder a las siguientes preguntas sobre los servicios al sistema `wait` de UNIX y `WaitForSingleObject` de Windows cuando está se aplica sobre un manejador de proceso.

a) ¿Cuál es la semántica de estos servicios?
b) Indicar en qué situaciones puede producirse cambio de proceso y en qué casos no.
c) Describir los pasos que se realizan en estos servicios desde que se llama a la rutina de biblioteca hasta que ésta devuelve el control al programa de usuario. Indicar cómo se transfiere el control y argumentos al sistema operativo, cómo realizaría su labor el sistema operativo y cómo devuelve el control y resultados al programa de usuario.

20. Escribir un programa similar al programa 3.15, que utilice los servicios de Windows para temporizar la ejecución de un proceso.

4

GESTIÓN DE MEMORIA

La memoria es uno de los recursos más importantes del computador y, en consecuencia, la parte del sistema operativo responsable de tratar con este recurso, el gestor de memoria, es un componente básico del mismo. Sin embargo, la gestión de memoria no es una labor del sistema operativo únicamente, sino que se realiza mediante la colaboración de distintos componentes de muy diversa índole que se reparten las tareas requeridas para llevar a cabo esta labor. Entre estos componentes, además del propio sistema operativo, están el lenguaje de programación, el compilador, el montador y el hardware de gestión de memoria. Aunque este capítulo se centra en el sistema operativo, se estudia cómo se integran los diversos componentes para proporcionar la funcionalidad requerida. Por otra parte, hay que resaltar que el gestor de memoria es una de las partes del sistema operativo que está más ligada al hardware. Esta estrecha relación ha hecho que tanto el hardware como el software de gestión de memoria hayan ido evolucionando juntos. Las necesidades del sistema operativo han obligado a los diseñadores del hardware a incluir nuevos mecanismos que, a su vez, han posibilitado el uso de nuevos esquemas de gestión de memoria. De hecho, la frontera entre la labor que realiza el hardware y la que hace el software de gestión de memoria es difusa y ha ido también evolucionando. El sistema de gestión de memoria ha sufrido grandes cambios según han ido evolucionando los sistemas operativos. Por ello, en esta presentación, se ha optado por desarrollar este tema de manera que no sólo se estudie el estado actual del mismo, sino que también se pueda conocer su evolución desde los sistemas más primitivos, ya obsoletos, hasta los más novedosos. Consideramos que este enfoque va a permitir que el lector entienda mejor qué aportan las características presentes en los sistemas actuales.

Por lo que se refiere a la organización del capítulo, en primer lugar, se analizarán aspectos generales sobre la gestión de memoria, lo que permitirá fijar los objetivos del sistema de memoria y establecer un contexto general para el resto del capítulo. Posteriormente, se mostrarán las distintas fases que conlleva la generación de un ejecutable y se estudiará cómo es el mapa de memoria de un proceso. En las siguientes secciones se analizará cómo ha sido la evolución de la gestión de la memoria, desde los sistemas multiprogramados más primitivos hasta los sistemas actuales basados en la técnica de memoria virtual. Por último, se estudiarán algunos de los servicios de gestión de memoria de UNIX y de Windows. El índice del capítulo es el siguiente:

- *Introducción.*
- *Aspectos generales de la gestión de memoria.*
- *Modelo de memoria de un proceso.*
- *Esquemas de gestión de la memoria del sistema.*
- *Memoria virtual.*
- *Servicios de gestión de memoria.*

4.1. INTRODUCCIÓN

Según se ha visto en capítulos anteriores, todo proceso requiere una **imagen de memoria** compuesta tanto por el código del programa como por los datos. Como muestra la figura 4.1, dicha imagen consta generalmente de una serie de **regiones**.

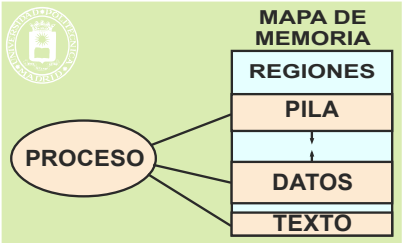


Figura 4.1 *Un proceso UNIX clásico incluye las regiones de memoria de Texto, Datos y Pila.*

Si bien la imagen de memoria del proceso clásico UNIX se organiza en las tres regiones de Texto, Datos y Pila, veremos en este capítulo que el proceso puede incluir otras regiones, por ejemplo, para dar soporte a los *threads* o a las bibliotecas dinámicas.

También se ha visto que el mapa de memoria del computador puede estar soportado solamente por la memoria principal del computador, lo que denominamos **memoria real**, o puede estar soportado por la memoria principal más una parte del disco denominada zona de intercambio o *swap*, lo que denominamos **memoria virtual**.

Finalmente, destacaremos que tanto el proceso como el procesador solamente entienden de direcciones del mapa de memoria. El procesador, de acuerdo al programa en ejecución, genera una serie de solicitudes de escritura o de lectura a determinadas direcciones del mapa de memoria, sin entrar en cuál es el soporte de esa dirección. Si el soporte es lento (porque la dirección está en *swap*) la unidad de memoria no deja esperando al procesador sino que le envía una interrupción para que se ejecute el sistema operativo y ponga en ejecución otro proceso.

La imagen de memoria del proceso se genera a partir del **fichero ejecutable**. Aunque analizaremos más adelante la estructura del fichero ejecutable, destacaremos aquí que dicho fichero contiene, entre otras cosas, el código objeto del programa, así como sus datos estáticos con valor inicial.

Diversos tipos de objetos de memoria

La imagen de memoria de un proceso debe almacenar el código y los datos. Sin embargo, no todos los datos de un programa tienen las mismas características. Por un lado, hay **datos de tipo constante** que, al igual que el **código**, no deberán ser modificados por ninguna sentencia del programa. Por otro lado, dependiendo del tipo de uso previsto para cada dato, se presentan las siguientes alternativas en cuanto al tiempo de vida del mismo:

- **Datos estáticos**. Se trata de datos que existen durante toda la ejecución del programa. Al inicio de la ejecución del programa se conoce cuántos datos de este tipo existen y se les habilita el espacio de almacenamiento requerido, que se mantendrá durante toda la ejecución del programa, es decir, tienen una dirección fija.

  En esta categoría están los datos declarados como globales y los datos declarados como estáticos.

- **Datos dinámicos asociados a la ejecución de una función**. La vida de este tipo de datos está asociada a la activación de una función, creándose en la invocación de la misma y destruyéndose cuando ésta termina. Estos datos se crean por el propio programa en la región de **pila** del proceso, formando parte del registro de activación de rutina o RAR. Igualmente, el programa les dará, en su caso, el valor inicial correspondiente.

  Dentro de esta categoría están los datos locales declarados dentro de las funciones así como los argumentos de las mismas.

  Cuando termina la función, termina la vida de estos datos, pero es de destacar que el RAR no se borra. Se sobrescribirá cuando se llame a otra función.

  Cada llamada a función o procedimiento exige la creación del correspondiente RAR. En este sentido, al ejecutar una función recursiva se irán apilando sucesivos RAR, uno por cada ciclo de recursividad.

- **Datos dinámicos controlados por el programa**. Se trata de datos dinámicos, pero cuyo tiempo de vida no está vinculado a la activación de una función, sino bajo el control directo del programa, que creará los datos cuando los necesite. El espacio asociado a los datos se liberará por el programa cuando ya no se requiera. Este tipo de datos se almacena habitualmente en una región denominada ***heap***.

  Ejemplo son los datos creados con la función `malloc` del lenguaje C o la función `new` de C++ o Java.

Regiones típicas del proceso

Las regiones típicas de un proceso son las tres de texto, datos y pila, que detallamos seguidamente:

- Región de **texto**: Contiene el código máquina del programa así como las constantes y las cadenas definidas en el mismo.
- Región de **datos**. Se organiza en las tres subregiones siguientes:
  - Datos con valor inicial: contiene las variables globales inicializadas.
  - Datos sin valor inicial: contiene las variables globales no inicializadas.
  - Datos creados dinámicamente o *heap*. En este espacio es dónde se crean las variables dinámicas.
- Región de **pila**: soporta el entorno del proceso más los registros de activación de los procedimientos.

La estructuración en regiones de los procesos depende del diseño del sistema operativo. En este sentido, puede haber una sola región de datos que englobe los datos con valor inicial, los datos sin valor inicial y los datos creados dinámicamente. O bien, puede haber regiones separadas para cada uno de ellos.

Una limitación importante de los sistemas con memoria virtual es que las regiones han de estar alineadas a página, ocupando, por tanto, siempre un número entero de páginas.

Gestor de memoria

El **gestor de memoria** del sistema operativo es el encargado de cubrir las necesidades de memoria de los procesos, por lo que tiene encomendadas las dos funciones siguientes:

- Como **servidor de memoria** debe asignar al proceso las regiones de memoria que puede utilizar, es decir, los rangos de direcciones del mapa de memoria que puede utilizar. También debe recuperar dichas regiones cuando el proceso ya no las necesite. De forma más concreta debe realizar las funciones siguientes:
  - Crear y mantener la imagen de memoria de los procesos a partir de los ficheros ejecutables, ofreciendo a cada proceso los espacios de memoria necesarios y dando soporte para las regiones necesarias.
  - Proporcionar grandes espacios de memoria para cada proceso.
  - Proporcionar protección entre procesos, aislando unos procesos de otros, pero permitiendo que los procesos puedan compartir regiones de memoria de una forma controlada.
  - Controlar los espacios de direcciones de los mapas de memoria ocupados y libres.
  - Tratar los errores de acceso a memoria: detectados por el *hardware*.
  - Optimizar las prestaciones del sistema.
- Como **gestor de recursos físicos** debe gestionar la asignación de memoria principal y de *swap*, en el caso de memoria virtual. De forma más concreta debe realizar las funciones siguientes:
  - Controlar los espacios de direcciones de memoria principal y de intercambio ocupados y libres.
  - Asignar espacios de memoria principal y de *swap*.
  - Recuperar los recursos físicos de almacenamiento asignados cuando el proceso ya no los necesite.

Conviene no confundir estas dos funciones de servidor de memoria y de gestor de recursos físicos, puesto que cambiar de soporte físico una página, copiándola de *swap* a un marco de página de memoria principal, no representa asignar nueva memoria al proceso. Éste seguirá disponiendo del mismo espacio de direcciones; solamente se ha cambiado el soporte físico de una parte de ese espacio.

Protección de memoria

La protección de memoria es necesaria tanto en monoproceso como multiproceso, puesto que hay que proteger al sistema operativo del o de los procesos activos y hay que proteger a unos procesos frente a otros. Desde el punto de vista de la memoria hay que garantizar que los accesos a memoria que realizan los procesos son correctos. Para ello es necesario validar cada una de las direcciones que generen los procesos.

Evidentemente, esta validación exige una supervisión continua, que no puede realizar el sistema operativo, por lo que, asociado a la unidad de memoria, es necesario disponer de un *hardware* que analice la dirección y el tipo de cada acceso a memoria. Este *hardware* generará una excepción cuando detecte un acceso inválido.

El tratamiento del acceso inválido lo hace el sistema operativo, como varemos más adelante.

Otro aspecto de la protección de memoria es el relativo a la reutilización de los elementos físicos de memoria, ya sea memoria principal o *swap*. Antes de asignar un recurso físico de memoria a un proceso hay que garantizar que éste no podrá leer información dejada en ese recurso por otro proceso que tuvo previamente asignado el recurso. Es por tanto, necesario escribir en el recurso rellenándolo con la información inicial del propio proceso o **rellenándolo con 0** si no existe información inicial.

4.2. JERARQUÍA DE MEMORIA

En el capítulo "1 Conceptos arquitectónicos del computador" se introdujo el concepto de jerarquía de memoria, necesario para tener un gran almacenamiento a un precio económico y que satisfaga las necesidades de velocidad de los procesadores. Como muestra la figura 4.2, el almacenamiento básico permanente está formado por discos de gran capacidad de almacenamiento y baratos, pero lentos. La memoria principal es lenta comparada con la velocidad de los procesadores, por lo que se intercala la memoria cache, que, en los computadores actuales está organizada, a su vez, por tres niveles de cache.

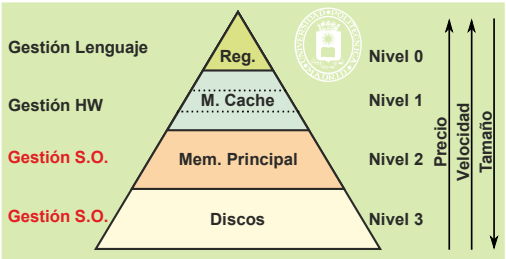


Figura 4.2 *Jerarquía de memoria*

4.2.1. Migración de la información

La explotación correcta de la jerarquía de memoria exige tener, en cada momento, la información adecuada en el nivel adecuado. Para ello, la información ha de moverse de nivel, esto es, ha de migrar de un nivel a otro. Esta migración puede ser bajo demanda explícita o puede ser automática. La primera alternativa exige que el programa solicite explícitamente el movimiento de la información, como ocurre, por ejemplo, con un programa editor, que va solicitando la parte del fichero que está editando el usuario en cada momento. La segunda alternativa consiste en hacer la migración transparente al programa, es decir, sin que éste tenga que ser consciente de que se produce. La migración automática se utiliza en las memorias cache y en la memoria virtual, mientras que la migración bajo demanda se utiliza en los otros niveles.

La **migración automática** toma como base la secuencia de direcciones —o referencias— que genera el procesador al ir ejecutando un programa máquina. Sean k y $k+1$ dos niveles consecutivos de la jerarquía, siendo k el nivel más rápido. La existencia de una migración automática de información permite que el programa referencie la información en el nivel k y que, en el caso de que no exista una copia de esa información en dicho nivel k, se traiga ésta desde el nivel $k+1$ sin que el programa tenga que hacer nada para ello. Se dice que hay **acierto** cuando el programa en ejecución referencia una información que se encuentra en el nivel rápido y que hay **fallo** en el caso contrario.

El funcionamiento correcto de la migración automática exige un mecanismo que consiga tener en el nivel k aquella información que necesita el programa en ejecución en cada instante, de forma que se produzcan muy pocos fallos. Lo ideal sería que el mecanismo pudiera predecir la información que el programa necesitará para tenerla disponible en el nivel rápido k. El mecanismo se basa en los siguientes aspectos:

- Tamaño de las porciones transferidas.
- Política de extracción.
- Política de reemplazo.
- Política de ubicación.

La **política de extracción** define qué información se sube del nivel $k+1$ al k y cuándo se sube. La solución más corriente es la denominada **por demanda,** que consiste en subir aquella información que referencia el programa, justo cuando sucede el fallo. El éxito de la jerarquía de memoria se basa en gran parte en la proximidad espacial (véase sección "4.2.5 La proximidad referencial"), para cuya explotación no se sube exclusivamente la información referenciada sino una porción mayor. En concreto, para la memoria cache se transfieren **líneas** de unas pocas palabras, mientras que para la memoria virtual se transfieren **páginas** de uno o varios KiB. El tamaño de estas porciones es una característica muy importante de la jerarquía de memoria.

El nivel k tiene menor capacidad de almacenamiento que el nivel $k+1$, por lo que normalmente está lleno. Por ello, cuando se sube una porción de información hay que eliminar otra. La **política de reemplazo** determina qué porción hay que eliminar, tratando de seleccionar una que ya no sea de interés para el programa en ejecución.

Por razones constructivas pueden existir limitaciones en cuanto al lugar en el que se pueden almacenar las diversas porciones de información, la **política de ubicación** determina dónde se puede almacenar cada porción.

4.2.2. Parámetros característicos de la jerarquía de memoria

La eficiencia de la jerarquía de memoria se mide mediante los dos parámetros siguientes:

- Tasa de aciertos o *hit ratio* (*Hr*).
- Tiempo medio de acceso efectivo (*Tef*).

La **tasa de aciertos** Hr_k del nivel k de la jerarquía se define como la probabilidad de encontrar en ese nivel la información referenciada. La tasa de fallos Fr_k es igual a $1-Hr_k$. La tasa de aciertos Hr_k ha de ser muy alta para que sea rentable el uso del nivel k de la jerarquía. Los factores más importantes que determinan Hr_k son los siguientes:

- Tamaño de la porción de información que se transfiere al nivel k.
- Capacidad de almacenamiento del nivel k.
- Política de reemplazo.
- Política de ubicación.
- Programa específico que se esté ejecutando (cada programa tiene su propio comportamiento).

El tiempo de acceso a una información depende de que se produzca o no fallo en el nivel k. Denominaremos tiempo de acierto al tiempo de acceso cuando la información se encuentra en el nivel k, mientras que denominare-

mos penalización de fallo al tiempo que se tarda en realizar la migración de la porción cuando se produce fallo. El **tiempo medio de acceso efectivo** (***Tef***) de un programa se obtiene promediando los tiempos de todos los accesos que realiza el programa a lo largo de su ejecución.

Tef = Ta + (1 − Hr)•Pf

- *Ta* Tiempo de acierto.
- *Pf* Penalización de fallo.
- *Hr* Tasa de aciertos.

Cuanto mayor sea la penalización de fallo, mayor ha de ser la tasa de aciertos para que el tiempo medio de acceso efectivo sea pequeño, es decir, para que compense el uso de la jerarquía de memoria.

4.2.3. Coherencia

Un efecto colateral de la jerarquía de memoria es que existen varias copias de determinadas porciones de información en distintos niveles. Al escribir sobre la copia del nivel *k*, se produce una discrepancia con la copia del nivel inferior *k+1*; esta situación se denomina falta de coherencia. Se dice que una porción de información está **sucia** si ha sido escrita en el nivel *k* pero no ha sido actualizada en el *k+1*.

La consistencia de la jerarquía de memoria exige medidas para eliminar la falta de coherencia. En concreto, una porción sucia en el nivel *k* ha de ser escrita en algún momento al nivel inferior *k+1* para eliminar la falta de coherencia. Con esta operación de escritura se **limpia** la porción del nivel *k*.

También se puede dar el caso contrario, es decir, que se modifique directamente la copia del nivel *k+1*. Para que no se produzcan accesos erróneos, se deberá anular o borrar la copia del nivel superior *k*.

La coherencia adquiere mayor relevancia y dificultad cuando el sistema tiene varias memorias de un mismo nivel. Esto ocurre, por ejemplo, en los multiprocesadores que cuentan con una memoria cache privada por cada procesador (véase sección "1.8 Multiprocesador y multicomputador"). Si lo que escribe un procesador se quedase en su cache y no fuese accesible por los otros procesadores se podrían producir errores de coherencia, por ejemplo, que un programa ejecutando en el procesador N2 no accediese a la copia correcta de la información que le genera otro programa ejecutando en el procesador N1. Para evitar este problema, los buses de conexión de las caches con la memoria principal deben implementar unos algoritmos llamados *snoop* (fisgones) que llevan constancia de los contenidos de las caches y copian la información para que no surjan incoherencias.

Existen diversas políticas de actualización de la información creada o modificada, que se caracterizan por el instante en el que se copia la información al nivel permanente.

4.2.4. Direccionamiento

La jerarquía de memoria presenta un problema de direccionamiento. Supóngase que el programa en ejecución genera la dirección *X* del dato *A,* al que quiere acceder. Esta dirección *X* está referida al nivel *k+1*, pero se desea acceder al dato *A* en el nivel *k*, que es más rápido. Para ello, se necesitará conocer la dirección *Y* que ocupa *A* en el nivel *k*, por lo que será necesario establecer un mecanismo de traducción de direcciones *X* en sus correspondientes *Y*.

El problema de traducción no es trivial. Supóngase que el espacio de nivel *k+1* es de 32 GiB, lo que exige direcciones de 35 bits ($n = 35$), y que el espacio de nivel *k* es de 4 GiB, lo que requiere direcciones de 32 bits ($m = 32$). El traductor tiene aproximadamente 34 mil millones de valores de entrada distintos y 4 mil millones de direcciones finales.

Para simplificar la traducción y, además, aprovechar la proximidad espacial, se dividen los mapas de direcciones de los espacios *k+1* y *k* en porciones de tamaño 2^p. Estas porciones constituyen la unidad de información mínima que se transfiere de un nivel al otro. El que la porción tenga tamaño 2^p permite dividir la dirección en dos partes: los $m - p$ bits más significativos sirven para identificar la porción, mientras que los *p* bits menos significativos sirven para especificar el byte o la palabra dentro de la porción (véase la figura 4.3). Dado que al migrar una porción del nivel *k+1* al *k* se mantienen las posiciones relativas dentro de la misma, el problema de traducción se reduce a saber dónde se coloca la porción en el espacio *k*.

Suponiendo, para el ejemplo anterior, que las porciones son de 4 K byte ($p = 12$), el problema de direccionamiento queda dividido por 4096. Pero sigue siendo inviable plantear la traducción mediante una tabla directa completa, pues sería una tabla de unos 8 millones de entradas y con 1 millón salidas válidas.

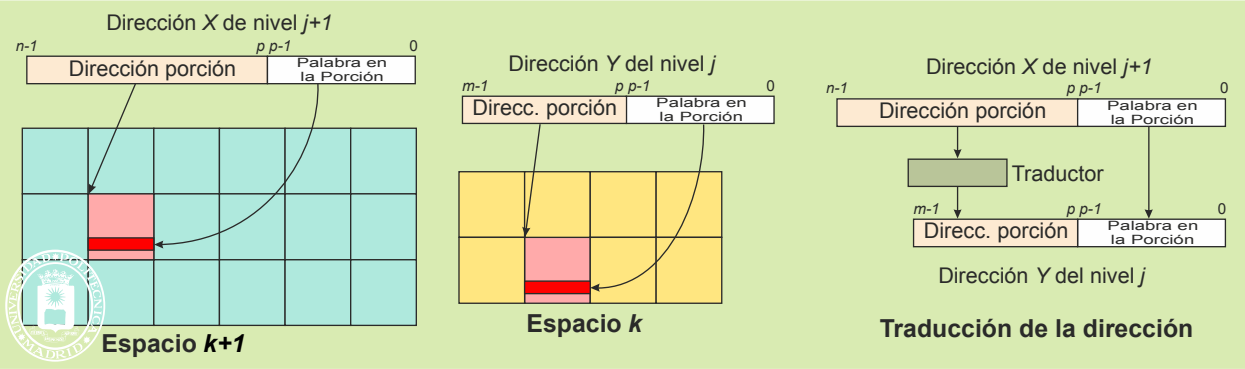


Figura 4.3 *El uso de porciones de 2^p facilita la traducción de direcciones.*

4.2.5. La proximidad referencial

La proximidad referencial es la característica que hace viable la jerarquía de memoria, de ahí su importancia. En términos globales la proximidad referencial establece que un programa en ejecución utiliza en cada momento una pequeña parte de toda su información, es decir, de todo su código y de todos sus datos.

Para exponer el concepto de proximidad referencial de forma más específica partimos del concepto de traza. La **traza** de un programa en ejecución es la lista ordenada en el tiempo de las direcciones de memoria que referencia para llevar a cabo su ejecución. Esta traza *R* estará compuesta por las direcciones de las instrucciones que se van ejecutando y por las direcciones de los datos empleados. R se representa así:

Re = re(1), re(2), re(3), ... re(j)

donde *re(i)* es la *i*-ésima dirección generada por la ejecución del programa e.

Adicionalmente, se define el concepto de distancia *d(u, v)* entre dos direcciones *u* y *v* como la diferencia en valor absoluto $|u - v|$. La distancia entre dos elementos *j* y *k* de una traza *R(e)* es, por tanto, $d(re(j), re(k)) = |re(j) - re(k)|$.

También se puede hablar de traza de E/S, refiriéndonos, en este caso, a la secuencia de las direcciones de periférico empleadas en operaciones de E/S.

La proximidad referencial presenta dos facetas: la proximidad espacial y la proximidad temporal.

La **proximidad espacial** de una traza postula que hay una alta probabilidad de referenciar direcciones cercanas a las utilizadas últimamente. Dicho de otra forma: dadas dos referencias *re(j)* y *re(i)* próximas en el espacio (es decir, que la distancia de sus direcciones $i - j$ sea pequeña), existe una alta probabilidad de que su distancia en la traza *d(re(j), re(i))* sea muy pequeña. Además, como muchos trozos de programa y muchas estructuras de datos se recorren secuencialmente, existe una gran probabilidad de que la referencia siguiente a *re(j)* coincida con la dirección de memoria siguiente (aclaración 4.1). Este tipo especial de proximidad espacial recibe el nombre de **proximidad secuencial**.

Aclaración 4.1. Dado que las memorias principales se direccionan a nivel de byte, pero se acceden a nivel de palabra, la dirección siguiente no es la dirección actual más 1. En máquinas de 64 bits (palabras de 8 bytes) la dirección siguiente es la actual más 8.

La proximidad espacial se explica si se tienen en cuenta los siguientes argumentos:

- Los programas son fundamentalmente secuenciales, a excepción de las bifurcaciones, por lo que su lectura genera referencias consecutivas.
- La gran mayoría de los bucles son muy pequeños, de unas pocas instrucciones máquina, por lo que su ejecución genera referencias con distancias de direcciones pequeñas.
- Las estructuras de datos que se recorren de forma secuencial o con referencias muy próximas son muy frecuentes. Ejemplos son los vectores, las listas, las pilas, las matrices, etc. Además, las zonas de datos suelen estar agrupadas, de manera que las referencias que se generan suelen estar próximas.

La **proximidad temporal** postula que un programa en ejecución tiende a volver a referenciar direcciones empleadas en un pasado cercano. Esto es, existe una probabilidad muy alta de que la próxima referencia *re(j + 1)* esté entre las *n* referencias anteriores *re(j – n + 1), re(j – n + 2), ... , re(j – 2), re(j – 1), re(j)*. La proximidad temporal se explica si se tienen en cuenta los siguientes argumentos:

- Los bucles producen proximidad temporal, además de proximidad espacial.
- El uso de datos o parámetros de forma repetitiva produce proximidad temporal.
- Las llamadas repetidas a subrutinas también son muy frecuentes y producen proximidad temporal. Esto es muy típico con las funciones o subrutinas aritméticas, de conversión de códigos, etc.

Conjunto de trabajo

La proximidad referencial tiene como resultado que las referencias producidas por la ejecución de un programa están agrupadas en unas pocas zonas de memoria, tal y como muestra la figura 4.4. Se denomina **conjunto de trabajo**

a la suma de las zonas que utiliza un programa en un pequeño intervalo de tiempo. Puede observarse también que, a medida que avanza la ejecución del programa, va variando su conjunto de trabajo.

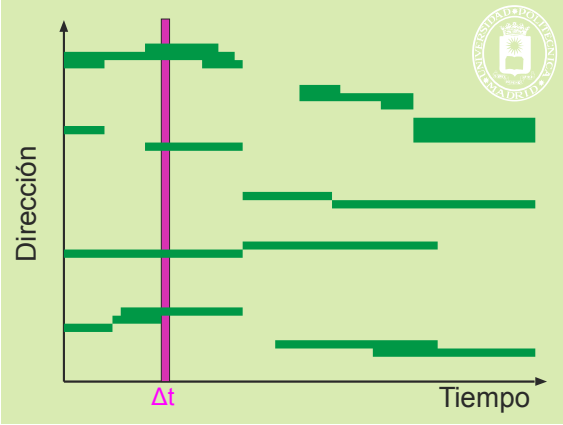


Figura 4.4 *Proximidad referencial.*

El objetivo principal de la gestión de la jerarquía de memoria será conseguir que residan en las memorias más rápidas aquellas zonas de los programas que están siendo referenciadas en cada instante, es decir, su conjunto de trabajo.

4.2.6. Concepto de memoria cache

El término cache deriva del verbo francés *cacher*, que significa ocultar, esconder. Con este término se quiere reflejar que la memoria cache no es visible al programa máquina, puesto que no está ubicada en el mapa de memoria. Se trata de una memoria de apoyo a la memoria principal que sirve para acelerar los accesos. La memoria cache alberga información recientemente utilizada, con la esperanza de que vuelva a ser empleada en un futuro próximo. Los aciertos sobre cache permiten atender al procesador más rápidamente que accediendo a la memoria principal.

El bloque de información que migra entre la memoria principal y la cache se denomina línea y está formado por varias palabras (valores típicos de línea son de 32 a 128 bytes). Toda la gestión de la cache necesaria para migrar líneas se realiza por *hardware*, debido a la gran velocidad a la que debe funcionar. El tiempo de tratamiento de un fallo tiene que ser del orden del tiempo de acceso a la memoria lenta, es decir, de los 60 a 200 ns que se tarda en acceder a la memoria principal, puesto que el procesador se queda esperando a poder realizar el acceso solicitado.

En la actualidad, debido a la gran diferencia de velocidad entre los procesadores y las memorias principales, se utilizan varios niveles de cache, incluyéndose los más rápidos en el mismo chip que el procesador. A título de ejemplo, indicaremos que el procesador «Itanium 2 processor 6M» anunciado por Intel en el año 2004 tiene tres niveles de memoria cache. El nivel más rápido lo forman dos caches L1 de 16 KiB cada una, una dedicada a las instrucciones y otra a los datos. El segundo nivel es una única L2 de 256 KiB, mientras que el tercer nivel L3 tiene 6 MiB. Las memorias L1 se acceden en un ciclo (el procesador trabaja a 1,5 GHz), mientras que la L2 lo hace en 5 ciclos y la L3 en 14 ciclos.

Aunque la memoria cache es una memoria oculta, no nos podemos olvidar de su existencia, puesto que repercute fuertemente en las prestaciones de los sistemas. Plantear adecuadamente un problema para que genere pocos fallos de cache puede disminuir espectacularmente su tiempo de ejecución.

4.2.7. Concepto de memoria virtual y memoria real

Máquina de memoria real

Una máquina de memoria real es una máquina convencional que solamente utiliza memoria principal para soportar el mapa de memoria. Por el contrario, una máquina con memoria virtual soporta su mapa de memoria mediante dos niveles de la jerarquía de memoria: la memoria principal y una memoria de respaldo (que suele ser el disco, aunque puede ser una memoria expandida, es decir una memoria RAM auxiliar). Sobre la memoria de respaldo se proyecta el mapa de memoria, que se denomina virtual para diferenciarlo de la memoria real. Las direcciones generadas por el procesador se refieren a este mapa virtual pero, sin embargo, los accesos reales se realizan sobre la memoria principal, más rápida que la de respaldo.

Máquina con memoria virtual

La **memoria virtual** es un mecanismo de migración automática, por lo que exige una gestión automática de la parte de la jerarquía de memoria formada por la memoria principal y una parte del disco denominada zona de intercambio. Esta gestión la realiza el sistema operativo con ayuda de una unidad *hardware* de gestión de memoria, llamada **MMU** (*Memory Management Unit*) que veremos en la sección “4.2.9 Unidad de gestión de memoria (MMU)”. Como muestra la figura 4.5, esta gestión incluye toda la memoria principal y la parte del disco que sirve de respaldo a la memoria virtual.

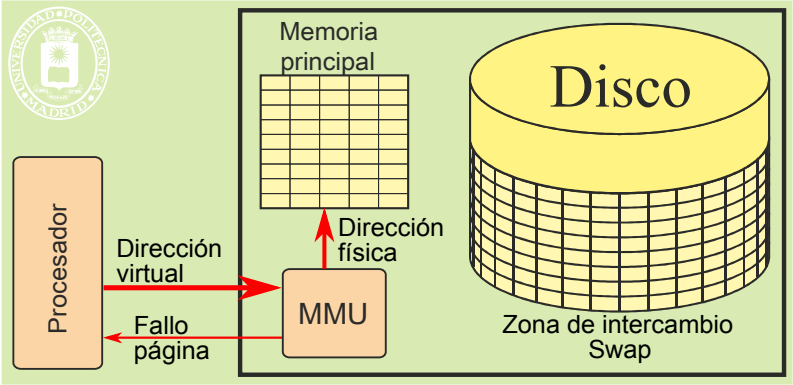


Figura 4.5 *Fundamento de la memoria virtual.*

Como muestra la figura 4.6, ambos espacios virtual y físico se dividen en páginas. Se denominan **páginas virtuales** a las páginas del espacio virtual, **páginas de intercambio** o de *swap* a las páginas residentes en la zona de intercambio y **marcos de página** a los espacios en los que se considera dividida la memoria principal. Normalmente, cada marco de página puede albergar una página virtual cualquiera, sin ninguna restricción de direccionamiento.

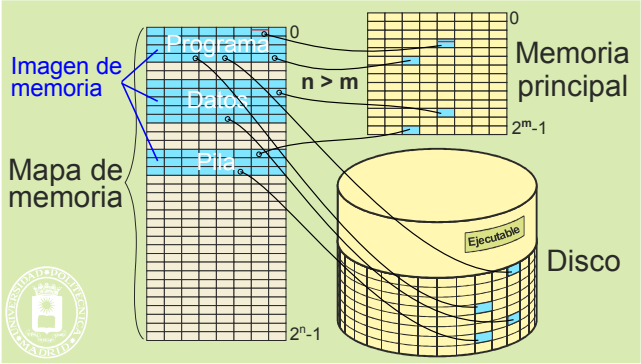


Figura 4.6 *Tanto el mapa de memoria como la memoria principal y la zona de intercambio del disco se dividen en páginas de igual tamaño.*

Los aspectos principales en los que se basa la memoria virtual son los siguientes:

- Las direcciones generadas por las instrucciones máquina, tanto para referenciar datos como otras instrucciones, están referidas al mapa de memoria que constituye el espacio virtual. En este sentido, se suele decir que el procesador genera direcciones virtuales.
- El mapa virtual asociado a cada programa en ejecución está soportado físicamente por una zona del disco, denominada de **intercambio** o ***swap***, y por la memoria principal. Téngase en cuenta que toda la información del programa ha de residir obligatoriamente en algún soporte físico, ya sea disco o memoria principal, aunque también puede estar duplicada en ambos.
- Los trozos de mapa virtual no utilizados por los programas no tienen soporte físico, es decir, no ocupan recursos reales.
- Aunque el programa genera direcciones virtuales, para que éste pueda ejecutar, han de residir en memoria principal las instrucciones y los datos utilizados en cada momento. Si, por ejemplo, un dato referenciado por una instrucción máquina no reside en la memoria principal es necesario realizar un transvase de información (migración de información) entre el disco y la memoria principal, antes de que el programa pueda seguir ejecutando. Dado que el disco es del orden del millón de veces más lento que la memoria principal, el procesador no se queda esperando a que se resuelva el fallo, al contrario de lo que ocurre con la cache.
- En cada instante, solamente reside en memoria principal una fracción de las páginas del programa, fracción que se denomina **conjunto residente**. Por tanto, la traducción no siempre es posible. Cuando la palabra solicitada no esté en memoria principal la MMU producirá una excepción *hardware síncrona*, denominada **excepción de fallo de página**. El sistema operativo resuelve el problema cargando la página necesaria en un marco de página, modificando, por tanto, el conjunto residente.
- No hay que confundir el conjunto residente, que acabamos de definir, con el conjunto de trabajo, que especifica las páginas que deben estar en memoria para que el proceso no sufra fallos de página. El conjunto residente ha de irse adaptando a las necesidades del proceso para contener al conjunto de trabajo. Sin embargo, como el conjunto de trabajo solamente es conocido a posteriori, es decir, una vez ejecutado el programa, alcanzar este objetivo de forma óptima es imposible.
- Los fallos de página son atendidos por el sistema operativo, que se encarga de realizar la adecuada migración de páginas para traer la página requerida por el programa a un marco de página, actualizando el conjunto residente. Se denomina **paginación** al proceso de migración necesario para atender los fallos de página.

4.2.8. La tabla de páginas

La tabla de páginas es una estructura de información que almacena la ubicación de páginas virtuales en marcos de memoria. Puede existir una **única tabla común** para todos los procesos, lo que conlleva un mapa de memoria único que comparten todos los procesos, o una **tabla por proceso**, lo que implica que cada proceso tiene su propio mapa de memoria.

Lo más frecuente es que cada proceso tenga su propio **mapa de memoria independiente**, es decir, su propio espacio virtual independiente, tal y como se muestra en la figura 4.7. El mapa vigente viene determinado por el valor contenido en el registro RIED, valor que indica la posición en la que se ubica la tabla de páginas del proceso que tiene asignado ese mapa de memoria.

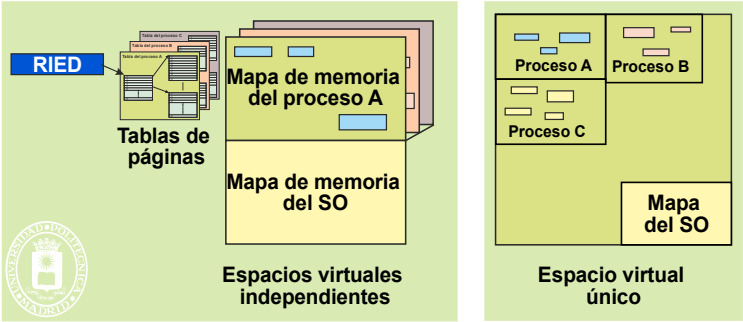


Figura 4.7 *En el caso más frecuente cada proceso tiene su propio mapa de memoria y su propia tabla de páginas que define dicho mapa. Por el contrario, en el caso de un espacio virtual único existe una tabla de páginas común a todos los procesos.*

Además, la tabla de páginas se puede organizar como una tabla directa o como una tabla inversa.

- La **tabla directa**, que puede ser única o que puede haber una por proceso, tiene una entrada por cada página virtual, entrada que contiene el número de marco que la alberga o una indicación de que no está en memoria principal.
- La **tabla inversa**, que solamente puede ser única, tiene una entrada por cada marco de página que tenga el sistema. Dicha entrada contiene el número de página virtual que está almacenado en el marco, así como el proceso al que pertenece.

Tabla de páginas directa

La tabla de páginas directa puede ser una tabla de un solo nivel o puede tener una estructura en árbol con dos o más niveles.

Tabla de páginas directa de un nivel

La figura 4.8 muestra una primera solución muy sencilla, en la que cada programa en ejecución tiene asignado una tabla de páginas directa de un nivel. Para que la tabla no tenga elementos nulos (lo que penalizaría su tamaño), se supone que toda la memoria asignada al programa es contigua. Como cada programa dispone de su propio espacio virtual, se puede colocar al principio del mismo, en la página virtual 0 y sucesivas.

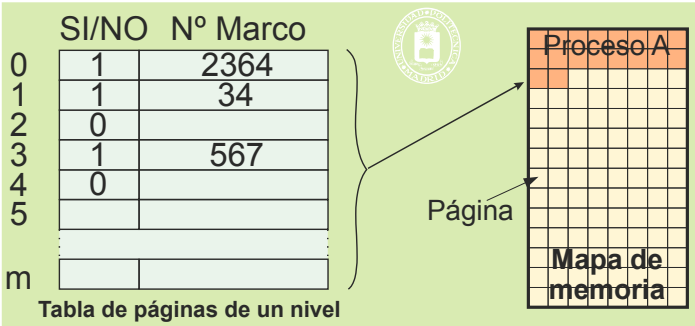


Figura 4.8 *Tabla de páginas directa de un nivel y espacio virtual asignado.*

La tabla tiene una entrada por página y el número de la página virtual se utiliza como índice para entrar en ella. Cada elemento de la tabla tiene un bit para indicar si la página está en memoria principal y el número de marco o la página de intercambio en el que se encuentra la mencionada página.

La figura 4.9 muestra un ejemplo de traducción para el caso de tabla de páginas directa de un nivel. Se supone que las páginas son de 1 KiB, por lo que los 10 bits inferiores de la dirección virtual sirven para especificar el byte dentro de la página, mientras que el resto especifican la página virtual.

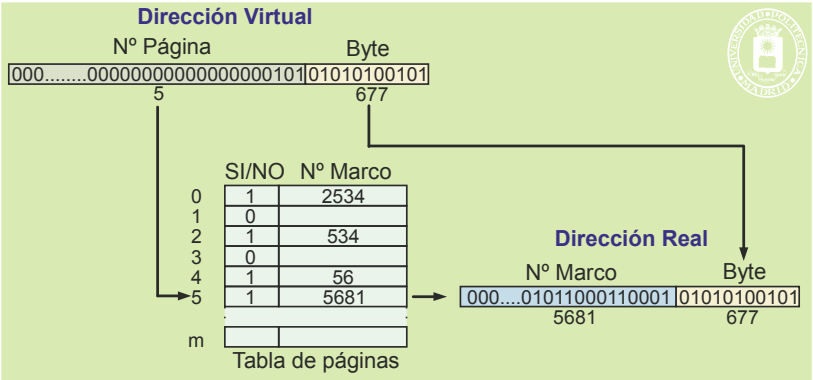


Figura 4.9 *Ejemplo de traducción mediante tabla de páginas directa de un nivel. La página virtual es la 5. Entrando en la sexta posición de la tabla se observa que la página está en memoria principal y que está en el marco nº 5681. Concatenando el nº de marco con los 10 bits inferiores de la dirección virtual se obtiene la dirección de memoria principal donde reside la información buscada.*

El mayor inconveniente de la tabla de un nivel es su falta de flexibilidad. Si no se quiere que existan entradas nulas en la tabla, la memoria virtual asignada ha de ser contigua (advertencia 4.1) y la ampliación de memoria solamente puede hacerse al final de la zona asignada. Sin embargo, los programas están compuestos por varios elementos, como son el propio programa objeto, la pila y los bloques de datos. Además, tanto la pila como los bloques de datos han de poder crecer. Por ello, un esquema de tabla de un nivel obliga a dejar grandes huecos de memoria virtual sin utilizar, pero que están presentes en la tabla con el consiguiente desperdicio de espacio.

Advertencia 4.1. El espacio virtual asignado es contiguo, sin embargo, los marcos de página asignados estarán dispersos por toda la memoria principal.

Por ello, se emplean esquemas de tablas de páginas directas de más de un nivel.

Tabla de páginas directa multinivel

La tabla de páginas directa multinivel tiene una estructura en árbol con dos o más niveles. Por simplicidad consideraremos tablas de dos niveles, pero los computadores actuales pueden llegar a tener hasta 4 niveles.

La figura 4.10 muestra el caso de tabla de páginas de dos niveles. Con este tipo de tabla, la memoria asignada está compuesta por una serie de **bloques de memoria** virtual. Cada bloque está formado por una serie contigua de bytes. Su tamaño puede crecer, siempre y cuando no se solape con otro bloque. La dirección se divide en tres partes. La primera identifica el bloque de memoria donde está la información a la que se desea acceder. Con este valor se entra en una subtabla de bloques, que contiene un puntero por bloque, puntero que indica el comienzo de la subtabla de páginas del bloque. Con la segunda parte de la dirección se entra en la subtabla de páginas seleccionada. Esta subtabla es similar a la tabla mostrada en la figura 4.8, lo que permite obtener el marco en el que está la información deseada.

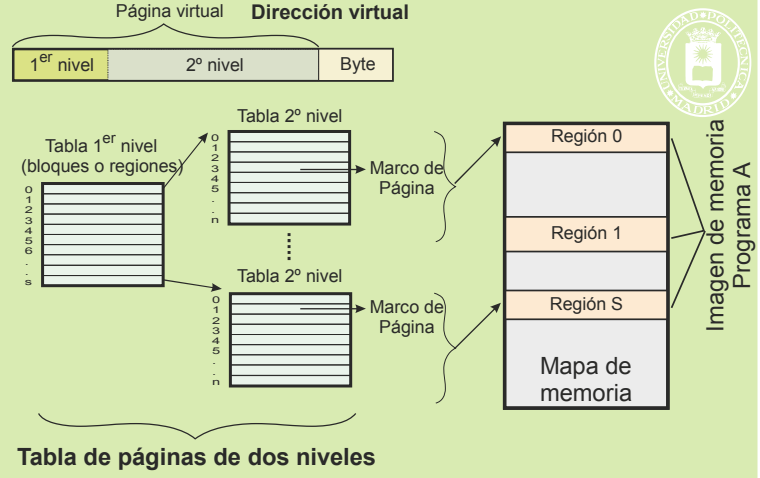


Figura 4.10 *Tabla de páginas de dos niveles.*

La ventaja del diseño con varios niveles es que permite una asignación de memoria más flexible que con un solo nivel, puesto que se pueden asignar bloques de memoria virtual disjuntas, por lo que pueden crecer de forma independiente. Además, la tabla de páginas no tiene grandes espacios vacíos, por lo que ocupa solamente el espacio imprescindible.

Como muestra la figura 4.11, tanto las entradas de la tabla de páginas de primer como segundo nivel suelen incluir los tres bits `rwx` (*read, write* y *execution*) que determinan los permisos de accesos. Los bits de la tabla de primer nivel afectan a todas las páginas que dependen de esa entrada. Los bits de la tabla de segundo nivel permiten restringir alguno de estos permisos para páginas específicas. Las entradas de la tabla de 2º nivel incluye otras infor-

maciones como el bit de presente/ausente (que indica si la página está en un marco de memoria), el bit de referenciada (que indica que el marco ha sido referenciado) y el bit de modificada (que indica que el marco está sucio).

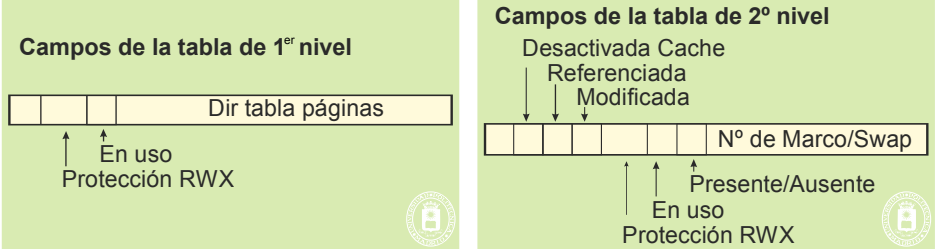


Figura 4.11 *Formatos de las entradas de las tablas de páginas de primer y segundo nivel.*

La figura 4.12 muestra un ejemplo de traducción mediante tabla de páginas de dos niveles.

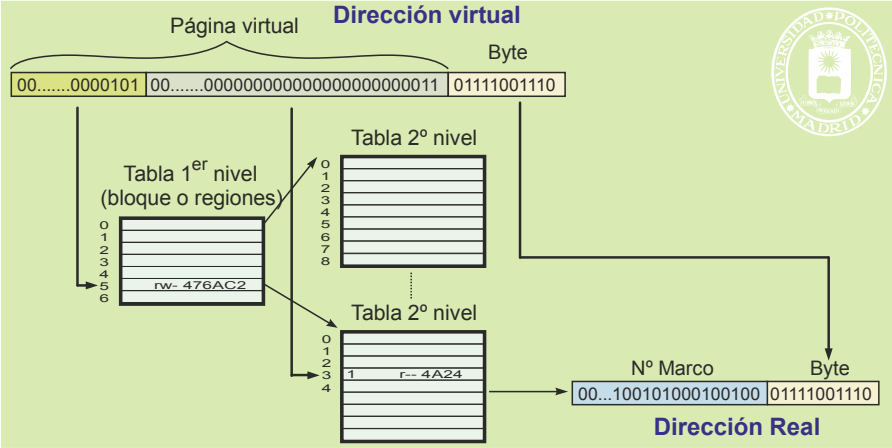


Figura 4.12 *Tabla de páginas de dos niveles. La región direccionada es la 5, por lo que hay que leer la sexta entrada de la tabla de regiones. Con ello se obtiene la dirección donde comienza la tabla de páginas de esta región. La página direccionada es la 3, por lo que entramos en la cuarta posición de la tabla anterior. En esta tabla encontramos que el marco es el H'4A24 por lo que se puede formar la dirección física en la que se encuentra la información buscada.*

Soporte físico de la memoria virtual

La información asociada a la memoria virtual se puede clasificar en metainformación e información.

Metainformación

La metainformación la constituyen las tablas de página, que son almacenadas en memoria principal en espacio del sistema operativo.

Información

La información corresponde a las páginas que soportan la imagen de memoria de cada proceso. El soporte físico puede ser uno de los siguientes:

- Páginas almacenadas en la zona de intercambio del disco.
- Páginas almacenadas en marcos de página de la memoria principal.
- En el caso de ficheros proyectados, el propio fichero.
- Paginas sin valor inicial, que no tienen soporte físico y que denominamos a rellenar con 0. Solamente se asigna un marco de página cuando se escriba en ellas por primera vez. El marco ha de ser borrado previamente.

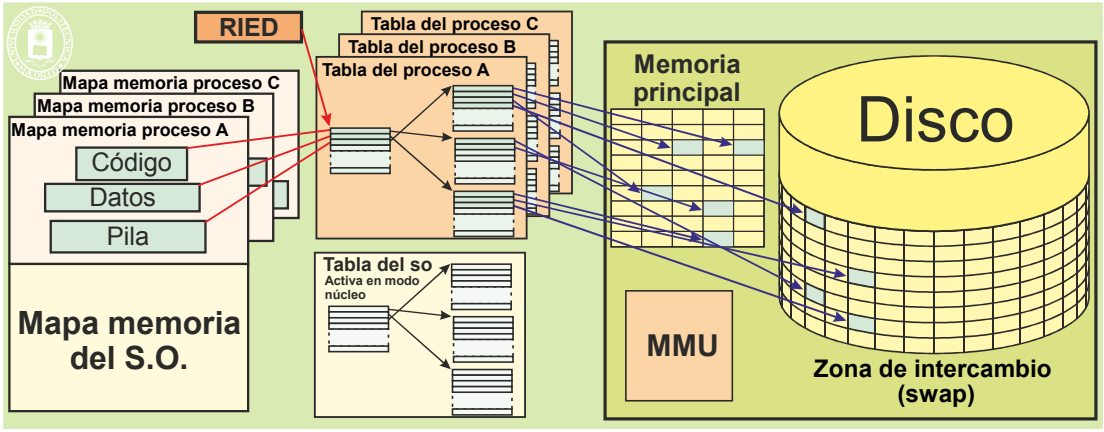


Figura 4.13 *Visión general de un sistema con memoria virtual, mostrando la tablas de páginas, los marcos de página, la zona de intercambio y la MMU.*

En el resto del libro nos centraremos en el caso de mapa de memoria independiente por proceso, mapa soportado por una tabla directa multinivel.

4.2.9. Unidad de gestión de memoria (MMU)

El sistema operativo determina las páginas virtuales, que puede utilizar cada programa en su ejecución, y asigna el soporte físico de cada página, ya sea un marco o una página de intercambio. Esta información queda reflejada en la tabla de páginas, que es construida y mantenida por el sistema operativo.

Sin embargo, el sistema operativo no es más que un programa, por lo que, en un computador monoprocesador, cuando está ejecutando un programa de usuario no está ejecutando el sistema operativo y, viceversa, cuando está ejecutando éste no lo está el programa de usuario. Esto hace que sea imposible que las funciones que hay que hacer en cada acceso a memoria de un programa, es decir, una o varias veces por cada instrucción máquina, las realice otro programa. Estas funciones son:

- Traducción de las direcciones.
- Marcado de páginas como sucias y accedidas.
- Vigilancia de los accesos para detectar posibles accesos incorrectos.

Estas funciones deben hacerse por un *hardware* especial que denominamos **MMU** (*Memory Management Unit*). Trataremos en esta sección las funciones de traducción de direcciones y de marcado de páginas sucias, y dejaremos la vigilancia para la sección "1.7.2 Mecanismos de protección de memoria".

La traducción de cada dirección de memoria la realiza la MMU usando la información de la tabla de páginas del programa en ejecución. Dado que el sistema operativo mantiene una tabla de páginas por cada programa activo, existe un registro para indicar a la MMU la dirección de memoria donde se encuentra la tabla que debe utilizar. Dicho registro, que denominaremos RIED (Registro Identificador del Espacio de Direccionamiento), apunta a la posición de inicio de la correspondiente tabla de primer nivel.

La MMU realiza los pasos descritos en la figura 4.12 para hacer la traducción. Recalcamos que esta traducción hay que hacerla por *hardware* dada la alta velocidad a la que debe hacerse (una fracción del tiempo de acceso de la memoria principal, o de la memoria cache si existe), pues de lo contrario se penalizarían gravemente las prestaciones del computador.

Como la tabla de páginas se almacena en memoria principal, en un sistema con tabla de páginas de dos niveles, la MMU debe hacer dos accesos a memoria principal por cada traducción de dirección. Esto contradice lo dicho anteriormente sobre la velocidad a la que debe hacerse la traducción y parece un contrasentido: para acceder a memoria hay que traducir la dirección virtual, lo que supone realizar un acceso a memoria por cada nivel que tenga la tabla de páginas. Esto representa un retardo inadmisible en los accesos a memoria. Para solventar este problema, se dota a la MMU de una unidad denomina **TLB** (*Translation Look-aside buffer*), que analizamos seguidamente.

El que la MMU acceda a la tabla de páginas implica que debe conocer su estructura así como el formato concreto que las entradas de las tablas de 1er y 2º nivel. Esto significa que la estructura de la tabla de páginas está grabada en el *hardware*. El sistema operativo debe adaptarse a dicha estructura que es inmutable para cada tipo de MMU.

Excepciones generadas por la MMU

Cuando la MMU encuentra una situación anómala genera una excepción. Estas excepciones no siempre se refieran a errores, pueden ser avisos. El sistema operativo entra a ejecutar cuando se produce una de ellas, tratándola. Las situaciones más corrientes son las siguientes:

- Acceso a una dirección que no pertenece a ninguna de las regiones asignadas. Esta es una situación de error, por lo que el sistema operativo enviará una señal al proceso, que suele matarlo.
- Acceso a una dirección correcta pero con un tipo de acceso no permitido. En general esta es una situación de error, pero en algunos casos es simplemente un aviso. Para los casos de error la acción del sistema ope-

rativo consiste en enviar una señal al proceso, que suele matarlo. Las situaciones de aviso son las siguientes.

- Desbordamiento de pila. Para detectar el desbordamiento se pone la última página de la pila como de lectura solamente. Al crecer la pila, cuando se intente escribir en dicha página, la MMU generará una excepción de tipo de acceso a memoria incorrecto. El sistema operativo observa que se intenta escribir al final de la región de pila y, si puede, aumentará automáticamente dicha región. En caso contrario enviará una señal al proceso, que suele matarlo.
- Escritura en un página COW. Como se verá más adelante, las páginas en COW se marcan como de solo escritura. Cuando se intenta escribir en una de estas páginas, la MMU generará una excepción de tipo de acceso a memoria incorrecto. El sistema operativo observa que se intenta escribir en un página COW, por lo que desdobla dicha página.

- Fallo de página. Se trata de un aviso, para que el sistema operativo asigne un marco de página a dicha página, actualizando, además, la correspondiente tabla de páginas.

TLB (Translation Look-aside buffer)

La **TLB** es una memoria asociativa muy rápida que almacena las parejas página-marco de las páginas a las que se ha accedido recientemente. La TLB es, pues, una cache especializada para la tabla de páginas capaz de albergar del orden de 128 a 512 elementos. En la mayoría de los casos la MMU no accederá a la memoria principal, al encontrar la información de traducción en la TLB. Solamente cuando hay un fallo en la TLB debe acceder a la memoria principal. La velocidad y la tasa de aciertos de la TLB deberán ser lo suficientemente elevadas para que el tiempo medio efectivo de traducción sea pequeño, comparado con el tiempo de acceso a memoria principal (o del tiempo de acceso de su memoria cache, en caso de existir).

La MMU trata los fallos de la TLB accediendo a memoria principal y sustituyendo una de las parejas página-marco de la misma por la nueva información. De esta forma, se va renovando la información de la TLB con los valores de interés en cada momento.

Un problema inherente a la TLB es que las parejas página-marco que almacena pertenecen a un programa en ejecución, y deben ser anuladas al pasar a ejecutar otro programa. La técnica más utilizada es invalidar las parejas página-marco cuando se pasa de un programa a otro. Esto implica que la recarga de la TLB se realiza a base de fallos en la TLB. Sin embargo, hay computadores que disponen de instrucciones máquina para leer y cargar la TLB. Estas instrucciones máquina solamente se pueden ejecutar en nivel privilegiado, y el sistema operativo las utiliza para cargar la TLB, cuando pone en ejecución un programa, y para salvarla, cuando lo quita.

Finalmente, destacaremos que la encargada de marcar las páginas como sucias suele ser la MMU. En efecto, al mismo tiempo que hace la traducción de la dirección, en caso de que el acceso sea de escritura, marca esa página como sucia. Por razones de velocidad la marca se realiza en la TLB. Más tarde, esta información se pasa a la tabla de páginas, para que el sistema operativo sepa que la página está sucia y lo tenga en cuenta a la hora de hacer la migración de páginas. En algunas máquinas, sin embargo, esta operación la realiza el sistema operativo, limitándose la MMU a generar una excepción *hardware* síncrona cuando escribe en una página que tiene su bit de modificado desactivado.

Para dejar al sistema operativo plena libertad para organizar la tabla de páginas según le convenga, en algunas máquinas la MMU accede solamente a la TLB y no a memoria principal. Cuando la MMU encuentra un fallo de TLB genera una excepción *hardware* síncrona, encargándose el sistema operativo de acceder a la tabla de páginas y de actualizar, en su caso, la TLB. Esta solución es más lenta pero más flexible.

Operación en modo real

Como se ha indicado, el sistema operativo es el encargado de crear y mantener las tablas de páginas para que la MMU pueda hacer su trabajo de traducción. Ahora bien, cuando arranca el computador no existen tablas de páginas, por lo que la MMU no puede realizar su función. Para evitar este problema la MMU incorpora un modo de funcionamiento denominado real, en el cual se limita a presentar la dirección recibida del procesador a la memoria principal. En modo real el computador funciona, por tanto, como una máquina carente de memoria virtual.

Tratamiento del fallo de página

La paginación está dirigida por la ocurrencia de fallos de página, que indican al sistema operativo que debe traer una página de *swap* a un marco, puesto que un proceso la requiere (esto es lo que se denomina paginación por demanda). A continuación, se especifican los pasos típicos en el tratamiento de un fallo de página:

- La MMU produce una excepción, dejando, habitualmente, en un registro especial la dirección que provocó el fallo.
- Se activa el sistema operativo, que accede a la tabla de páginas para buscar la dirección que produjo el fallo. Si ésta no pertenece a ninguna página asignada o si el acceso no es del tipo permitido, se aborta el proceso o se le manda una señal. En caso contrario, se realizan los pasos que se describen a continuación. Evidentemente, si la dirección es del sistema y el proceso estaba en modo usuario cuando se produjo el fallo, el acceso es también inválido.
- Se consulta la tabla de marcos para buscar uno libre.

- Si no hay un marco libre, se aplica el algoritmo de reemplazo para seleccionar la página que se expulsará. El marco seleccionado se desconectará de la página a la que esté asociado, activando el bit de ausente en la entrada correspondiente. Si la página está modificada, se copia al disco:
  - Si la página pertenece a una región de tipo compartida y con soporte en fichero, hay que escribirla en el bloque correspondiente del fichero.
  - En el resto de los casos, hay que escribirla en *swap*, almacenándose en la entrada de la tabla de páginas la dirección de *swap* que contiene la página.
- Una vez que se obtiene el marco libre, ya sea directamente o después de una expulsión, se inicia la carga de la nueva página sobre el marco y, al terminar la operación, se rellena la entrada correspondiente a la página para que esté marcada como válida y apunte al marco utilizado.

Téngase en cuenta que, en el peor de los casos, un fallo de página puede causar dos operaciones de entrada/salida al disco. En contraste, puede no haber ninguna operación sobre el disco si hay marcos libres o la página expulsada no está modificada y, además, la página que causó el fallo está marcada como rellenar a ceros.

4.3. NIVELES DE GESTIÓN DE MEMORIA

En la gestión de la memoria se pueden distinguir los tres niveles siguientes:

- **Nivel de procesos**. En este nivel se determina cómo se reparte el espacio de memoria entre los procesos existentes. Se trata de un nivel gestionado por el sistema operativo.
- **Nivel de regiones**. Establece cómo se reparte el espacio asignado al proceso entre las regiones del mismo. Nuevamente, es un nivel manejado por el sistema operativo.
- **Nivel de datos dinámicos**. Como se verá a lo largo del capítulo, existen algunas regiones, como la región de datos dinámicos o *heap* y la pila, que mantienen en su interior diversos bloques de información creados dinámicamente por el programa. Por tanto, se requiere una estrategia de gestión que determine cómo se asigna y recupera el espacio de la región para dar un adecuado soporte a esos bloques de información. Este nivel está gestionado por el propio lenguaje de programación.

El **SO** tiene una visión **macroscópica** de la memoria de un proceso, consistente en la imagen de memoria y en las regiones que componen dicha imagen de memoria. Por lo que se centra en los niveles de procesos y de regiones.

Por el contrario, el **programa** tiene una visión **microscópica** de la memoria, consistente en variables y estructuras de datos declarados en el programa. Centrándose en el nivel de datos dinámicos.

Las bibliotecas del lenguaje utilizado en el desarrollo del programa gestionan el espacio disponible en la región de datos dinámicos y solamente llaman al SO cuando tienen que variar el tamaño de la región, o crear una nueva región. Por lo tanto, crear una variable dinámica que quepa en la correspondiente región no implica al SO. Sin embargo, crear una variable dinámica que no quepa en la región de datos dinámicos obliga a llamar al SO para que aumente dicha región, o cree una nueva.

4.3.1. Operaciones en el nivel de procesos

En el nivel de procesos se realizan operaciones vinculadas con la gestión del mapa de memoria, que son las siguientes:

- **Crear la imagen de memoria del proceso**. Antes de comenzar la ejecución de un programa, hay que iniciar su imagen de memoria tomando como base el fichero ejecutable que lo contiene. En UNIX se trata del servicio `exec`, mientras que en Windows es `CreateProcess`, ya estudiados en el capítulo "3 Procesos".
- **Eliminar la imagen de memoria del proceso**. Cuando termina la ejecución de un proceso, hay que liberar todos sus recursos, entre los que está su imagen de memoria. El servicio `exec` de UNIX, requiere liberar la imagen actual del proceso antes de crear la nueva imagen basada en el ejecutable especificado.
- **Duplicar la imagen de memoria del proceso**. El servicio `fork` de UNIX crea un nuevo proceso cuya imagen de memoria es un duplicado de la imagen del padre.

Para completar este nivel, y aunque no tenga que ver directamente con la solicitud y liberación de espacio, falta una operación genérica adicional asociada al cambio de contexto:

- **Cambiar de imagen de memoria de proceso**. Cuando se produce un cambio de contexto, habrá que activar de alguna forma la nueva imagen y desactivar la anterior.

4.3.2. Operaciones en el nivel de regiones

En el nivel de regiones, las operaciones están relacionadas con la gestión de las regiones.

- **Crear una región para un proceso**. Esta operación será activada al crear la imagen del proceso, para crear las regiones iniciales de la misma. Además, se utilizará para ir creando las nuevas regiones que aparezcan según se va ejecutando el proceso.
- **Eliminar una región de un proceso**. Al terminar un proceso hay que eliminar todas sus regiones. Además, hay regiones que desaparecen durante la ejecución del proceso. Es necesario recuperar los espacios de memoria o de *swap* asignados a la región.
- **Cambiar el tamaño de una región**. Algunas regiones, como la pila y el *heap*, tienen un tamaño dinámico que evoluciona según lo vaya requiriendo el proceso.
- **Duplicar una región de la imagen de un proceso en la imagen de otro** (*duplicar_región*). El duplicado de la imagen asociado al servicio `fork` requiere duplicar cada una de las regiones del proceso padre.
- **Compartir una región**. Compartir una región entre dos o más procesos exige que todos ellos puedan acceder a la misma región de memoria.

En sistemas con memoria real se asigna y memoria principal. Sin embargo, en sistemas con memoria virtual se asigna lo más económico, es decir, *swap* o rellenar con 0. No se asignan marcos de memoria principal

Características de una región

Una región está formada por una serie de direcciones contiguas del mapa de memoria y alberga un determinado tipo de información. La región está definida por una dirección de comienzo y un tamaño. Dividir la imagen de memoria en varias regiones permite ajustar las características de la región a las necesidades de la información que contiene. Por ejemplo, dado que el código no se modifica, la región que lo contenga deberá ser de tamaño fijo y no deberá tener permiso de escritura.

Las principales características de una región son las siguientes:

- **Fuente.** La fuente es lugar donde se almacena el valor inicial de la información de la región. Existen dos fuentes principales: el fichero ejecutable y rellenar a ceros, esto último cuando no existe valor inicial de la información.
- **Compartida/privada** La región puede ser compartida entre varios procesos o privada.
- **Protección**. La región puede tener los niveles de protección típicos de lectura, escritura y ejecución (RWX).
- **Tamaño**. La región puede ser de tamaño fijo o variable.

4.3.3. Operaciones en el nivel de datos dinámicos

Las operaciones identificadas en esta sección sólo son aplicables a las regiones que almacenan internamente múltiples datos creados dinámicamente, como es el caso del *heap* o de la pila. En este nivel se pueden distinguir las siguientes operaciones relativas al *heap*:

- **Reservar memoria**. En el caso del *heap* y tomando como ejemplo el lenguaje C, se trataría de la función `malloc`. En C++ y Java, sería el operador `new`.
- **Liberar memoria reservada**. En este caso, sería la función `free` del lenguaje C, mientras que en C++ correspondería a la operación `delete`. En Java la liberación se realiza de forma automática, mediante un mecanismo de recolección de basura.
- **Cambiar el tamaño de la memoria reservada**. La función `realloc` de C cumpliría esta función.

La reserva de espacio en la pila se hace por el propio programa al crear el RAR (registro de activación de rutina) en la llamada a una función o procedimiento. En un sistema con memoria virtual, el crecimiento de la pila lo hace automáticamente el sistema operativo mediante el siguiente truco: La última página de la región de pila se asigna sin permiso de escritura. Si, al ir creciendo la pila, se intenta escribir en dicha página, significa que ya queda poca pila, por lo que el sistema operativo, si puede, aumentará dicha región.

4.4. ESQUEMAS DE GESTIÓN DE LA MEMORIA DEL SISTEMA

En esta sección se analiza cómo se reparte la memoria del sistema entre los distintos procesos que están ejecutándose en un momento dado. Consideraremos las dos situaciones de memoria real y memoria virtual, analizando los siguientes esquemas de gestión de memoria:

- Memoria real
  - Asignación contigua
  - Segmentación
- Memoria virtual
  - Paginación
  - Segmentación paginada

Dado que todos los procesadores de propósito general actuales incluyen una unidad de gestión de memoria que soporta memoria virtual, los sistemas operativos actuales de propósito general utilizan exclusivamente gestión de memoria basada en memoria virtual. Solamente en sistemas muy sencillos o especiales se aplica la asignación contigua o la segmentación.

4.4.1. Asignación contigua

Si bien la asignación contigua podría utilizarse tanto en sistemas de memoria real como virtual, solamente se justifica su utilización en sistemas de memoria real. En este caso, el proceso recibe una única **zona contigua de memoria principal**, delimitada por una dirección de comienzo y un tamaño o una dirección de final, como muestra la figura 4.14.

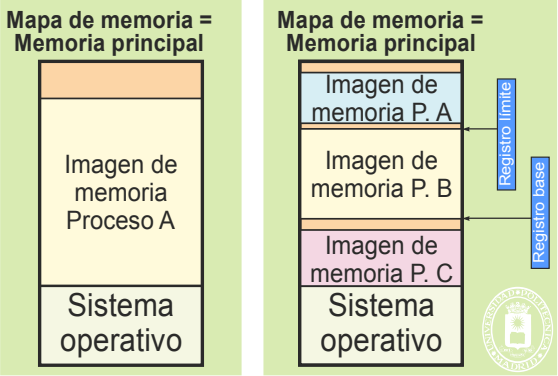


Figura 4.14 *La imagen de memoria de un proceso en una máquina con memoria real se alberga en una zona contigua de memoria principal. La figura muestra los dos casos de sistema monoproceso y multiproceso. Los registros base y límite delimitan el espacio asignado al proceso en ejecución en un momento determinado.*

Para el caso de un sistema **monoproceso** la memoria principal se divide entre la que ocupa el sistema operativo y la disponible para el proceso.

En el caso de un sistema **multiproceso**, el gestor de memoria debe buscar un hueco suficientemente grande para albergar al proceso.

Las operaciones de crear y duplicar una imagen requieren buscar una zona libre de la memoria principal donde quepa la imagen. Eliminar la imagen consiste en marcar como libre la zona de memoria asignada a esa imagen.

Dado que, al crear una imagen, hay que buscar un hueco de memoria principal libre en el que quepa, se produce un nuevo hueco con el espacio sobrante. Esto conlleva la pérdida de memoria denominada por **fragmentación externa**, puesto que quedarán huecos pequeños en los que no quepan nuevos procesos y que, por lo tanto, quedan sin utilizarse. Este problema podría solventarse haciendo lo que se llama una **compactación** de memoria, consistente en realojar todos los procesos de forma que queden contiguos, dejando un solo hueco al final. Sin embargo, esta es una operación muy costosa, por lo que no suele utilizar.

Protección de memoria

La protección de memoria se implementa mediante un sencillo *hardware* que incluye:

- Una pareja de registros, denominados de forma genérica **registros valla**, que delimitan la menor y mayor dirección de memoria principal que puede utilizar en proceso. Típicamente al registro que delimita la menor dirección se le denomina **registro base** y al que delimita la dirección superior se le denomina **registro límite**. El sistema operativo asigna y mantiene los valores base y límite de cada proceso.
- Un circuito comparador que comprueba cada dirección de memoria generada por el proceso contra dichos registros. En caso de sobrepasarse dichas direcciones este circuito genera una excepción de violación de memoria.

La operación de **cambio de contexto** requiere cambiar los valores contenidos en los registros base y límite, que delimitan el espacio de memoria asignado al proceso en ejecución.

Soporte de regiones

Este sistema no presenta ninguna funcionalidad para el soporte de las regiones del proceso. Todas las regiones han de residir en la única zona de memoria principal asignada al mismo. La figura 4.15 muestra el esquema que se ha usado típicamente en los sistemas UNIX con memoria real.

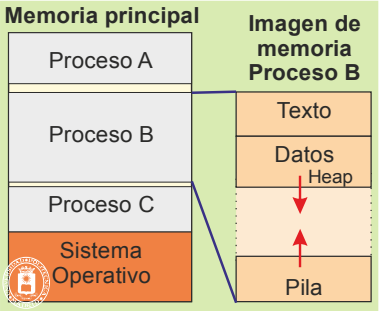


Figura 4.15 *Regiones de texto, datos y pila de un proceso UNIX típico en memoria real. El* heap *se encuentra incluido en la región de datos.*

Crecimiento de la imagen de memoria

Para que las regiones de datos y de pila puedan crecer se deja un espacio libre entre ellos y se hace que crezcan en sentidos opuestos. Ese espacio libre está soportado por memoria principal, con el coste correspondiente. Si el espacio libre es muy grande se desperdicia memoria principal, sin embargo, si es demasiado pequeño las regiones de datos y pila pueden chocar. En este caso, habría que buscar un hueco de memoria libre suficientemente grande y realojar al proceso. Esta operación es compleja y costosa, por lo que la opción más utilizada es la de matar al proceso.

La creación de nuevas regiones se debería hacer en el espacio libre entre datos y pila, lo que no suele estar contemplado en estos sistemas.

Compartir memoria

La asignación contigua no se presta a que los procesos compartan memoria, puesto que los procesos deberían ser colindantes de forma que compartiesen una porción de memoria principal.

4.4.2. Segmentación

La segmentación es una extensión del sistema contiguo, en el que se le asignan al proceso **varias zonas o segmentos de memoria principal contigua**, típicamente, una por región.

Las operaciones de crear y duplicar una imagen requieren buscar, para cada región del proceso, una zona libre de la memoria principal donde quepa dicha región.

Protección de memoria

La protección de memoria exige tener una pareja de registros valla por segmento, como muestra la figura 4.16. Por tanto, el *hardware* de soporte de la segmentación contendrá una serie de registros cuyo contenido debe, en cada momento, corresponder con los segmentos asignados al proceso que está ejecutando. El proceso podrá tener como máximo tantos segmentos como parejas de registros valla tenga dicho *hardware*.

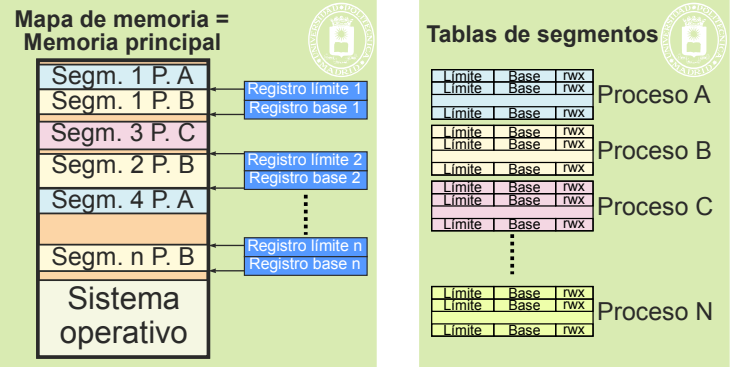


Figura 4.16 *La segmentación permite que el proceso tenga un segmento por cada pareja de registros valla que tenga el procesador. El sistema operativo mantiene una tabla de segmentos en los que almacena los registros valla de todos los procesos.*

El conjunto de los registros valla se almacenan en la tabla de segmentos de cada proceso. Estas tablas son creadas y mantenidas por el sistema operativo. Si el *hardware* lo soporta, se pueden incluir para cada segmento unos bits de protección. De esta forma el *hardware* puede comprobar no solamente si la dirección es válida sino, además, si el tipo de acceso a memoria está permitido. Puede haber un bit para permitir lectura, otro para permitir escritura y un tercero para permitir ciclo *fetch*, es decir, ciclo de acceso a instrucción de máquina para ejecutarla.

La operación de **cambio de contexto** requiere copiar en todas las parejas de registros valla los valores almacenados en la tabla de segmentos del proceso.

Soporte de regiones

Este sistema se adecua a las necesidades de memoria de los procesos puesto que se puede asignar cada región de la imagen de memoria a un segmento diferente, que tenga justo el tamaño que necesita la región.

Presenta el problema de pérdida de memoria por fragmentación externa, introducido anteriormente.

Compartir memoria

Los procesos pueden fácilmente compartir memoria, puesto que basta con que el proceso PA tenga en sus registros de valla Rp los mismos valores que el proceso PB tenga en sus registros valla Rq. Es de destacar que el segmento compartido puede ser, por ejemplo, el 2 del proceso PA y el 4 del proceso PB.

Las direcciones con las que los procesos acceden al segmento compartido son las mismas, puesto que se trata siempre de direcciones de memoria principal. Por otro lado, los permisos de acceso pueden ser distintos para cada proceso, puesto que se establecen en su propia tabla de segmentos.

Crecimiento de la imagen de memoria

En el caso de que las necesidades de memoria de uno de los segmentos del proceso crezcan por encima de la memoria asignada, habría que buscar un hueco de memoria libre suficientemente grande y realojar al segmento (por supuesto, si el segmento tuviese memoria libre contigua no haría falta realojarlo).

4.4.3. Memoria virtual. Paginación

El esquema de gestión basado en memoria virtual es el utilizado prácticamente por la totalidad de los sistemas operativos de propósito general actuales, como Windows o Linux.

La creación y duplicación de una imagen requiere la creación de la correspondiente tabla de páginas y la creación de las regiones que tenga la imagen. Eliminar la imagen consiste en eliminar la correspondiente tabla de páginas, recuperando el espacio que ocupa. Además, hay que eliminar las regiones que componen la imagen, recuperando los recursos de almacenamiento que tengan asignados.

El cambio de contexto consiste en cambiar el valor almacenado en el registro RIED, de forma que apunte a la tabla de páginas del nuevo proceso.

Asignación de recursos de almacenamientos

En el caso de la memoria virtual solamente se dota de recursos de almacenamiento a las páginas asignadas a cada proceso. El resto del espacio de direcciones no tiene soporte alguno, por tanto, no gasta recursos del computador. En concreto, el recurso de almacenamiento de una página puede ser uno de los tres siguientes:

- **Marco de memoria**. Cuando la página se lleva a memoria ocupa un marco de memoria principal.
- **Página de intercambio**. Cuando la página reside en el disco.
- **Rellenar a ceros o página anónima**. Una página sin contenido inicial, como puede ser una página de una zona de la pila que todavía no se ha utilizado, no tiene asignado recurso de almacenamiento hasta que el proceso escribe en ella por primera vez. En ese momento se le asigna un marco de memoria, que deberá ser totalmente borrado, para evitar que pueda leer información dejada por otro proceso que previamente utilizó dicho marco.

Protección de memoria

La protección de memoria la realiza la MMU, utilizando para ello la información contenida en la tabla de páginas activa en cada momento. Véase en la figura 4.12, página 153, cómo realiza la MMU la conversión de la dirección virtual a dirección real.

Esta protección es muy completa puesto que, además de detectar que el acceso se realiza sobre una dirección válida, es decir, sobre una página asignada al proceso, se comprueba si el tipo de acceso está permitido para esa página.

Soporte de regiones

Este sistema se adecua a las necesidades de memoria de los procesos, puesto que se puede asignar cada región de la imagen de memoria a una entrada de la tabla de páginas de primer nivel. Como muestra la figura 4.17, se puede tener un gran número de regiones, cada una con sus propios bits protección de acceso y con grandes espacios libres entre ellas.

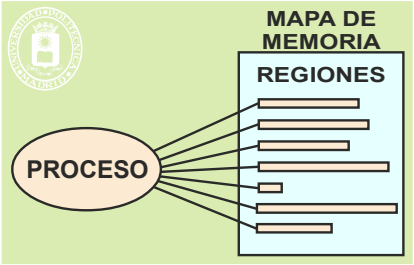


Figura 4.17 *En sistemas con memoria virtual hay una gran flexibilidad para asignar regiones, que no tienen que ser contiguas, puesto que el espacio libre entre ellas no ocupa recursos físicos (ni marcos de página, ni swap)*

Destacaremos que el tamaño de los bloques en un sistema paginado es siempre un **múltiplo del tamaño de la página**, por lo que a cada región es necesario asignarle un número entero de páginas (véase figura 4.18). Evidente-

mente, es difícil que el segmento tenga justamente ese tamaño, por lo que la última página del segmento no estará completamente llena. La pérdida de memoria que ello conlleva se denomina pérdida por **fragmentación interna**.

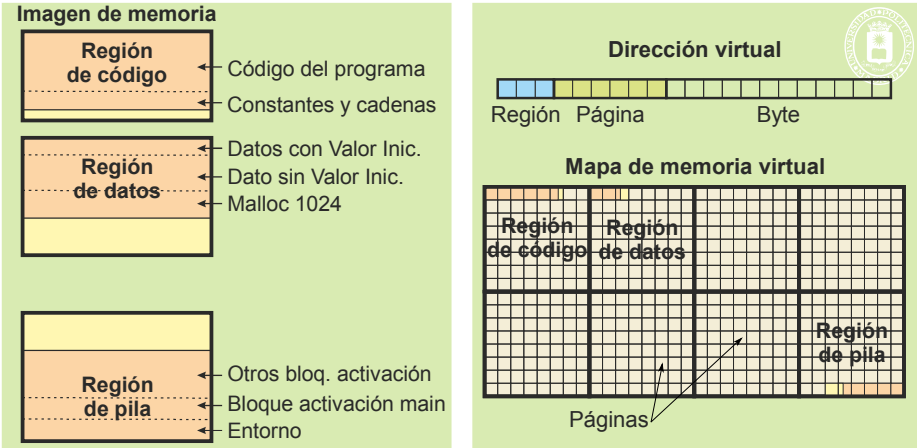


Figura 4.18 *Las regiones se componen de un número entero de páginas, lo que produce fragmentación interna. La tabla de primer nivel permite dividir el mapa de memoria en grandes fragmentos que sirven para soportar las regiones del proceso.*

Compartir memoria

En un sistema con memoria virtual los procesos pueden fácilmente compartir memoria. En efecto, como muestra la figura 4.19, basta con que los procesos compartan una subtabla de segundo nivel.

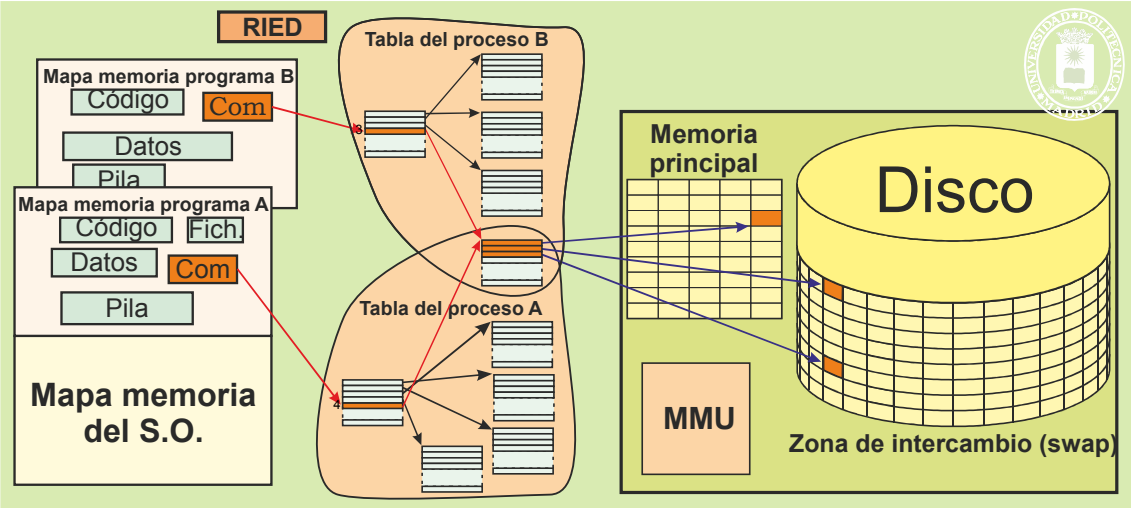


Figura 4.19 *En un sistema con memoria virtual, para que varios procesos compartan memoria basta con que compartan una subtabla de páginas de 2º nivel.*

Al compartir una subtabla de segundo nivel comparten los mismos recursos de almacenamiento, por lo que si uno escribe en una página, los demás pueden leer lo que éste escribió. Sin embargo, un detalle muy importante es que los procesos pueden referirse a la subtabla compartida a través de entradas distintas de sus tablas de primer nivel (en la figura, el proceso A utiliza la entrada 4, mientras que el proceso B utiliza la 3). Esto significa que las direcciones bajo las cuales los procesos ven la zona compartida pueden ser diferentes para cada uno de ellos.

El sistema operativo tiene que mantener un **contador de referencias** con el número de procesos que comparten la región. Dicho contador se incrementa por cada nuevo proceso que comparta la región y se decrementa cuando uno de esos procesos muere o libera la región. Solamente cuando el contador llega a 0 significa que la región ya no la utiliza nadie, por lo que se pueden recuperar los marcos y páginas de intercambio asignados a la región, así como la propia tabla de páginas.

La figura 4.20 muestra cómo la zona compartida está en posiciones diferentes en los procesos PA y PB. Esto presenta un problema de **autorreferencia** puesto que si el proceso PA almacena en **b** la dirección absoluta de **a** (por ejemplo, porque es un puntero a un elemento de una cadena) almacenará X + 16. Si, seguidamente, el proceso PB desea utilizar dicha cadena, encontrará en **b** la dirección X + 16, que no es una dirección de la zona compartida, y que, incluso, puede no ser una dirección permitida para él. Por tanto, en las zonas de memoria compartida han de usarse exclusivamente direcciones relativas a posiciones de la propia zona. Para el ejemplo, habría que guardar en **b** simplemente 16, con lo que, para acceder correctamente al elemento **a**, cada proceso sumará la dirección de comienzo de la zona compartida.

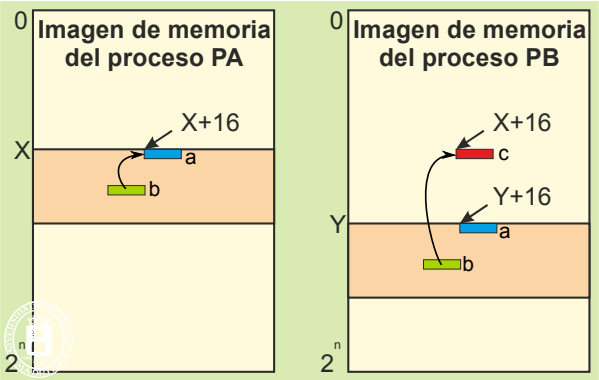


Figura 4.20 *Las direcciones en las que cada proceso ve la zona de memoria compartida suelen ser distintas, por lo que aparece el problema de la autorreferencia.*

Crecimiento de la imagen de memoria

La imagen de memoria puede crecer fácilmente en un sistema de memoria virtual. El crecimiento se puede hacer de las dos formas siguientes:

- Se puede añadir un nuevo bloque de memoria, para lo cual utilizará una nueva entrada de la tabla de primer nivel y se añadirá la correspondiente subtabla de 2º nivel.
- Se puede ampliar un bloque existente. Para ello, basta con añadir nuevas entradas en su correspondiente subtabla de 2º nivel.

La ampliación de memoria implica asignar los recursos de almacenamiento correspondientes. En general, se asignará el recurso más económico. Si la ampliación no tiene valor inicial, se asignarán páginas a rellenar a cero (por ejemplo, cuando se amplía el *heap* o la pila). Si existe valor inicial se asignarán páginas de intercambio.

Para que los bloques puedan crecer se deben ubicar adecuadamente separadas, de forma que quede espacio entre ellas (espacio, que recordamos, no consume recursos de almacenamiento).

4.4.4. Segmentación paginada

La segmentación paginada consiste en emplear los dos mecanismos descritos anteriormente. Existe un primer nivel de definición de segmentos, y un segundo nivel de paginación, estando cada segmento paginado de forma individual. La figura 4.21 muestra esta secuencia.

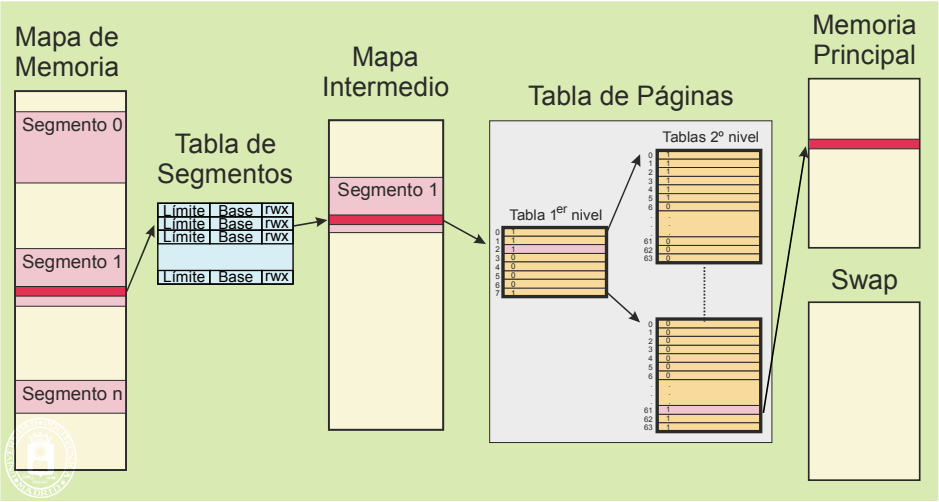


Figura 4.21 *En la segmentación paginada se producen dos traducciones de dirección. Primero se aplica el nivel de segmentación y luego se aplica paginación.*

Conceptualmente, la segmentación paginada no aporta gran cosa sobre el modelo paginado con tabla de dos o más niveles analizado en el apartado anterior, puesto que la tabla de primer nivel se puede considerar equivalente a la tabla de segmentos.

Pocos procesadores implementan este doble mecanismo de traducción de direcciones y los sistemas operativos no hacen uso del mismo para mejorar su portabilidad. La arquitectura clásica de Intel x86 emplea segmentación paginada por razones históricas y de compatibilidad hacia modelos más antiguos. Intel introduce la segmentación en el año 1978 con el 8086 y en el año 1985 con el 80386 introduce la segmentación paginada. La arquitectura de 64 bits x64 Intel considera obsoleta esta solución.

4.5. CICLO DE VIDA DE UN PROGRAMA

En general, las aplicaciones se desarrollan utilizando lenguajes de alto nivel, dividiendo la funcionalidad en varios módulos (ficheros de código fuente), para facilitar, de esta manera, un desarrollo incremental, el mantenimiento y la reutilización del código.

En este contexto, una biblioteca es una colección de módulos objeto relacionados entre sí y preparados para su utilización. Las bibliotecas pueden ser desarrolladas por el propio usuario, por terceros.

Una vez preparados los módulos fuente, éstos deben pasar por varias fases, como muestra la figura 4.22, hasta que se carga el ejecutable en memoria formando la imagen del correspondiente proceso:

- Compilación.
- Montaje o enlace.
- Carga.
- Ejecución.

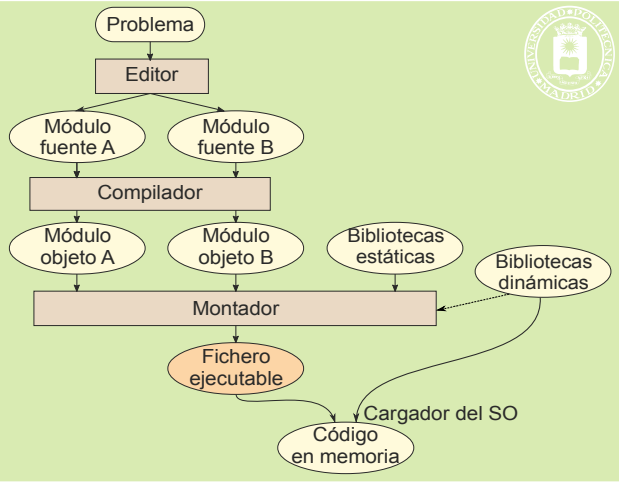


Figura 4.22 *Fases en el procesamiento de un programa.*

Punto de enlace

Una llamada a procedimiento se realiza mediante la pareja de instrucciones de máquina CALL y RET (véase figura 4.23). La instrucción CALL ha de tener la dirección de memoria donde comienza el procedimiento. Llamaremos **punto de enlace** a esa dirección de comienzo del procedimiento.

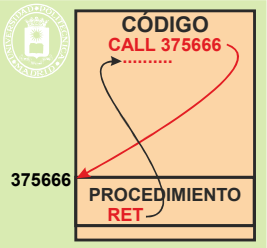


Figura 4.23 *La llamada a procedimiento requiere las instrucciones de máquina CALL y RET.*

Compilación

El compilador se encarga de procesar de forma independiente cada fichero fuente, generando el correspondiente fichero objeto. El compilador es capaz de resolver todos los puntos de enlaces de los procedimientos incluidos en el módulo, pero deja sin resolver las referencias a procedimientos y datos externos.

La información que contiene el fichero objeto es la siguiente:

- **Cabecera** que contiene información de cómo está organizada el resto de la información.
- **Texto**: Contiene el código y las constantes del módulo. Se prepara para ubicarlo en la dirección 0 de memoria.
- **Datos con valor inicial**: Almacena el valor de todas las variables estáticas con valor inicial.
- **Datos sin valor inicial**. Se corresponde con todas las variables estáticas sin valor inicial.
- **Información de reubicación**. Información para facilitar la labor del montador.
- **Tabla de símbolos**. Incluye referencias a todos los símbolos externos (datos o procedimientos) que se utilizan en el módulo pero que están declarados en otros módulos, por lo que no se ha podido hacer su enlace. Igualmente incluye las referencias exportadas.
- **Información de depuración**. Se incluye si el módulo se compila para depuración.

Montaje

El resultado del montaje es la generación de un fichero ejecutable. El montaje requiere realizar las siguientes operaciones:

- Reubicar los módulos de acuerdo a la posición asignada dentro del código total.
- Resolver las referencias externas de cada módulo.
- Agrupar todos los todos los ficheros objeto que forman parte de la aplicación. Se incluirán o no las bibliotecas utilizadas, dependiendo de que el montaje sea estático o dinámico.
- En caso de montaje dinámico, preparar la tabla de símbolos para el acceso a las bibliotecas dinámicas.

El montaje puede ser estático o dinámico, alternativas que se analizan seguidamente.

Montaje estático

En el montaje estático se crea un ejecutable autocontenido, que tiene resueltos todos los puntos de enlace. La figura 4.24 muestra esta solución.

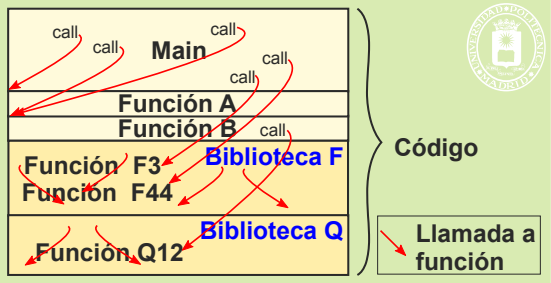


Figura 4.24 *En el montaje estático el ejecutable incluye todos los módulos desarrollados así como todas las bibliotecas utilizadas.*

Los mayores inconvenientes de esta solución son los siguientes:

- Los ejecutables son grandes puesto que se incluyen al completo las bibliotecas usadas, ocupando mucho espacio de disco. Parecería lógico incluir solamente las funciones realmente usadas, pero esto no es fácil, puesto que las funciones de las bibliotecas suelen tener muchas dependencias a otras funciones de la propia biblioteca.
- Imágenes de memoria con grandes trozos iguales (las bibliotecas) pero repetidos.
- La actualización de la biblioteca implica volver a montar cada programa que las usa. Esto también se puede ver como una ventaja: la actualización de la biblioteca no afecta a los programas ya preparados, que pueden ser incompatibles con la nueva versión.

Montaje dinámico

En el montaje dinámico las bibliotecas no se incluyen en el ejecutable. El ejecutable no es un programa completo, por lo que al crear la imagen de memoria del proceso hay que añadir las bibliotecas dinámicas, resolviendo, además, los puntos de enlace

Las ventajas del montaje dinámico son las siguientes:

- Menor tamaño de los ejecutables, puesto que las bibliotecas no se incluyen.
- Las bibliotecas sólo están almacenadas en el disco una vez, ocupando menos espacio.
- Los códigos de las bibliotecas se incluyen en la imagen de memoria como regiones compartidas, por lo que se emplean menos marcos de página para la ejecución de los procesos que las comparten.
- Se produce una actualización automática de las bibliotecas en los programas que las usan. Esto a veces da lugar a problemas porque la nueva versión puede no ser compatible con el programa. Por eso, puede ser necesario mantener en el sistema más de una de versión de la biblioteca.

Los inconvenientes más importantes son los siguientes:

- Como ya se ha indicado, pueden existir bibliotecas del mismo nombre incompatibles (versiones incompatibles).
- Si no se lleva un control muy estricto de las bibliotecas que se utiliza cada aplicación, puede ser problemática la eliminación de bibliotecas obsoletas.
- Existe una sobrecarga en tiempo de ejecución, puesto que hay que completar las funciones de montaje resolviendo los puntos de enlace relativos a las funciones de la biblioteca.

Existen las tres alternativas de montaje siguientes:

- Montaje automático **en tiempo de carga** en memoria. Se montan todas las bibliotecas en el momento inicial de carga del programa. Esta solución hace más lenta la carga del programa y puede incluir el montaje de bibliotecas que no se lleguen a utilizar en esa ejecución del programa.
- Montaje automático **al invocar el procedimiento**. Se monta la biblioteca la primera vez que el programa hace uso de una función de la misma. La carga introduce un pequeño retardo en la ejecución del programa, mientras se hace dicho montaje (véase figura 4.25).
- Montaje **explícito**. En esta solución el programa utiliza servicios para solicitar al sistema operativo el montaje de la biblioteca.

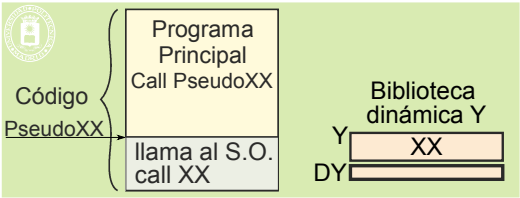


Figura 4.25 *Carga al invocar el procedimiento XX de la biblioteca Y. El código incluye una función PseudoXX. Lo primero que hace PseudoXX es comprobar si la biblioteca Y está montada. En caso negativo solicita el montaje al sistema operativo. Seguidamente llama a XX con los mismos parámetros con los que se llamó a PseudoXX.*

Carga y ejecución

La carga consiste en generar la imagen del proceso a partir del fichero ejecutable, así como de las bibliotecas dinámicas en caso de montaje dinámico. en tiempo de carga

Una vez cargado, el programa se ejecutará cuando el planificador del sistema operativo le asigne el procesador.

La figura 4.26 muestra las dos regiones que requiere cada biblioteca dinámica: una región de texto y una región de datos, que incluye los datos son y sin valor inicial, pero no incluye heap. La región de código de la biblioteca dinámica puede ser compartida, mientras que la región de datos de la biblioteca dinámica ha de ser privada a cada proceso que utiliza la biblioteca.

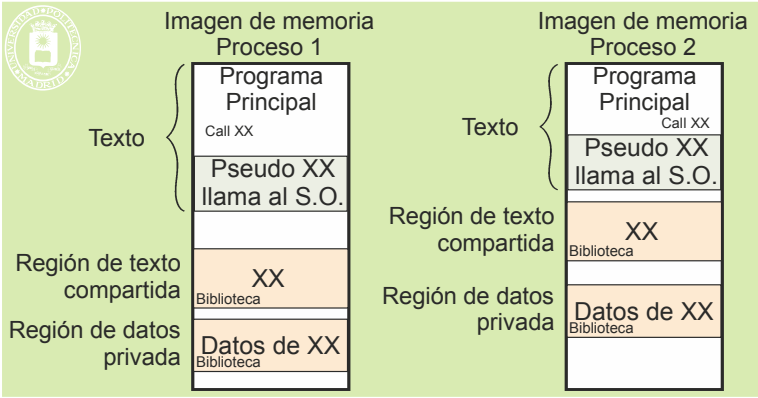


Figura 4.26 *Cada biblioteca dinámica requiere dos regiones: una de texto y otro de datos, que incluye los datos son y sin valor inicial, pero no incluye* heap. *La región de texto de la biblioteca estará en direcciones distintas en cada proceso que las comparte.*

Ejemplo

Vamos a considerar el siguiente sencillo programa, almacenado en el fichero `mi_cos.c`:

```
#include <stdio.h>
#include <math.h>
double coseno(double v) {
  return cos(v);
}
int main() {
  double a, b = 2.223;
  a = coseno(b);
  printf("coseno de %lf:%.f\n",b,a);
  return 0;
}
```

Si compilamos y montamos dicho programa con compilación estándar, es decir, dinámica con el mandato `gcc mi_cos.c -lm -o cos_dyn.exe`, obtenemos el ejecutable que hemos llamado `cos_dyn.exe`.

Con el mandato `nm cos_dyn.exe` obtenemos los símbolos de dicho programa. Entre ellos encontramos los nombres de las funciones, obteniendo el siguiente resultado:

```
         U cos@@GLIBC_2.0
080484cb T coseno
080484f0 T main
         U printf@@GLIBC_2.0
```

Observamos que obtenemos las direcciones de las funciones coseno y main, declaradas en nuestro programa: la T (Global text symbol) significa que forman parte del segmento de texto. Por el contrario, las funciones cos y printf no tienen dirección y aparecen marcadas como U (Undefined symbol) de indefinido, porque están en las bibliotecas dinámicas.

Con el mandato `ldd cos_dyn.exe` obtenemos las bibliotecas dinámicas que necesita este ejecutable. Obtenemos:

```
libm.so.6 => /lib/i386-linux-gnu/i686/cmov/libm.so.6 (0xb776d000)
libc.so.6 => /lib/i386-linux-gnu/i686/cmov/libc.so.6 (0xb75c2000)
```

Compilamos el programa de forma estática, con el mandato `gcc mi_cos.c -static -lm -o cos_sta.exe`, obtenemos el ejecutable, que hemos llamado `cos_sta.exe`.

Obtenemos los símbolos del programa con el mandato `nm cos_sta.exe` . Entre los símbolos obtenidos destacamos los siguientes resultados:

```
0804ae60  T  cos
08048aac  T  coseno
08048ad1  T  main
08055260  T  printf
```

La ejecución del mandato `ldd cos_sta.exe` nos devuelve el siguiente mensaje, que nos indica que no es un ejecutable dinámico, por lo que no tiene bibliotecas:

not a dynamic executable

Con el mandato `size` obtenemos los tamaños de los ejecutables:

- Para el caso del ejecutable dinámico `size cos_dyn.exe` nos devuelve el tamaño de la región de texto (text) de datos con valor inicial (data) y datos sin valor inicial (bss), así como el total expresado en decimal (dec) y en hexadecimal (hex):

```
 text   data    bss    dec    hex filename
 1494    292      4   1790    6fe cos_dyn
```

- Por el contrario para el ejecutable estático `size cos_sta.exe` nos devuelve:

```
  text    data     bss     dec     hex filename
 604907    4004    4988  613899   95e0b cos_st.exe
```

4.6. CREACIÓN DE LA IMAGEN DE MEMORIA DEL PROCESO

La creación de la imagen de memoria de un proceso se realiza a partir del fichero ejecutable y, en caso de montaje dinámico, de las bibliotecas dinámicas. Analizaremos primero la organización del fichero ejecutable, para ver seguidamente la creación de la imagen.

4.6.1. El fichero ejecutable

El fichero ejecutable contiene toda la información necesaria para que el sistema operativo pueda crear un proceso. Existen distintos formatos de ficheros ejecutables (p. ej. Executable and Linkable Format (ELF)). En general, el ejecutable contiene una cabecera y unas secciones, como se indica en la figura 4.27.

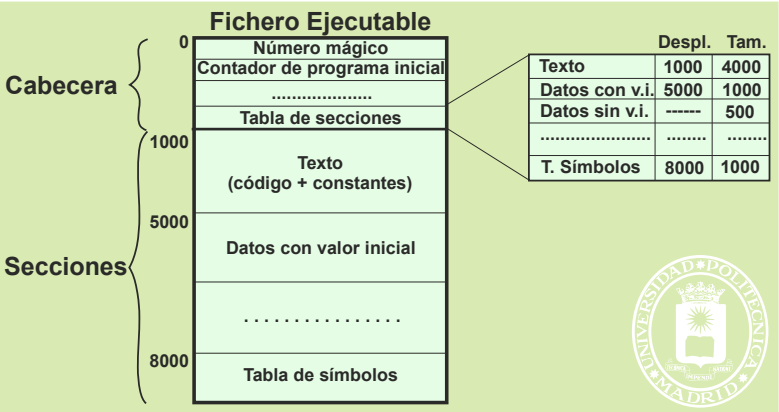


Figura 4.27 *Estructura de un fichero ejecutable.*

La **cabecera** incluye las siguientes informaciones:

- **Número mágico**. Es una clave que sirve para determinar que el fichero es realmente un ejecutable que tiene un determinado formato.
- Valor que los **registros** del computador deben tomar al iniciar la ejecución del proceso. Es especialmente importante el contador de programa inicial, puesto que tiene que apuntar a la dirección de comienzo del programa.
- **Tabla de secciones**. La tabla de secciones tiene una entrada por cada sección, indicando su tipo, dirección de comienzo en el fichero y tamaño.

Las principales secciones que contiene un ejecutable son: la sección de texto, la sección de datos con valor inicial y las tablas de símbolos.

La **sección de texto** incluye el código del programa en lenguaje máquina así como las constantes y las cadenas de texto.

La sección de **datos con valor inicial** incluye los datos estáticos con valor inicial. Esto es, los datos cuya vida se extiende a lo largo de toda la vida del proceso y, además, tienen valor inicial.

Es de destacar que los **datos sin valor inicial no tienen sección**, puesto que no tiene sentido almacenar unos datos que no tienen valor. Sin embargo, su tamaño está especificado en la tabla de secciones, puesto que en la imagen de memoria sí hay que reservar espacio para ellos.

En las tablas de símbolos se encuentran dos tipos de informaciones. Por un lado, puede estar una información de **depuración** que permita al depurador asociar los nombres de las variables con su posición en el mapa de memoria así como las líneas de código fuente con la posición en el programa máquina. Por otro lado, en el caso de montaje dinámico hay **información para enlazar con las bibliotecas dinámicas**.

Biblioteca dinámica

La biblioteca dinámica tiene una estructura similar a la de un fichero ejecutable, constando de una cabecera (con su tabla de secciones) más las secciones de texto, datos con valor inicial y tablas de símbolos.

4.6.2. Creación de la imagen de memoria. Montaje estático

La figura 4.28 muestra la relación entre el fichero ejecutable y la imagen de memoria del proceso.

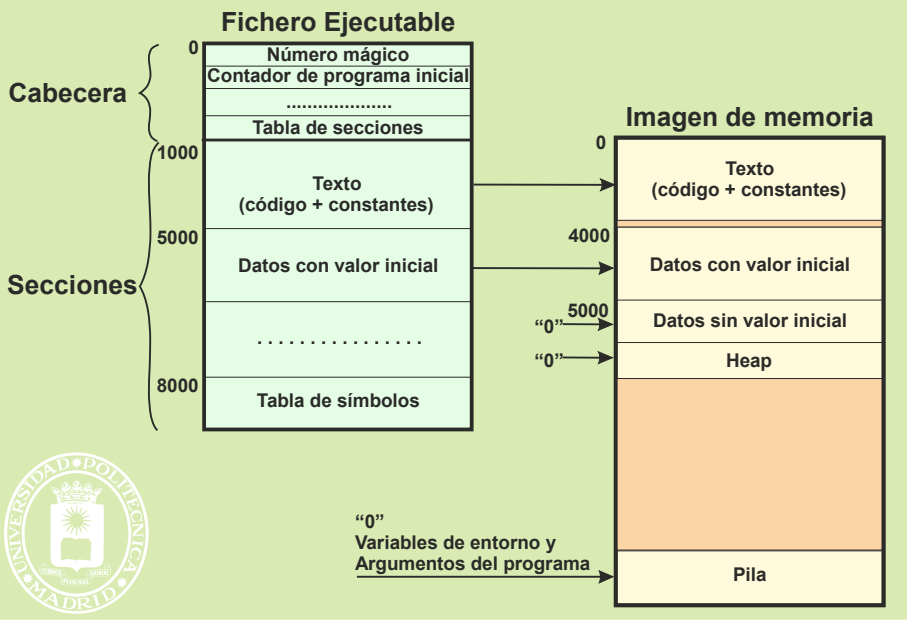


Figura 4.28 *Creación de la imagen de memoria de un proceso a partir del fichero ejecutable.*

Se puede observar que la sección de **texto** del fichero ejecutable corresponde directamente con la región de texto de la imagen de memoria.

En muchos casos la **región de datos** del proceso incluye los siguientes elementos:

- Datos con valor inicial. Estos datos se corresponden directamente con la sección de datos con valor inicial del ejecutable.
- Datos sin valor inicial. El tamaño de estos datos viene dado en la tabla de secciones del ejecutable, pero como no existe valor inicial se deben rellenar a ceros, lo que se indica en la figura por "0" →.
- El *heap* inicial. Como no tiene valor inicial, se marca como a rellenar a ceros. Además, su tamaño inicial viene determinado por el sistema operativo

Finalmente, la región de **pila** sólo contiene la pila inicial, el resto del espacio destinado a pila no tiene información válida, por lo que se marca también a rellenar a ceros. La pila inicial la construye el sistema operativo en base a las variables de entorno del proceso padre y a los argumentos de llamada al programa.

4.6.3. Creación de la imagen de memoria. Montaje dinámico

Como muestra la figura , por cada biblioteca dinámica se necesitan dos regiones adicionales, la de texto y la de datos.

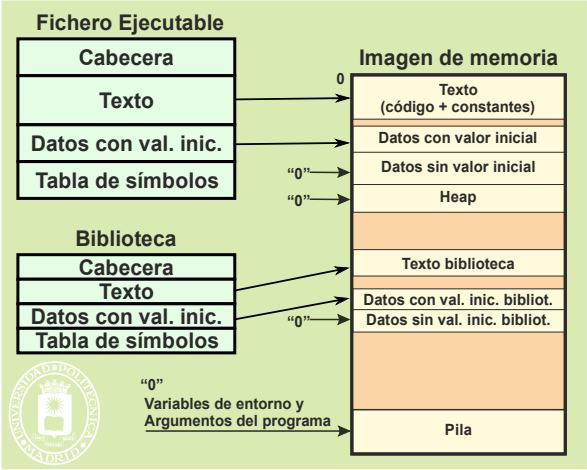


Figura 4.29 *Creación de la imagen de memoria de un proceso a partir del fichero ejecutable y de una biblioteca dinámica.*

La nueva región de texto se corresponde con el texto de la biblioteca. La región de datos de la biblioteca se forma con los datos con valor inicial incluidos en la biblioteca más el espacio requerido para los datos sin valor inicial. Hay que destacar que esta región de datos **no incluye *heap***, puesto que el *heap* es del proceso y no de la biblioteca.

4.6.4. El problema de la reubicación

El problema de la reubicación apare debido a las dos razones siguientes. Por un lado, las instrucciones de máquina de un programa pueden contener direcciones absolutas, por lo que el programa, para que funcione correctamente, debe ser cargado en una dirección determinada del mapa de memoria. Por otro lado, el sistema operativo debe tener suficiente flexibilidad para asignar al proceso las direcciones de memoria que mejor convengan, por ejemplo, que estén libres.

Para resolver esta situación se pueden utilizar las dos soluciones siguientes:

- Se pueden evitar las direcciones absolutas en los programas, dando lugar a lo que se llama **código reubicable**, puesto que puede ejecutar en cualquier zona de memoria. El inconveniente de esta solución es que los programas reubicables son menos eficientes que los no reubicables.
- Se pueden modificar las direcciones absolutas del programa para adecuarlas a la posición en memoria que ocupa realmente el programa. Esta operación se denomina **reubicación**. La figura 4.30 presenta un ejemplo de las modificaciones necesarias para reubicar un programa preparado para ejecutar a partir la posición 0, cuando se carga a partir de la posición 10.000. La reubicación es una operación costosa que habría que hacer a la hora de generar la imagen de memoria del proceso, por ello, se utilizan técnicas *hardware* que realizan la reubicación de forma automática.

Adicionalmente, se debe ajustar el valor inicial del contador de programa para que se ajuste a la posición real del programa.

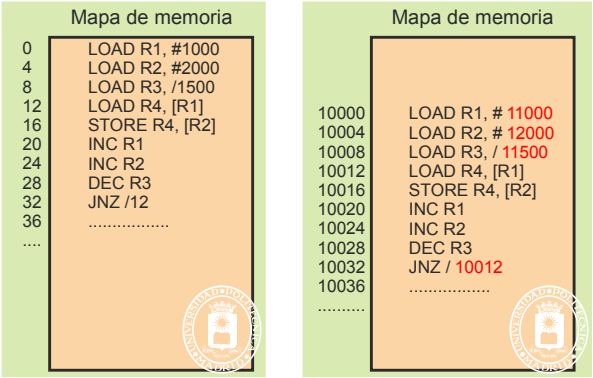


Figura 4.30 *Reubicación de un programa preparado para la posición 0 si se carga en la posición 10.000.*

Memoria real

La solución empleada en sistemas de memoria real utiliza los registros base y límite, como se puede observar en la figura 4.31. El programa se prepara para ejecutar en la dirección 0, pero a todas las direcciones que genera el procesador se la suma el registro base. Por ejemplo, con un valor base de 10.000, la dirección 1.500 se convierte en la dirección 11.500. Con esta solución, desde el punto de vista de los programas, su espacio de direcciones empieza siempre en la dirección 0.

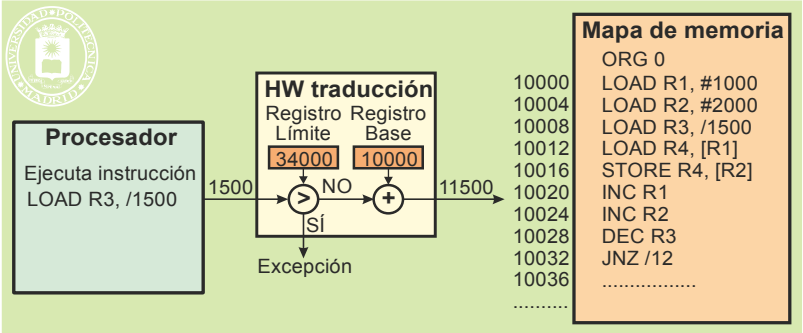


Figura 4.31 *Reubicación hardware en sistemas de memoria real.*

Por otro lado, el registro límite contiene el tamaño de la zona de memoria principal asignado al proceso. El *hardware* compara el valor de la dirección generada por el procesador con el contenido del registro límite. Si se rebasa el límite, se genera una excepción y se cancela el acceso a memoria.

Memoria virtual

En memoria virtual con mapas de memoria independientes por proceso no existe problema de reubicación, puesto que se dispone para cada proceso de todo el rango de direcciones. Se puede, por tanto, preparar los programas para que ejecuten a partir de la dirección 0. Nótese que la dirección 0 del proceso A no tiene nada que ver con la dirección 0 del proceso B.

Bibliotecas dinámicas

Las bibliotecas dinámicas presentan un problema adicional. El sistema operativo las colocará donde exista espacio libre suficiente. Esto significa que hay que resolver un problema de reubicación, puesto que no se conoce a priori su ubicación. Es más, veremos que, en sistemas con memoria virtual, la región de texto de una biblioteca se comparte entre todos los procesos activos que la utilicen. Sería muy poco flexible que dicha región estuviese cargada en la misma dirección en cada uno de los procesos (se producirían fácilmente colisiones a la hora de crear las imágenes de memoria), por lo que el código de las bibliotecas dinámicas se hace **reubicable**. De esta forma, con independencia de la ubicación de la región de texto de la biblioteca, su código ejecutará correctamente.

4.6.5. Fichero proyectado en memoria

El servicio del sistema operativo de proyección de fichero crea una nueva región en la imagen de memoria del proceso que lo solicita, haciendo corresponder las entradas de la tabla de páginas de dicha región con un cierto número de bloques de un fichero, como se muestra en la figura 4.32. Los bloques tienen el tamaño de la página.

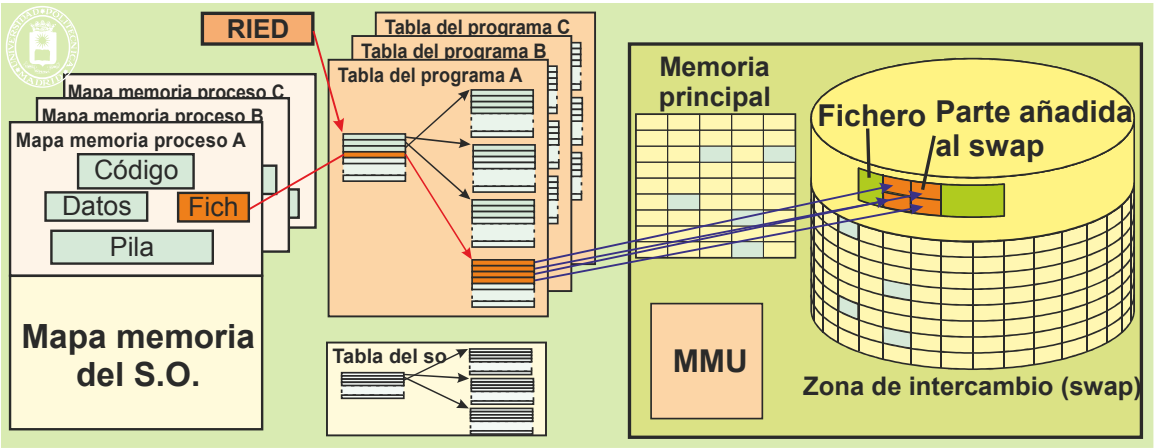


Figura 4.32 *Proyección de un fichero en una región de memoria.*

La parte del fichero proyectada se trata de forma similar al espacio de intercambio o *swap*, pudiéndose considerar como una extensión del mismo.

Una vez que el fichero está proyectado, si el programa accede a una dirección de memoria perteneciente a la región asociada al fichero, estará accediendo a la información del fichero. El programa ya no tiene que usar (ni debe usar) los servicios del sistema operativo para leer (`read`) y escribir (`write`) en el fichero.

El acceso a un fichero mediante su proyección en memoria disminuye considerablemente el número de llamadas al sistema operativo necesarias para acceder a un fichero. Con esta técnica, una vez que el fichero está proyectado, no hay que realizar ninguna llamada adicional. Esta reducción implica una mejora considerable en los tiempos de acceso, puesto que la activación de una llamada al sistema tiene asociada una considerable sobrecarga computacional.

La proyección se puede hacer de forma compartida o privada, y se le pueden dar diferentes permisos.

Proyección compartida

En la proyección compartida, los cambios realizados sobre la región:

- Son visibles por otros procesos que tengan proyectado el fichero también de forma compartida.
- Las modificaciones que se hagan sobre la región de memoria modifican el fichero en el disco.

Las regiones de código se suelen construir proyectando la sección de código de los ficheros ejecutables en modo compartido y con permisos de lectura y ejecución. Ello puede exigir que esta sección se ajuste a un número entero de páginas.

Proyección privada

En la proyección privada, los cambios realizados sobre la región:

- No son visibles por otros procesos que tengan proyectado el fichero.
- No modifican el fichero en el disco.
- Si se escribe en la región es necesario realizar una copia privada de la página modificada.

Las regiones de datos con valor inicial se suelen construir proyectando la sección de datos con valor inicial de los ficheros ejecutables en modo privado y con permisos de lectura y escritura. Ello puede exigir que esta sección se ajuste a un número entero de páginas.

4.6.6. Ciclo de vida de las páginas de un proceso

Cuando se crea una nueva imagen en un sistema con memoria virtual partiendo del fichero ejecutable, es decir, cuando se realiza un servicio `exec`, y se emplea la técnica de ficheros proyectados en memoria, el soporte físico de las páginas es el siguiente:

- Texto y datos con valor inicial: las páginas residen en el fichero ejecutable.
- Datos sin valor inicial: las páginas no tienen soporte físico, son páginas a rellenar a ceros.
- Pila. La pila inicial la crea el sistema operativo. El entorno se copia del padre y el bloque de activación del main se genera con información del fichero ejecutable. El resto de las páginas de la región de pila no tiene soporte físico, son páginas a rellenar a ceros.

A medida que el proceso ejecuta las páginas van cambiando de soporte físico, según se muestra en la figura 4.33.

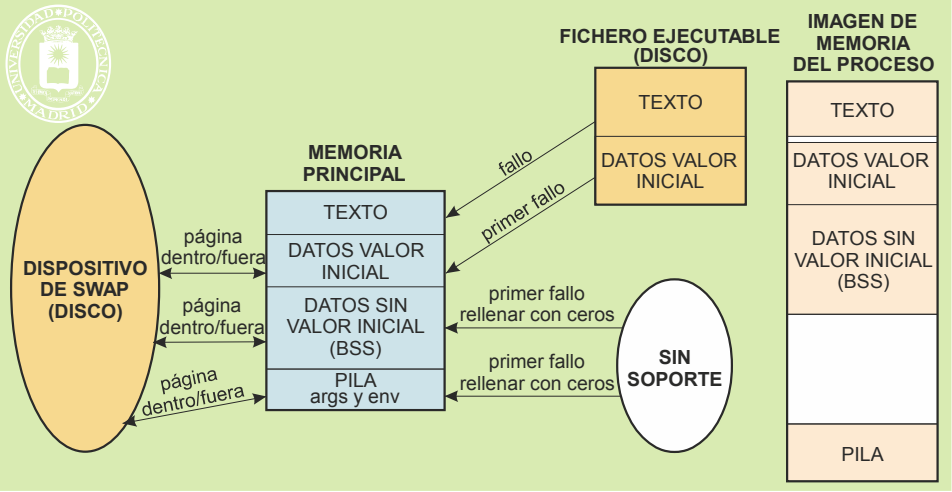


Figura 4.33 *Evolución del soporte físico de las páginas de la imagen de memoria de un proceso.*

Texto: Cuando se produce un fallo de página, la página migra del fichero ejecutable a un marco de memoria. Cuando se expulsa una página del texto de su marco no se hace nada, puesto que la copia del fichero ejecutable sigue siendo válida.

Datos con valor inicial: Cuando se produce el primer fallo de página, la página se copia del fichero ejecutable a un marco. Cuando dicha página es expulsada del marco, si la página ha sido modificada (está sucia), se manda al *swap*. Seguidamente, la página podrá migrar de *swap* a un marco de memoria y viceversa.

Datos sin valor inicial: Cuando se produce el primer fallo de página, se le asigna un marco de página, previamente borrado para evitar que el proceso pueda acceder a información dejada en el marco anteriormente. Seguidamente, la página podrá migrar de *swap* a marco de memoria y viceversa.

Pila: La pila inicial la crea el sistema operativo en uno o varios marcos de página que quedan asignados a la región. El resto de las páginas se comportan de igual forma que las de los datos sin valor inicial.

Fichero proyectado en memoria como privado: Se comporta de igual forma que los datos con valor inicial.

Fichero proyectado en memoria como compartido: Las páginas migran del fichero a marcos de memoria. Si se expulsa una página que no ha sido modificada, no se hace nada, pero si la página está sucia se copia al disco, quedando el fichero modificado. Cuando el proceso termina, todas las paginas sucias del fichero proyectado se copian al disco.

4.6.7. Técnica de *copy on write* (COW)

El *copy on write* es una técnica de optimización para la copia de regiones que tienen permisos de escritura. Es especialmente útil para implementar el servicio `fork`, puesto que es necesario copiar toda la imagen de memoria del padre.

Para copiar, por primera vez, una región con permisos de escritura se procede de la siguiente manera:

- Se marcan en la tabla de páginas de la región todas las páginas como sólo de lectura y como COW.
- Se crea un contador de COW por página con un valor de 2.
- Se copian las tablas de páginas de la región a copiar. Por lo que los dos procesos comparten los mismos marcos de páginas y páginas de intercambio, pero cada uno a través de su propia tabla de páginas. Es de notar que la información de la región no se ha copiado, solamente la tabla de páginas.
- Seguidamente, si un proceso intenta escribir en la región la MMU producirá una excepción de violación de memoria. El sistema operativo, al reconocer que se trata de una página en COW se encarga de desdoblar la página afectada, como se muestra en la figura 4.34, copiándola en un nuevo marco de memoria. De esta forma, cada proceso mantiene la información que le es propia, sin afectar a los demás.
- El proceso que recibe este nuevo marco lo tiene en propiedad, por lo que en su tabla de páginas se marca como de lectura y escritura y deja de estar en COW. Además, hay que decrementar el contador de COW. Si este contador llega a 1, implica que sólo un proceso está usando la página en COW, lo que significa que es privativa, por lo que se marca como de lectura y escritura y se elimina de COW.

Para copiar una región que tiene páginas en COW se procede como sigue: Para las páginas en propiedad se procede como se ha indicado más arriba para una copia por primera vez. Para las páginas que están en COW basta con incrementar el contador asociado. Seguidamente, se copia la tabla de páginas.

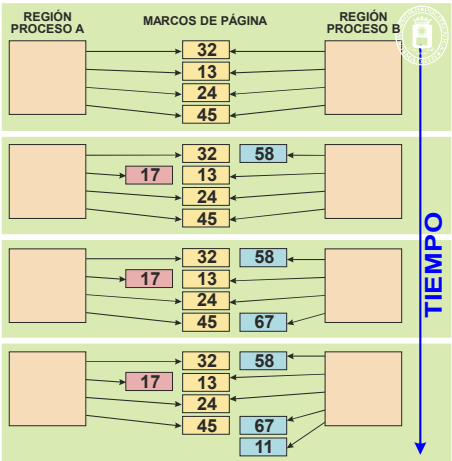


Figura 4.34 *Copy on write.* Al escribir el proceso B en la página 32 se procede a desdoblar dicha página, copiándola en un nuevo marco, que es el 58. Al escribir el proceso A en la página 13 se desdobla, copiándola en la 17.

La ejecución del servicio `fork` con COW supone compartir todas las regiones y marcando las privadas como COW tanto en el padre como en el hijo. Como resultado de esta optimización, en vez de duplicar el espacio de memoria sólo se duplica la tabla de páginas, lo que es especialmente interesante cuando después de ejecutar un servicio `fork`, el hijo ejecuta un servicio `exec`, que implica borrar la imagen que se acaba de copiar.

4.6.8. Copia por asignación

La copia por asignación es una optimización en la que el sistema operativo, en vez de copiar la información al espacio del proceso, le asigna un o varios marcos de página relleno con la información.

Tiene la limitación de que requiere copiar páginas enteras, pero es muy eficiente. Se emplea, por ejemplo, en operaciones de entrada: el disco escribe por acceso directo en un marco de página, que, completada satisfactoriamente la lectura, es asignado al proceso.

4.6.9. Ejemplo de imagen de memoria

En la figura 4.35 se puede observar la imagen de memoria de un proceso que tiene dos *threads*, una biblioteca dinámica, una región de memoria compartida y un fichero proyectado en memoria.

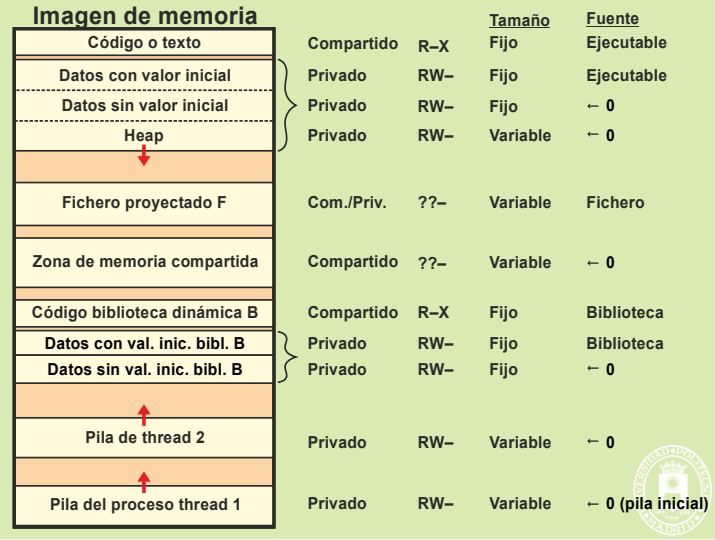


Figura 4.35 *Imagen de memoria de un proceso con dos threads, una biblioteca dinámica, una región compartida y un fichero proyectado en memoria.*

Las regiones de su imagen de memoria y sus correspondientes características son las siguientes:

- Región de **texto del programa**.
  - La fuente es la sección de texto del fichero ejecutable.
  - Las regiones de texto son inmutables, por lo que pueden ser compartidas entre varios procesos que estén ejecutando el mismo programa.
  - Las regiones de texto tienen que tener derechos de lectura y ejecución. Derecho de ejecución para poder ejecutar el programa. Derecho de lectura para poder leer las constantes y cadenas de texto.
  - Al ser inmutable, su tamaño es fijo.
- Región de **datos del programa**. La región de datos se compone, en este caso, de las tres subregiones de datos con valor inicial, datos sin valor inicial y *heap*.
  - La fuente de la subregión de datos con valor inicial es la sección de datos con valor inicial del fichero ejecutable. Sin embargo, la fuente de las subregiones de datos sin valor inicial y *heap* es a rellenar con ceros.
  - En generala los datos son modificables, por lo que las regiones de datos han de ser privadas (no se puede tolerar que lo escriba un proceso en sus variables afecte a otro proceso).
  - Por la misma razón los permisos de las regiones de datos han de ser de lectura y escritura.
  - Tanto los datos con y sin valor inicial son datos estáticos que no cambian de tamaño durante la ejecución del programa, por lo tanto, estas dos subregiones son de tamaño fijo. Sin embargo, el *heap* almacena datos creados dinámicamente, por lo que tiene que poder crecer para adaptarse a las necesidades del proceso.
- Región de **texto de la biblioteca**. La región de texto de la biblioteca tiene las mismas características que la región de texto del programa.
- Región de **datos de la biblioteca**. La región de datos de la biblioteca se diferencia de la del programa por no tener subregión de *heap*.
- **Pila** del ***thread* inicial** de proceso. En la pila se van apilando los bloques de activación de los procedimientos. Por esta razón tiene las siguientes características:
  - La pila no tiene fuente (salvo la pila inicial) por lo es a rellenar con ceros.
  - La pila tiene información de la ejecución en curso, por lo que debe ser privada.
  - En la pila se ha de poder leer y escribir, por lo que requiere esos dos permisos.
  - La pila debe ir creciendo a medida que se van anidando llamadas a procedimientos, por lo que tiene que poder crecer a medida que se necesite.
- **Pila** del **segundo *thread***. Por cada *thread* adicional que lance el programa se debe crear una pila cuyas características son iguales a las descritas anteriormente.
- Zona de **memoria compartida**.
  - La zona de memoria compartida no tiene fuente, por lo es a rellenar con ceros.
  - Por su propio objetivo, debe ser compartida.
  - Puede tener derechos de lectura y escritura, lo que dependerá de cómo se solicita el servicio al sistema operativo.
  - Una zona de memoria compartida puede variar de tamaño.
- **Fichero proyectado** en memoria
  - La fuente es el trozo de fichero que se proyecta.
  - Puede ser privada o compartida, dependiendo de cómo se solicita el servicio al sistema operativo.

- Puede tener derechos de lectura y escritura, lo que dependerá de cómo se solicita el servicio al sistema operativo.
- Una proyección de fichero puede variar de tamaño.

4.7. NECESIDADES DE MEMORIA DE UN PROCESO

Las necesidades de memoria de un proceso son básicamente dos: el código ejecutable y los datos. El código se encuentra en la región de texto. Los datos, además de su estructura y tamaño, tienen una serie de características que nos permiten hacer la siguiente clasificación:

- Por su **mutabilidad** los datos son:
  - Constante: Es un dato que tiene un valor fijo que no se cambia en ningún momento.
  - Variable: Es un dato cuyo valor puede cambiar con la ejecución del programa.
- **Ámbito** o Visibilidad, expresa en qué fragmento del programa el dato tiene sentido y puede, por tanto, ser utilizado.
  - Global. Es un dato que es visible para todo el código, que puede, por tanto, se usado por cualquier fragmento de código del programa.
  - Local. Es un dato que solamente puede ser utilizado en un contexto determinado del programa, por ejemplo, dentro de una función.
- **Vida**. La vida expresa en qué instantes de la ejecución del programa el dato tiene sentido, es decir, conserva su valor y puede ser utilizado.
  - Estático. Es un dato que está en la imagen de memoria durante toda la ejecución del programa.
  - Dinámico. Es un dato que, por el contrario, solamente está en el mapa de memoria durante parte de la vida del proceso. Se crea, apareciendo en la imagen de memoria, y se elimina posteriormente, desapareciendo.

Iremos analizando un programa ejemplo para ir viendo sus necesidades de memoria y cómo se plasman dichas necesidades en la imagen de memoria del proceso que lo ejecute, así como en el fichero ejecutable.

El fichero ejecutable

La estructura del fichero ejecutable se muestra en la figura 4.36 con el programa ejemplo.

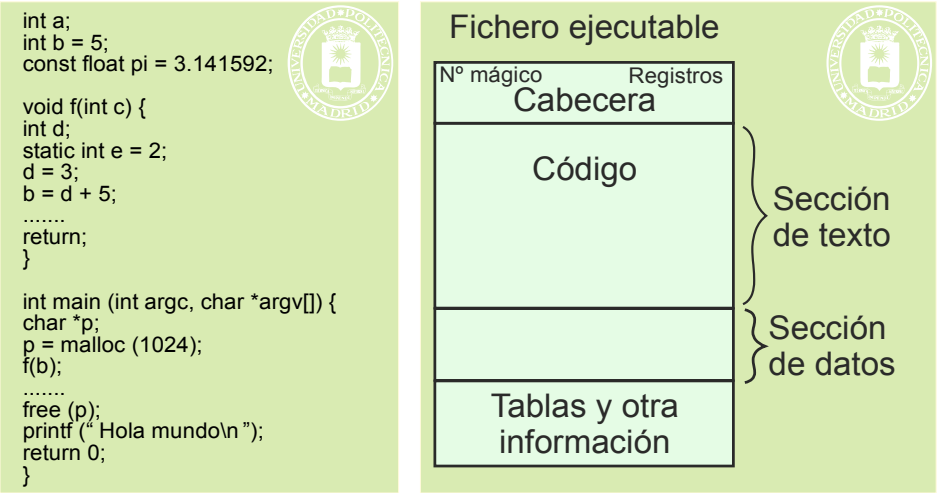


Figura 4.36 *El código del programa se almacena en la sección de texto del fichero ejecutable.*

La figura 4.37 muestra que tanto las constantes como las cadenas de texto (que, por supuesto, también son constantes) se incluyen en la sección de texto, junto con el código.

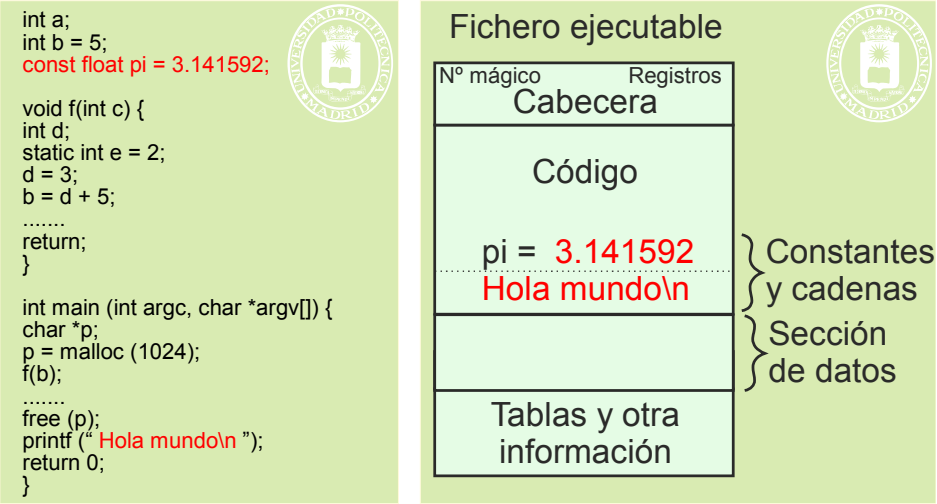


Figura 4.37 *Las constantes y las cadenas de texto se incluyen en sección de texto junto con el código.*

En la figura 4.38 podemos ver que la variable global '**b**' y la variable estática '**e**' con valor inicial se encuentran en la sección de datos con valor inicial del fichero ejecutable. Es de destacar que las variables globales y estáticas que no tienen valor inicial, como '**a**', no ocupan lugar en el ejecutable (esto es lógico, puesto que no tiene sentido almacenar algo que no tiene valor).

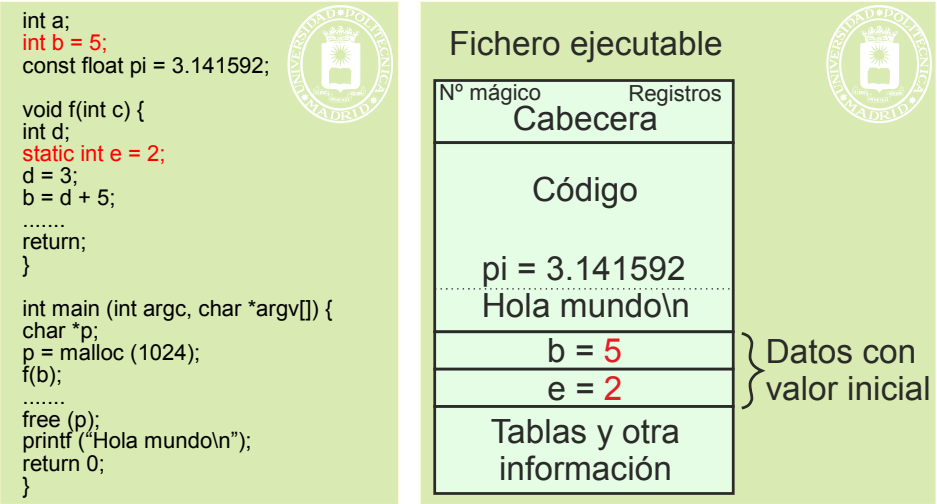


Figura 4.38 *Las variables globales y estáticas con valor inicial se encuentran en la sección de datos del fichero ejecutable.*

También es de destacar que las variables locales se generan dinámicamente por el propio programa, según veremos en los pasos siguientes.

Ejecución del programa

La imagen de memoria del proceso se establece inicialmente tal y como muestra la figura 4.39. La imagen incluye los siguientes elementos:

- La región de texto con el código, el valor de la constante '**pi**' y la cadena del `printf`.
- La región de datos con:
  - Los dos datos '**b**' y '**e**' con sus valores iniciales.
  - El espacio para el dato 'a' sin valor inicial, lo que se indica en la figura mediante un símbolo de interrogación.
  - El espacio del *heap* vacío. El tamaño inicial de *heap* lo determina el sistema operativo.
- La región de pila con:
  - La pila inicial compuesta por las variables de entorno más el bloque de activación de la función `main`. Es de destacar que tanto los argumentos de llamada del `main` (`argvc` y `argv`) como la variable '**p**' definida dentro del `main` se crean dinámicamente en la pila al construirse el bloque de activación de la función `main`. Dado que la variable '**p**' no tiene valor inicial su contenido se indica mediante un signo de interrogación.
  - El resto de la pila vacía. El tamaño inicial de la pila lo determina el sistema operativo.

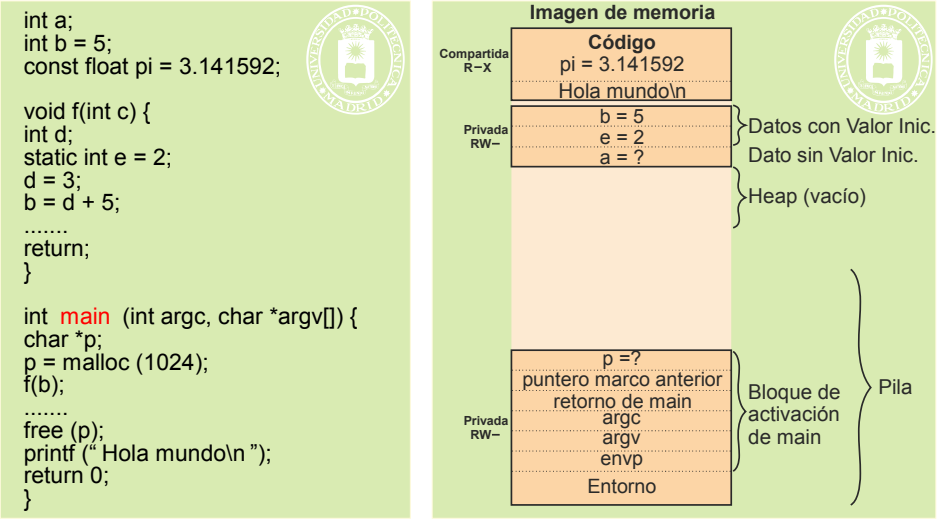


Figura 4.39 *Imagen de memoria del proceso en el instante inicial.*

Lo primero que hace el programa es ejecutar un `malloc` de 1024 bytes. La librería del lenguaje observa si hay un hueco de 1 KiB en el *heap* y, en caso, positivo reserva el espacio de 1KiB y carga en la variable '**p**' la dirección de comienzo de dicho espacio, como se muestra en la figura 4.40. En caso contrario solicitará al sistema operativo el incremento de la región de datos, para luego reservar el KiB.

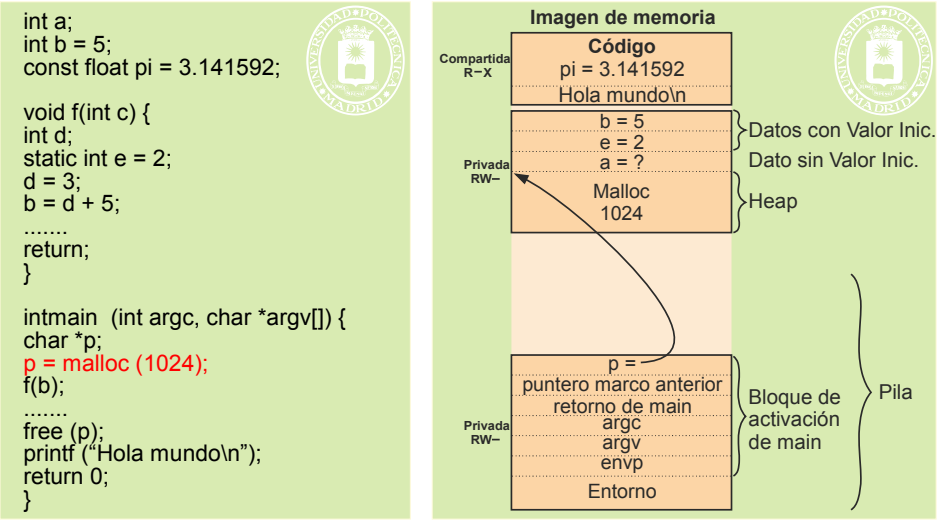


Figura 4.40 *Imagen de memoria del proceso al ejecutarse el malloc.*

Seguidamente, se hace la llamada a la función **f**. El propio código del programa crea el bloque de activación de dicha función, copiando en el argumento '**c**' el valor de la variable '**b**', es decir, 5. En la pila también se encuentra la variable '**d**' definida dentro de la función **f**. Sin embargo, la variable '**e**', al estar declarada como estática, no se encuentra en la pila sino con los datos con valor inicial.

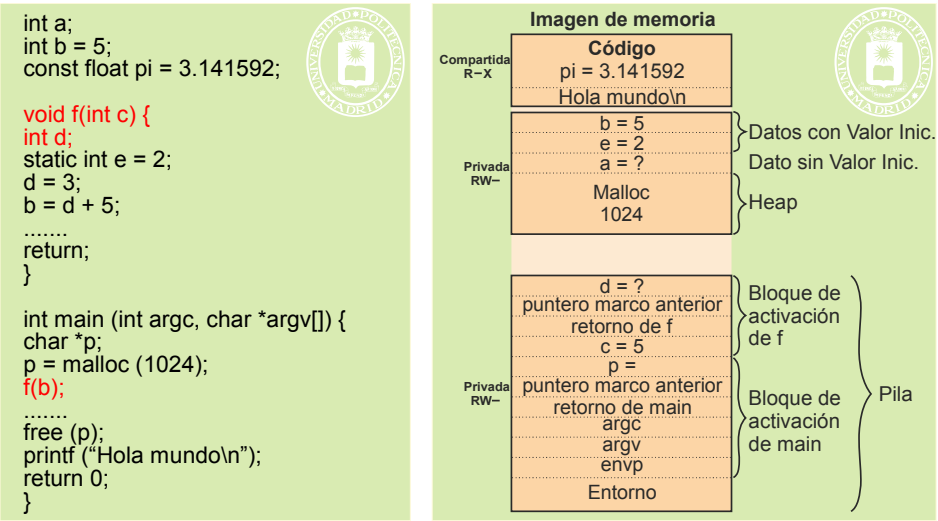


Figura 4.41 *Imagen de memoria del proceso al ejecutarse la llamada a la función **f**.*

Cuando la función **f** completa el `return` expira la vida de su bloque de activación, pero no se borra el contenido de la misma, por el contrario, se mantiene hasta que sea sobrescrito, pero deja de ser válido, convirtiéndose en basura.

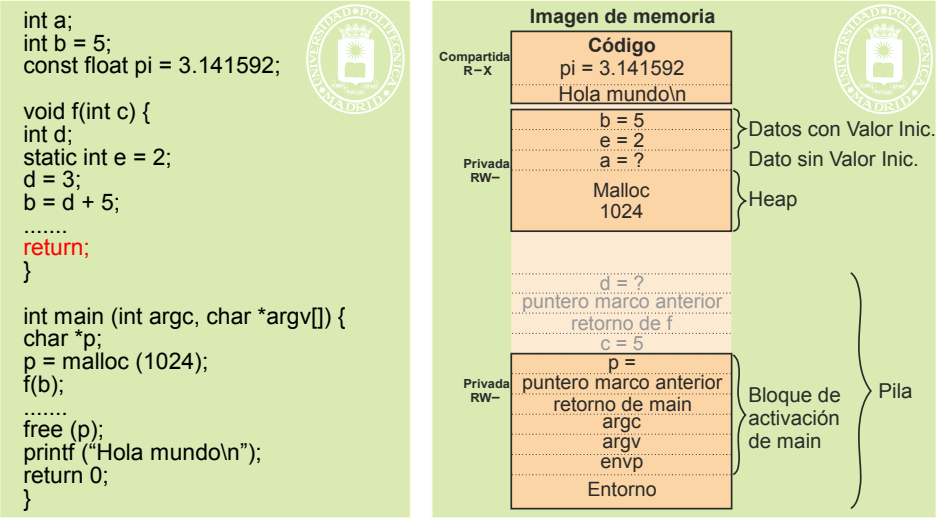


Figura 4.42 *Imagen de memoria del proceso al ejecutarse el return de la función **f**.*

Al ejecutarse el `free` la librería del lenguaje marca la zona de 1 KiB como libre, pero no borra su contenido, que se convierte en basura. Lo mismo ocurre con la variable '**p**' que no es borrada, por lo que conserva su valor, que ahora apunta a basura. Eso puede dar lugar a graves errores de programación si una vez ejecutado el `free` se utiliza '**p**' para acceder a memoria.

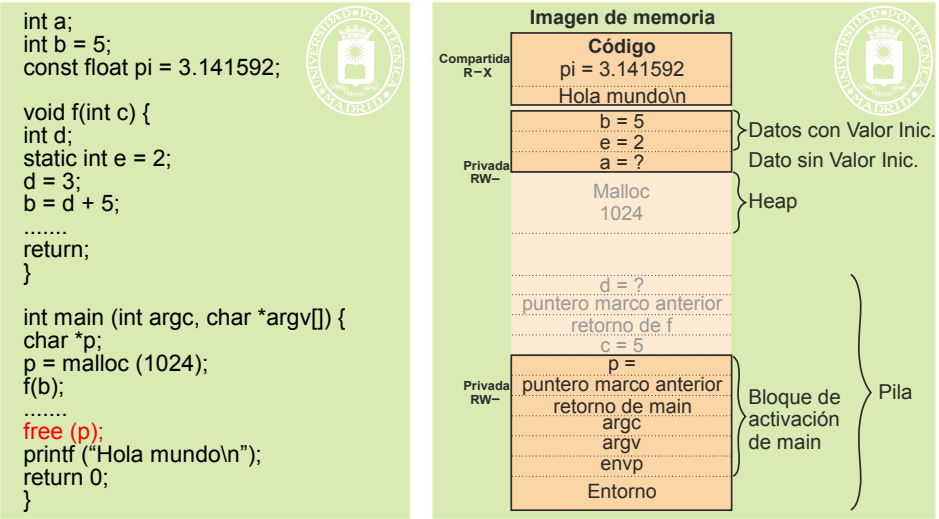


Figura 4.43 *Imagen de memoria del proceso al ejecutarse el free.*

Ejemplo de pila inicial

Para describir cómo se organiza la pila inicial en detalle vamos a considerar la invocación del mandato ls siguiente:

ls –l –a /home/pacorro

Además, vamos a suponer que el prototipo de dicho mandato es el siguiente:

int main(int argc, char* argv[], char* envp[]);

La figura 4.44 muestra en detalle la composición de la pila inicial, incluyendo el entorno y el bloque de activación del main.

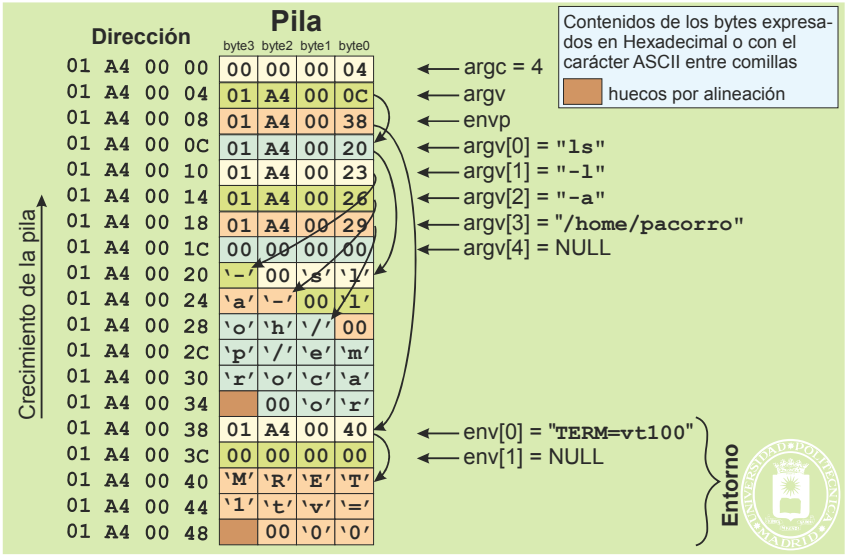


Figura 4.44 *Ejemplo de la pila inicial del proceso creado al ejecutar el mandato: ls –l –a /home/pacorro*

Bloque de activación (RAR) y los parámetros

Como se puede observar en la figuras 4.39 a 4.44 parámetros de un procedimiento constituyen los primeros elementos de su bloque de activación.

El programa llamante, al ir construyendo dicho bloque de activación, mete en la pila los valores asignados a los parámetros (en orden inverso: primero el último). Una vez pasado el control al procedimiento, éste es el único que accede a dichas posiciones de la pila.

En el caso de paso de parámetros **por valor**, lo que se introduce en la pila es el valor del dato. Este método solamente se puede utilizar para parámetros de entrada simples.

En el paso de parámetros **por referencia** se introduce en la pila la dirección del dato, es decir el puntero o manejador del dato, no el dato. El dato estará realmente almacenado en otro lugar (p. ej. en el RAR de procedimiento llamante, en el *heap*, etc.), y, por tanto, sobrevive a la función. Este método sirve tanto para parámetros de entrada como de salida y para datos simples y complejos.

El valor de la función es un parámetro de salida que se pasa por valor en un lugar predeterminado, por ejemplo en un registro del procesador.

4.8. UTILIZACIÓN DE DATOS DINÁMICOS

En esta sección se plantean varios temas relacionados con la utilización de datos, especialmente de datos dinámicos.

4.8.1. Utilización de memoria en bruto

Un programa puede obtener lo que denominaremos memoria en bruto mediante los siguientes procedimientos:

- Mediante las bibliotecas del lenguaje (con funciones tales como malloc, new, …).
- Mediante servicios del sistema operativo, servicios que se detallan más adelante, tales como:
  - Fichero proyectado en memoria.
  - Región de memoria compartida.
  - Nueva región de memoria.

En todos los casos, el programa recibe un puntero, referencia o manejador a esa zona memoria reservada a través del cual podrá utilizarla. La situación se presenta en la parte izquierda de la figura 4.45. Para utilizar dicha memoria el programador, que utiliza un lenguaje de alto nivel, debe proyectar sobre esa zona un conjunto de variables que le permitan referenciar partes de esa zona. Por ejemplo, se puede proyectar un vector de bytes, una matriz de enteros, un vector de estructuras, etc.

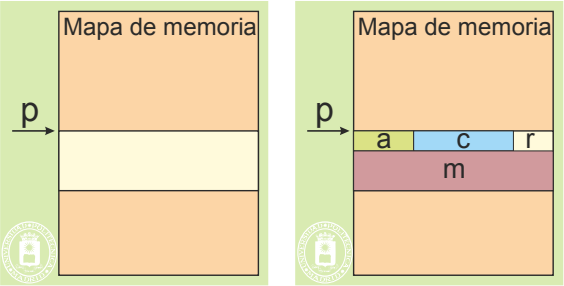


Figura 4.45 *Al solicitar un espacio de memoria al sistema operativo o mediante funciones del lenguaje tipo malloc el programa recibe un puntero al espacio reservado.*

Los datos así proyectados se utilizan a través del puntero devuelto por el servicio o por la función del lenguaje. En el siguiente ejemplo se puede observar que se proyecta un vector de estructura tipo `datos`.

```
struct datos {
    int a;
    char b;
    float c;
    int v[10];
};
int main () {
    struct datos *p;
    char *q, d;
    /* Se reserva memoria para 100 datos. Se proyecta la estructura datos */
    p = malloc (sizeof (struct datos)*100);
    q = p;
    /* se utiliza la memoria mediante p como un vector de estructuras*/
    p->a = 5;
    p->b = 'a';
    p++;
    p->a = 2.5;
    p->v[3] = 25;
    ....
    /* también se puede utilizar como un vector de char a través de q*/
    d = q[0];
    ....
    return 0;
}
```

Se puede observar que también se ha proyectado un vector de bytes, por lo que se puede utilizar la zona bajo estas dos ópticas, con los problemas que ello puede entrañar.

Cuando se utiliza una región de memoria entre varios procesos hay que tener extremado cuidado en proyectar los mismos tipos datos en todos los programas. En caso contrario, se producirán resultados erróneos. Una buena forma de evitar este problema es utilizar, en la elaboración de todos los programas que compartan la región, un único fichero de declaración de variables común para todos ellos (en C se utilizará un fichero .h común). Si, en cualquier momento, se modifican los tipos de datos proyectados, la modificación quedará reflejada en todos los programas.

4.8.2. Errores frecuentes en el manejo de memoria dinámica

Los errores más frecuentes a la hora de manejar memoria dinámica se comentan a continuación.

- **Usarla sin necesidad**. La memoria dinámica es mucho más costosa que una variable local. La figura 4.46 muestra la diferencia entre reservar memoria dinámica para un vector de 2000 bytes y la declaración como una variable local.

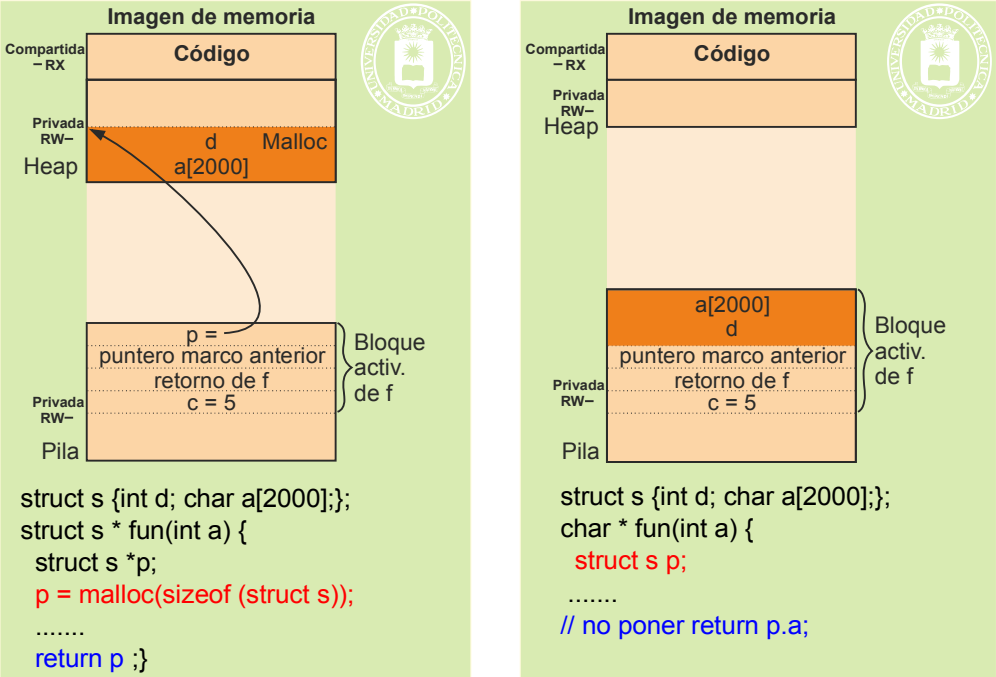


Figura 4.46 *Declaración de un vector de 2000 bytes de forma dinámica mediante un malloc y como variable local.*

En el caso del `malloc`, el espacio reservado sobrevive a la función que lo solicitó, por lo que ésta puede devolver p, para que siga siendo utilizado. Por eso la asignación dinámica es idónea para funciones que crean estructuras dinámicas (listas, árboles) que deben ser utilizadas por otras funciones.

En el caso de declaración local, el coste computacional es mucho menor que con el `malloc`. Sin embargo, la zona de memoria asignada ya no es válida al finalizar la función, puesto que el bloque de activación de la función se recupera en el retorno de f. Esto significa que la función que declara p no debe devolver dirección de p.a.

- **Pérdidas o goteras de memoria**. Las pérdidas o goteras de memoria se producen cuando el programa crea datos dinámicamente y no libera la memoria ocupada cuando esos datos ya no interesan. Por tanto, hay que prestar especial atención a la liberación de datos obsoletos.

  El termino **rejuvenecer el *software*** se refiere a cerrar y volver a lanzar dicho *software*. Es una técnica utilizada, puesto que elimina toda la basura que ha podido ir acumulando el programa en su ejecución.

- **Acceso a un dato obsoleto**. Este problema se produce cuando se accede a los datos mediante una referencia (p. ej. un puntero), después de haber eliminado el dato. Hay que tener en cuenta que al eliminar el dato no se elimina la referencia, por lo que se puede intentar su utilización.

  Si el dato obsoleto pertenece a una región o trozo de región eliminada, el *hardware* detectará el intento de violación de memoria y el sistema operativo enviará una señal al proceso, que suele matarlo.

  En caso contrario se accede a basura, generándose un resultado inesperado, lo que suele producir un error difícil de diagnosticar, puesto que puede aparecer solamente bajo determinadas condiciones de ejecución. Hay que tener en cuenta que la zona que ocupaba el dato puede no haber sido modificada aun.

- **Acceso a un dato no creado**. Se produce cuando se utiliza la referencia (puntero) antes de proyectar el dato.
- **Desbordamiento de un dato múltiple**. Este problema se produce cuando el índice usado en el dato múltiple queda fuera del rango válido. Por ejemplo, cuando accedemos al elemento a[101] de un vector declarado de 100 elementos. El acceso se realizará (nadie lo impide) a la posición de memoria siguiente al espacio reservado para ese vector. Esa dirección corresponderá a otro dato, por lo que el programa producirá un resultado inesperado, difícil de diagnosticar. Solamente en caso de salirse de la región en la que está ese dato, el *hardware* detectará el intento de violación de memoria.
- **Declaraciones compatibles**. Como se ha indicado anteriormente, en el uso de memoria compartida es necesario que las declaraciones de datos sean las mismas en todos los programas que comparten esa memoria.
- **Alineación en los datos**. No tener en cuenta la ocupación real de los datos debidos a la alineación. Esta ocupación puede depender del tipo de máquina para el que se compile el programa, así como de los criterios de compilación que se utilicen, por ejemplo, si se utiliza optimización.

4.8.3. Alineación de datos. Tamaño de estructuras de datos

El objetivo de la alineación de datos es evitar que el acceso a un dato de tipo simple requiera dos accesos a memoria. El problema surge porque algunos datos ocupan menos de una palabra de memoria. Según la figura 4.47, el dato A compuesto por 4 bytes requeriría el acceso a las dos palabras de memoria 400 y 404.

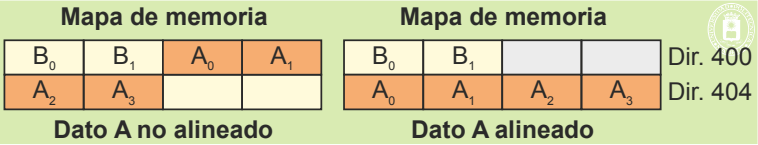


Figura 4.47 *Datos alineados y no alineados.*

Las principales características de la alineación de datos son las siguientes:

- La alineación depende de la arquitectura del computador y del compilador, por lo que el mismo programa puede dar problemas de alineación en una máquina y no darlo en otra.
- La alineación evita que un dato simple quede partido en dos palabras de memoria.
- La alineación se realiza por razones de eficiencia en tiempo de ejecución.
- La alineación supone una pérdida de espacio de memoria, puesto que hay que dejar huecos sin utilizar para conseguir que los datos estén alineados correctamente, como se desprende de la figura 4.48, que refleja el almacenamiento de la estructura siguiente:

```
struct ditur {
    int a;
    char b;
    double c;
    int v[5];
    double m;
    char n;
    char s;
    int r;
}
```

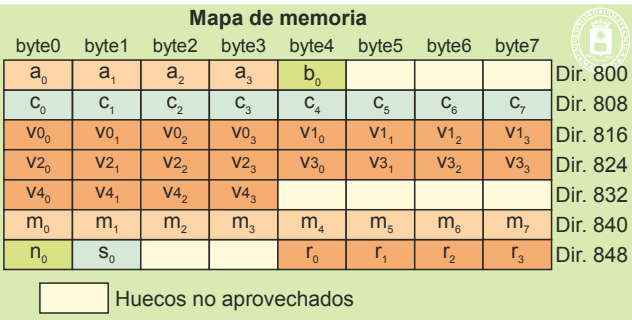


Figura 4.48 *La alineación hace que se pierda espacio, por lo que una estructura puede ocupar más espacio de la suma de los espacios necesarios para sus elementos.*

El compilador puede reordenar los elementos de una estructura para minimizar el espacio perdido. En todo caso, hay funciones de lenguaje como la función `sizeof` de C, que calcula el tamaño realmente ocupado por el elemento que se le pasa como parámetro.

Ejemplo de problema de alineación

Dado el siguiente programa:

```
char a[19];
int *v;
v = (int *) (&a[2])        //Se carga en v la dirección de memoria de a[2]
(*v)++;                    //Acceso no alineado al incrementar *v
v++;
```

En una máquina de 32 bits la 4.49 adjunta muestra el efecto del programa anterior. El entero *v se superpone sobre los caracteres a[2], a[3], a[4] y a[5], quedando no alineado.

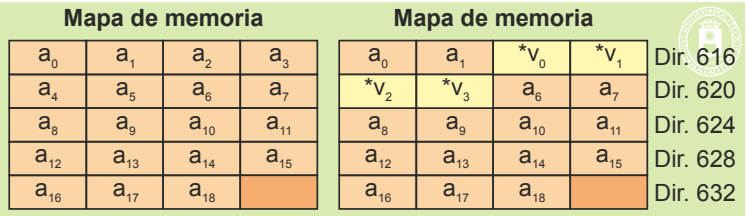


Figura 4.49 *Efecto del programa. El entero *v se superpone sobre los caracteres a[2], a[3], a[4] y a[5]*

Ejecutando el programa en una máquina Intel de tipo x86 el programa ejecuta sin problemas, porque se admiten accesos no alineados.

Sin embargo, ejecutando el programa en un máquina Sparc de 32 bits el programa genera un error de "Bus error" al intentar el incremento de ***v,** al no admitirse accesos no alineados. Por su lado, el incremento de **v** no da problemas en ninguna de las dos arquitecturas.

4.8.4. Crecimiento del *heap*

El *heap* es la zona de memoria donde el programa crea sus datos dinámicos. En lenguaje C la función `malloc` permite que el programa reserve espacio dinámico del *heap*. Para ello, la biblioteca de C incluida en el programa busca en el *heap* un hueco dónde quepa el `malloc` solicitado. Si lo encuentra, asigna el espacio y devuelve la dirección de memoria del mismo. Si no lo encuentra, solicita al sistema operativo el incremento de la región (por ejemplo, mediante el servicio `brk`) y, seguidamente, asigna el espacio. Aunque una vez liberado el espacio por el programa con la función `free` podría la biblioteca solicitar al sistema operativo la reducción de la región, no tiene mucho sentido, puesto que el programa puede necesitar más adelante dicho espacio. La figura 4.50 muestra la evolución del *heap* para cuando cabe la solicitud y para cuando no cabe.

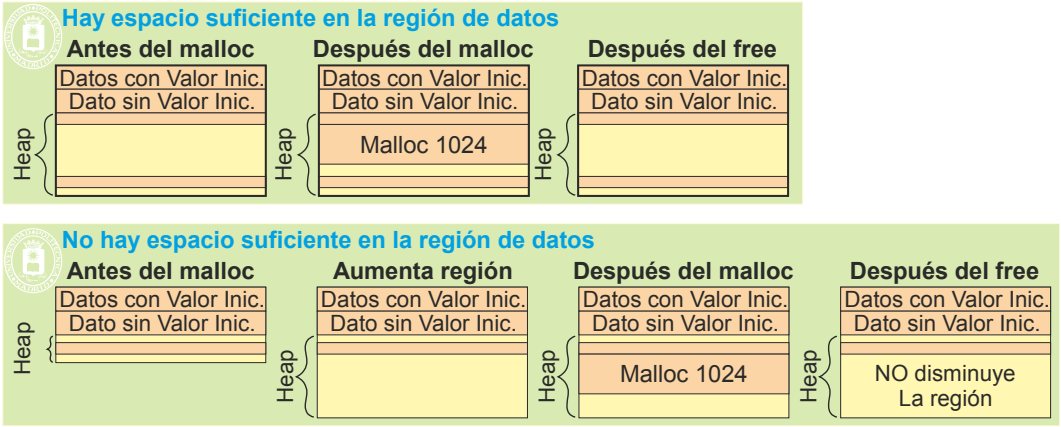


Figura 4.50 *Evolución del heap con un malloc y el correspondiente free.*

4.9. TÉCNICAS DE ASIGNACIÓN DINÁMICA DE MEMORIA

La asignación de memoria es un problema que se plantea con gran frecuencia y a distintos niveles. Por un lado, el sistema operativo se encarga de asignar memoria a los procesos, y de recuperarla. Por otro lado, las bibliotecas de los lenguajes se encargan de asignar y recuperar dinámicamente memoria dentro del *heap*.

En general se dispone de un espacio de memoria contiguo y de tamaño limitado sobre el que se realizan operaciones de asignación de memoria y de liberación de memoria. Seguidamente, se analizarán los tres métodos de asignación de memoria siguientes:

Particiones fijas
Particiones variables
Sistema *buddy* binario

4.9.1. Particiones fijas

En un sistema de particiones fijas se divide a priori el espacio disponible en una serie de trozos o particiones, que pueden ser todas del mismo tamaño o de tamaños diversos. Cuando surge una petición se ha de buscar una partición libre de tamaño adecuado a la petición. Para una memoria con las siguientes particiones: 60, 60, 90, 90 160 y 180 KiB, la figura 4.51 muestra la evolución de las siguientes operaciones:

Operación	Partición	Fragmentación interna
Situación inicial	Todas las particiones libres	
A petición de 140 KiB	160 KiB	20 KiB
B petición de 30 KiB	60 KiB	30 KiB
C petición de 70 KiB	90 KiB	20 KiB
D petición de 30 KiB	60 KiB	30 KiB
E petición de 30 KiB	90 KiB	60 KiB
F petición de 60 KiB	180 KiB	120 KiB

C libera 70 KiB

G petición de 80 KiB 90 KiB 10 KiB

Se puede observar que se asigna más memoria de la solicitada, por lo que se pierde memoria por **fragmentación interna**. La fragmentación interna puede ser especialmente grave si las particiones no se adaptan a las peticiones de memoria.

Es un sistema simple, que conlleva muy poca carga computacional y que puede ser interesante cuando las particiones se ajustan a las necesidades de las peticiones.

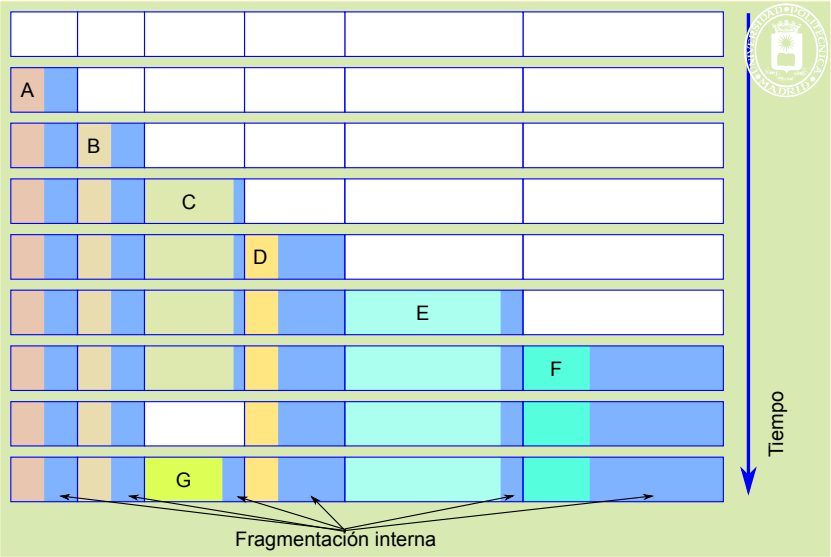


Figura 4.51 *En un sistema con particiones fijas se asigna una partición completa, por lo que se pierde memoria por fragmentación interna, al ser normalmente la partición mayor que la solicitud.*

4.9.2. Particiones variables

El sistema de particiones variables es el más utilizado. Se asigna la memoria solicitada en un hueco libre, generando, normalmente, otro hueco más pequeño. Cuando se libera memoria el hueco se asocia con los posibles huecos contiguos formando un solo hueco.

La figura 4.52 muestra un ejemplo en el que se parte de un espacio contiguo de 360 KiB y se producen las siguientes operaciones:

Operación	Huecos libres	Observaciones
Situación inicial	360 KiB	
A petición de 40 KiB	320 KiB	
B petición de 80 KiB	240 KiB	
C petición de 115 KiB	125 KiB	
D petición de 50 KiB	75 KiB	
E petición de 65 KiB	10 KiB	Hueco pequeño, difícil de utilizar
B libera 80 KiB	80 KiB, 10 KiB	
F petición de 60 KiB	20 KiB, 10 KiB	Dos huecos pequeños, difíciles de utilizar
A libera 40 KiB	40 KiB, 20 KiB, 10 KiB	Dos huecos pequeños, difíciles de utilizar
G petición de 35 KiB	5 KiB, 20 KiB, 10 KiB	Tres huecos pequeños, difíciles de utilizar
C libera 115 KiB	5 KiB, 135 KiB, 10 KiB	Dos huecos pequeños, difíciles de utilizar
H petición de 75 KiB	5 KiB, 60 KiB, 10 KiB	Dos huecos pequeños, difíciles de utilizar
I petición de 50 KiB	5 KiB, 10 KiB, 10 KiB	Tres huecos pequeños, difíciles de utilizar

Los huecos pequeños que quedan no se pueden aprovechar, porque no aparecen peticiones tan pequeñas. Esto supone una pérdida de memoria por **fragmentación externa**.

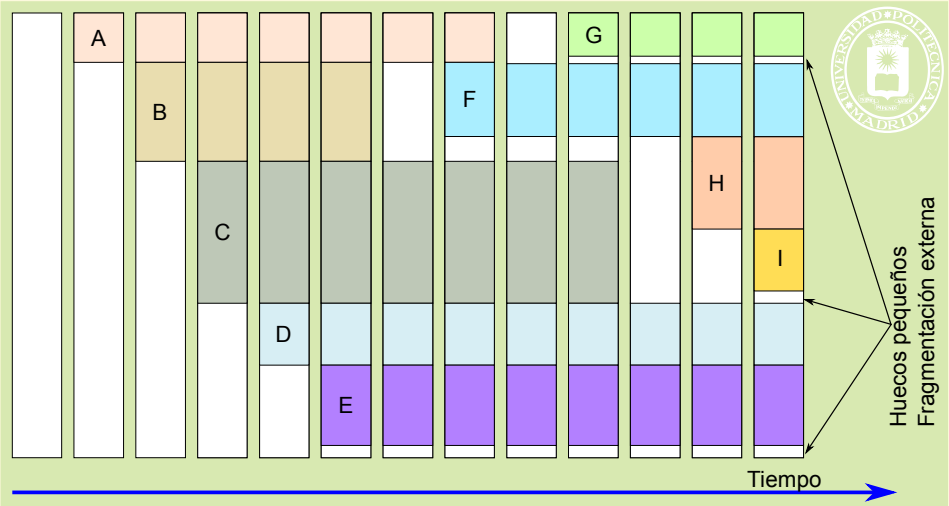


Figura 4.52 *Evolución del espacio de memoria al atenderse varias operaciones de asignación y liberación de memoria. Al quedar huecos pequeños inutilizables se produce fragmentación externa.*

Algoritmo de asignación de espacio

Cuando se produce una solicitud de reserva de espacio, hay que seleccionar qué hueco utilizar entre los existentes en ese instante. Existen cuatro estrategias básicas:

- **El mejor ajuste** (*best fit*). Se elige el hueco más pequeño que satisfaga la petición. A priori, puede parecer la mejor solución. Sin embargo, tiene algunos inconvenientes. Por un lado, se pueden generar huecos muy pequeños inutilizables, lo que produce fragmentación externa. Por otro lado, la selección del mejor hueco exige comprobar cada uno de ellos, lo que alarga considerablemente la búsqueda, o requiere mantenerlos ordenados por tamaño, lo que complica las operaciones de mantenimiento de la información de estado.
- **El peor ajuste** (*worst fit*). Se elige el hueco más grande. Con ello se pretende que no se generen huecos pequeños. Sin embargo, sigue siendo necesario recorrer toda la lista de huecos o mantenerla ordenada por tamaño. Las pruebas empíricas muestran que es una estrategia de poca utilidad. Entre otros problemas, tiende a hacer que desaparezcan los huecos grandes, que se necesitarán para servir peticiones de gran tamaño.
- **El primero que ajuste** (*first fit*). Aunque pueda parecer sorprendente a priori, ésta suele ser la mejor política en muchas situaciones. Es eficiente, ya que basta con encontrar una zona libre de tamaño suficiente, y proporciona un aprovechamiento de la memoria aceptable.
- **El próximo que ajuste** (*next fit*). Se trata de una pequeña variación del *first fit*, tal que cada vez que se realiza una nueva búsqueda se consideran primero los huecos que no se analizaron en la búsqueda previa, y se usa el primero de ellos que encaje. De esta forma, se va rotando entre los huecos disponibles a la hora de satisfacer las peticiones sucesivas. Esta estrategia distribuye de manera más uniforme las reservas a lo largo de la memoria, lo que puede ser beneficioso en ciertos problemas de gestión de memoria.

Gestión de la información de estado

Para poder gestionar el espacio de almacenamiento, es necesario mantener información que identifique los bloques y huecos existentes. Dicha información se organiza de forma que facilite tanto las operaciones de reserva como las de liberación (que facilite colapsar huecos contiguos en un hueco más grande).

La solución más usada consiste en enlazar los huecos disponibles en una **única lista**. Habitualmente, se trata de una lista doblemente encadenada y ordenada. Existen múltiples criterios a la hora de ordenar la lista, entre los que se encuentran los siguientes:

- Por la dirección de comienzo de cada hueco, el más habitual ya que facilita la gestión.
- Por el tamaño de los huecos, que sería apropiado para una política del mejor ajuste.
- En orden LIFO (último hueco generado es el primero usado), que puede generar un mejor uso de la cache.

Los elementos de la lista se pueden almacenar en los propios huecos o en una zona de memoria adicional.

Compactación

Para solucionar el problema de pérdida de memoria por fragmentación externa se puede utilizar una **compactación**, operación que consiste en correr las zonas asignadas, pegando unas con otras. De esta forma quedará un único hueco con todo el espacio libre, que podrá ser utilizado seguidamente.

La compactación es una operación costosa y difícil de realizar, puesto que modifica todas las referencias a los elementos informativos almacenados en las zonas asignadas.

Recolector de basura

Un **recolector de basura** o *garbage collector* es un mecanismo automático de gestión de memoria que intenta liberar la memoria ocupada por los objetos que el programa ya no utiliza.

Está incluido en algunos lenguajes de programación interpretados o semiinterpretados, utilizando una parte importante del tiempo de procesamiento.

4.9.3. Sistema *buddy* binario

En el sistema *buddy* binario se van creando particiones dividiendo por dos una partición disponible. De esta forma, si de parte de un espacio de M bytes se pueden formar particiones de M, M/2, M/4, M/8, M/16, etc. bytes. La división en particiones sucesivas más pequeñas se hace hasta tener una partición que se ajuste a la solicitud realizada. Al liberar memoria se agrupan particiones contiguas del mismo tamaño para formar una única de doble tamaño.

En la figura 4.53 se muestra un ejemplo de asignación mediante sistema *buddy* binario. Se parte de un espacio de 640 KiB y se realizan las operaciones siguientes:

Operación	Comentario
Situación inicial	Todo el espacio disponible está libre.
A petición de 30 KiB	Se dividen los 640 KiB en dos particiones de 320 KiB. Se divide una de 320 KiB en dos de 160 KiB. se divide una de 160 KiB en dos de 80 KiB. Finalmente, se divide una de 80 KiB en dos de 40 KiB y se asigna una. Como se requieren 30 KiB, se produce una pérdida por fragmentación interna de 10 KiB.
B petición de 60 KiB	Se asigna una partición de 80 KiB, sobrando 20 KiB.
C petición de 65 KiB	Se divide una partición de 160 KiB en dos de 80 KiB, y se asigna una de ellas, sobrando 15 KiB.
D petición de 130 KiB	Se divide una partición de 320 KiB en dos de 160 KiB, y se asigna una de ellas, sobrando 30 KiB.
C libera 65 KiB	Se combinan las dos particiones libres contiguas de 80 KiB para formar una partición libre de 160 KiB.
E petición de 100 KiB	Se asigna una partición de 160 KiB y sobran 60 KiB.
F petición de 10 KiB	Se divide una partición de 160 KiB en dos de 80 KiB. Se divide una de 80 KiB en dos de 40 KiB. Se divide una de 40 KiB en dos de 20 KiB. Finalmente se divide una de 20 KiB en dos de 10 KiB y se asigna una de ellas, no sobrando espacio.

De igual forma que para las particiones fijas hay pérdida de memoria por fragmentación interna.

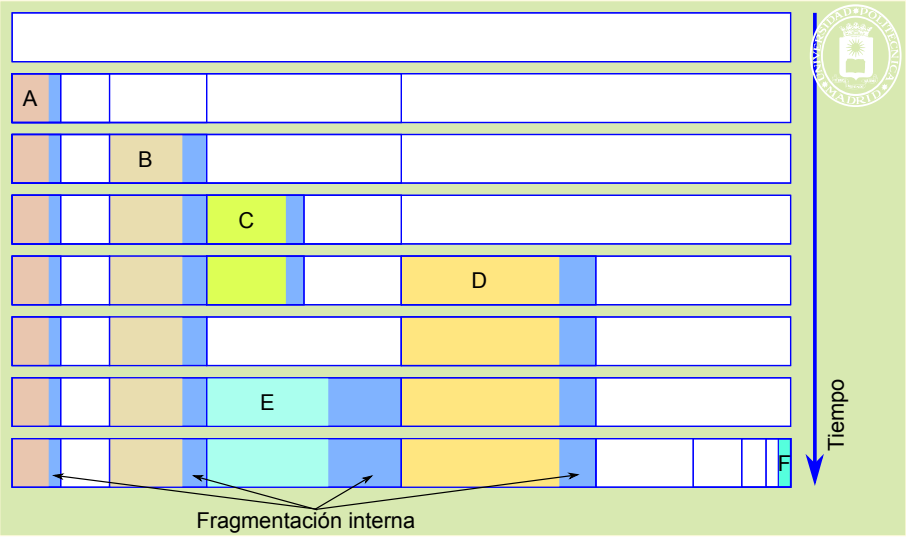


Figura 4.53 *Sistema de asignación de memoria buddy binario.*

4.10. ASPECTOS DE DISEÑO DE LA MEMORIA VIRTUAL

4.10.1. Tabla de páginas

La tabla de páginas pude ser **única**, común a todos los procesos, o **individual** por proceso. El disponer de una tabla de páginas propia supone disponer de un espacio virtual propio e independiente de los espacios virtuales de los otros procesos.

Por otro lado, la tabla de páginas se puede implementar como una tabla **directa** o **inversa**. La tabla directa es una tabla ordenada por direcciones virtuales, por lo que permite hacer la traducción directamente. Por su lado, la tabla inversa está ordenada por direcciones de memoria principal, por lo que exige una búsqueda en la misma para hacer la traducción de direcciones.

Otro aspecto importante es que la MMU puede diseñarse de manera que acceda a la tabla de páginas o que se limite a utilizar la TLB para hacer la traducción.

Si la MMU accede a la tabla de páginas tenemos las siguientes características:

- El diseño de la tabla de páginas viene determinado por la MMU.
- La MMU se encarga de tratar los fallos de acceso a la TLB, actualizándola, para lo que debe acceder a la tabla de páginas.
- La MMU detecta los intentos de violación de memoria (direcciones no permitidas y tipo de acceso no permitido).
- La MMU se encarga de mantener los bits de accedido y modificado en la TLB y en la tabla de páginas.

Si es el sistema operativo el único usuario de la tabla de páginas se tienen las siguientes características:

- El sistema operativo determina libremente el diseño de la tabla de páginas.
- La MMU detecta los fallos de acceso a la TLB, pero es el sistema operativo el que se encarga de actualizarla.
- Es el sistema operativo el encargado de detectar los intentos de acceso a direcciones no permitidas, mientras que la MMU se encarga de detectar los tipos de accesos no permitidos.
- La MMU se encarga de mantener los bits de accedido y modificado en la TLB, pero es el sistema operativo el responsable de mantenerlos en la tabla de páginas.

Tabla de páginas directa

El uso de una tabla directa de un solo nivel obliga a asignar un único bloque de memoria contiguo o presentará un gran número de entradas no válidas correspondientes a los huecos entre bloques de memoria no contiguos, con la consiguiente pérdida de memoria. Por lo tanto, se emplean tablas multinivel, para lo cual se divide la dirección de n bits en p+1 trozos de a0, a1, a2, … ap bits, de forma que a0 + a1 + a2 + … + ap = n. El valor de p indica el número de niveles de la tabla, como se muestra seguidamente:

Niveles de la tabla	Valor p	División dirección	Tamaño página	Tamaño subtablas
1	1	a0 y a1	2^{a0}	2^{a1}
2	2	a0, a1 y a2	2^{a0}	2^{a1} y 2^{a2}
3	3	a0, a1, a2 y a3	2^{a0}	2^{a1}, 2^{a2} y 2^{a3}
4	4	a0, a1, a2, a3 y a4	2^{a0}	2^{a1}, 2^{a4}, 2^{a2} y 2^{a4}
Etc.				

A título de ejemplo, la figura 4.54 muestra el modo de paginación IA-32e del Intel Core i7. Este procesador, por razones de compatibilidad hacia modelos más antiguos, presenta otros modos de paginación que no analizaremos aquí.

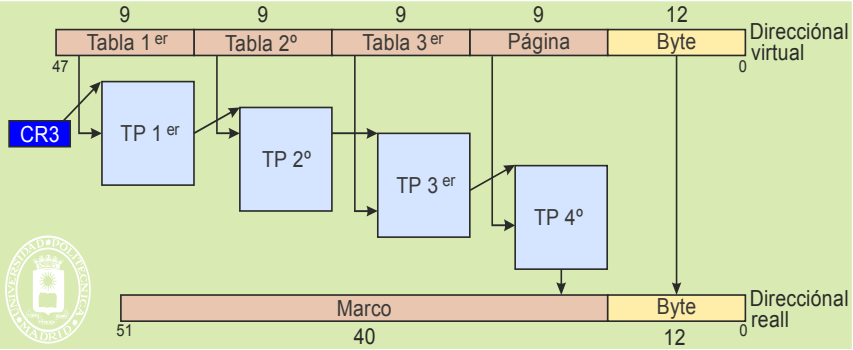


Figura 4.54 *Modelo de tabla de páginas del Intel Core i7 funcionando con paginación IA-32e.*

Las características más importantes son las siguientes:

- Permite un mapa de memoria privado para cada proceso. Este mapa utiliza direcciones de 48 bits, lo que implica un mapa de memoria de 256 TiB. El contenido del registro CR3 especifica el mapa activo en cada momento.
- Esta dirección se divide en 5 partes. Una de 12 bits que permite seleccionar el byte dentro de la página, lo que implica páginas de 4 KiB, y 4 de 9 bits que permiten seleccionar las subtablas de 4º, 3er, 2º y 1er nivel.
- Cada una de estas subtablas tiene exactamente 512 entradas de 8 B, por lo que ocupa 4KiB y se almacena en un marco de página, permaneciendo siempre en memoria principal. Algunas de las entradas de una subtabla pueden no estar en uso, por lo que se emplea un bit para indicar esta situación.
- Entre otros campos, cada entrada de dichas subtablas incluye un número de marco de página, es decir, una dirección de memoria principal frontera de página. En la TP 4º dicha valor indica el marco de página donde está la página referenciada. En el resto de las tablas, dicho valor indica el marco en el que se encuentra

la subtabla de nivel siguiente. En la misma línea el registro CR3 incluye el número de marco de página de la TP 1er.

- Este sistema permite direccionar 2^{40} marcos de páginas de 4 KiB cada una, es decir, un total de 8 PiB de memoria principal.
- Para cubrir el mapa de 256 TiB completamente, el tamaño que ocuparía la tabla de páginas de un proceso es el siguiente:
  - 1 página para la TP 1er.
  - 512 páginas para las 512 TP 2º.
  - 512^2 páginas para las TP 3er.
  - 512^3 páginas para las TP 4º, lo que supone 128 Mi_páginas = 512 GiB

De este ejemplo se pueden sacar una serie de conceptos aplicables a la mayoría de las tablas de páginas directas, como son los siguientes:

- La tabla de páginas reside en memoria principal, puesto que la MMU la debe utilizar sin tener que paginar.
- Es muy conveniente que las subtablas se almacenen cada una en un marco de página. De esta forma basta con el número de marco para acceder a cada subtabla. Al ocupar un marco de página, cada subtabla tendrá un número fijo de entradas, lo que conlleva a que algunas puedan no estar en uso, por lo que habrán de ser marcadas como inválidas.
- Con direcciones de muchos bits es necesario usar tablas de páginas con un alto número de niveles, de forma que cada subtabla tenga un tamaño que se ajuste al de la página.
- Todas las direcciones que se incluyen en las subtablas son direcciones de memoria principal, más concretamente, son números de marco de página.

Hiperpáginas

Cuando se asigna mucha memoria contigua (p. ej. en el SO) las subtabla de páginas son una pérdida de espacio. Para evitar esta situación se utiliza la técnica de la hiperpágina. Para ello, en la penúltima tabla se admiten dos alternativas, que apunte a una tabla de páginas o que apunte directamente a una hiperpágina.

La figura 4.55 muestra las dos alternativas de hiperpáginas que presenta el modo de paginación IA-32e del Intel Core i7. La TP 3er puede direccionar una hiperpágina de 2 MiB y la TP 2º puede direccionar una hiperpágina de 1 GiB.

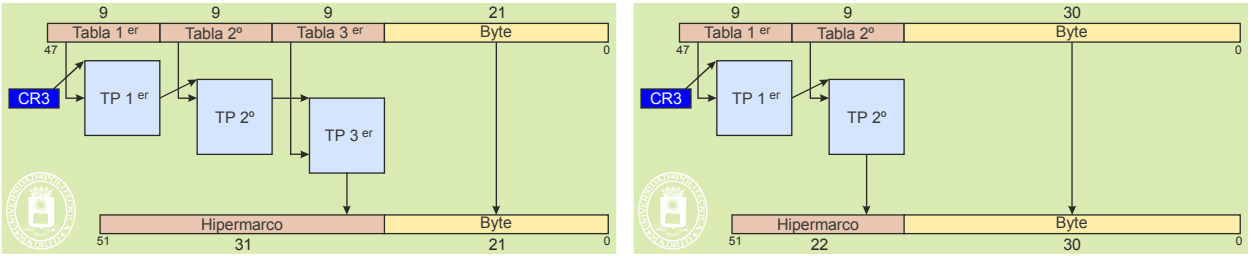


Figura 4.55 *El modelo de tabla de páginas del Intel Core i7 en funcionamiento de paginación IA-32e permite hiperpáginas de 2MiB y de 1 GiB.*

La técnica de la hiperpágina exige tener TLB independientes, una para páginas y otra para hiperpáginas.

Tabla de páginas inversa

Las características de la tabla de páginas inversa son las siguientes:

- Tiene una entrada por cada marco de memoria, que indica:
  - la página almacenada en el marco y
  - el proceso al que pertenece dicha página.
- Su tamaño es reducido puesto que es proporcional a la memoria principal.
  - Sin embargo, el sistema operativo debe mantener la información de todas las páginas asignadas, con el consiguiente uso de memoria. Para ello, suele emplear una tabla de segmentos con las páginas de cada segmento.
- Requiere una búsqueda, por lo que hay que implementar mecanismos para la búsqueda secuencial, para lo cual se organiza como una tabla *hash*.

Los esquemas de traducción con tabla inversa son el simple, el de subbloques y el de subbloques parciales, esquemas que se muestran en la figura 4.56.

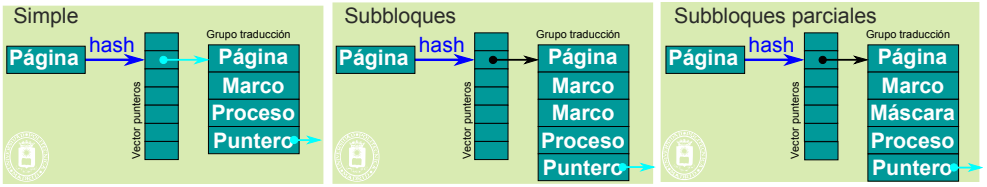


Figura 4.56 *Esquemas de traducción con tabla inversa.*

El funcionamiento general es el mismo para los tres esquemas. Primero se aplica la función *hash,* lo que permite entrar en el vector de punteros a grupos de traducción. Dado que puede haber colisiones en la función *hash*, el grupo de traducción incluye a su vez un puntero a grupo de traducción, para poder establecer una cadena de grupos de traducción por cada clave *hash*. Seleccionado el primer grupo de traducción de la cadena se comprueba si coinciden la página y el proceso. En caso de acierto, se ha terminado. En caso de fallo, se pasa al siguiente elemento de la cadena y se repite la comprobación. Si se llega al final de la cadena sin acierto, es que hay fallo de página.

En el **esquema simple** cada grupo de traducción corresponde a un único marco de memoria. Es un esquema sencillo, pero que requiere mucha memoria para almacenar todos los grupos de traducción.

En el **esquema de subbloques** cada grupo de traducción incluye dos números de marco, que delimitan un bloque contiguo de memoria principal, que queda asociado a un espacio contiguo de memoria virtual. De esta forma se reduce el tamaño total ocupado por la tabla de páginas. Este esquema es equivalente al de las hiperpáginas analizado anteriormente.

Finalmente, en el **esquema de subbloque parciales** el grupo de traducción incluye un número de marco de página mp y una máscara de n bits. La máscara determina los marcos de página utilizados en el rango de marcos [mp,mp+n-1]. Igual que en el caso anterior los espacios virtual y de memoria principal definidos por un grupo de traducción son contiguos.

4.10.2. Políticas de administración de la memoria virtual

Las políticas que definen el comportamiento de un sistema de memoria virtual son las siguientes:

- **Política de localización**. Permite localizar una determinada página dentro de la memoria secundaria.
- **Política de extracción**. Define cuándo se transfiere una página desde la memoria secundaria a la principal.
- **Política de ubicación**. Si hay varios marcos libres, establece cuál de ellos se utiliza para almacenar la página que se trae a memoria principal.
- **Política de reemplazo**. En caso de que no haya marcos libres, determina qué página debe ser desplazada de la memoria principal para dejar sitio a la página entrante.
- **Política de actualización**. Rige cómo se propagan las modificaciones de las páginas en memoria principal a la memoria secundaria.
- **Política de reparto de espacio entre los procesos**. Decide cómo se reparte la memoria física entre los procesos existentes en un determinado instante.

4.10.3. Política de localización

La política de localización permite determinar la posición de la página en memoria secundaria. En el caso de la tabla de páginas directa, la entrada de la tabla incluye un bit de presente/ausente y una dirección. En el caso de presente, dicha dirección identificará el correspondiente marco de página, en caso de ausente dicha dirección puede identificar la página en memoria secundaria o que es a rellenar a ceros.

En el caso de tabla de páginas inversa o en el caso de que la tabla directa no permita la solución anterior, el sistema operativo deberá mantener una estructura de información que permita conocer la ubicación de las páginas en memoria secundaria o que son a rellenar a ceros.

Como veremos más adelante, la técnica de *buffering* de páginas plantea la localización de páginas en unas listas de marcos libres y marcos modificados.

4.10.4. Política de extracción

La memoria virtual, en su forma ortodoxa, opera bajo demanda: sólo se trae a memoria principal una página cuando el proceso accede a la misma. Sin embargo, casi todos los sistemas operativos implementan algún tipo de **agrupamiento de páginas** o de **lectura anticipada de páginas**, de manera que cuando se produce un fallo de página, no sólo se trae a memoria la página involucrada, sino también algunas páginas próximas a la misma, puesto que, basándose en la propiedad de proximidad de referencias que caracteriza a los programas, es posible que el proceso las necesite en un corto plazo de tiempo. Estas técnicas suelen englobarse bajo el término de **prepaginación**. La efectividad de las mismas va a depender de si hay acierto en esta predicción, es decir, si finalmente las páginas traídas van a ser usadas, puesto que, en caso contrario, se ha perdido tiempo en traerlas, expulsando, además, de la memoria principal a otras páginas que podrían haber sido más útiles.

Dado que en la fase de arranque de un programa es donde más se hace patente la sobrecarga de la memoria virtual, al producirse numerosos fallos mientras el conjunto de trabajo del programa va cargándose en memoria, en algunos sistemas operativos se realiza un seguimiento de los primeros segundos de la ejecución del programa, guardando información sobre los fallos de página que provoca. De esta manera, cuando arranca nuevamente un programa, se puede realizar una lectura anticipada dirigida de las páginas que probablemente va a usar el mismo durante su fase inicial.

Para terminar, es conveniente realizar algunas consideraciones sobre el caso de que se produzca un fallo de página cuando el proceso estaba en modo sistema:

- Si la dirección de fallo corresponde a una dirección lógica de usuario y se estaba ejecutando una llamada al sistema, el fallo se produce debido a que se ha accedido al mapa de usuario del proceso para leer o escribir algún parámetro de la llamada. El tratamiento del fallo es el habitual (comprobar si está incluido en alguna región, buscar un marco libre, etc.)
- En caso de que la dirección de fallo corresponda a una dirección lógica de usuario y se estuviera ejecutando una interrupción, se trataría de un error en el código del sistema operativo, puesto que, dado que una interrupción tiene un carácter asíncrono y no está directamente vinculada con el proceso en ejecución, no debe acceder al mapa de usuario del proceso en ninguna circunstancia. El tratamiento sería el habitual ante un error en el sistema operativo (podría ser, por ejemplo, sacar un mensaje por la consola y realizar una parada ordenada del sistema operativo).
- Si la dirección de fallo es una dirección lógica de sistema, a su vez, podrían darse varios casos:
  - Si se trata de un sistema operativo que permite que páginas del sistema se expulsen a disco, se comprobaría si la dirección corresponde a una de esas páginas y, si es así, se leería de disco.
  - Si se usa un procesador que tiene una tabla única para direcciones de usuario y de sistema, y en el que, por tanto, se duplican las entradas correspondientes al sistema operativo en las tablas de páginas de todos los procesos, el fallo puede deberse a que se ha añadido una nueva entrada del sistema en la tabla de un proceso, pero no se ha propagado a los restantes. De esta forma, la propagación de este cambio se hace también por demanda, no tratándose de un error.
  - En todos los demás casos, se correspondería con un error en el código del sistema operativo.

4.10.5. Política de ubicación

La política de ubicación establece en qué marcos de página se puede almacenar una página. En sistemas con memoria principal uniforme no existe ninguna restricción en cuanto a la ubicación de páginas en marcos, por lo que sirve cualquiera. Sin embargo, algunos sistemas operativos intentan seleccionar el marco de manera que se mejore el rendimiento de la memoria cache. Esta técnica, denominada **coloración** de páginas, intenta que las páginas residentes en memoria en un momento dado estén ubicadas en marcos cuya distribución en las líneas de la cache sea lo más uniforme posible.

En multiprocesadores con memoria asimétrica es conveniente ubicar la página cerca del procesador que está ejecutando el proceso.

4.10.6. Política de reemplazo

La política determina qué página se expulsa cuando hay que traer otra de memoria secundaria y no hay espacio libre.

Las estrategias de reemplazo se pueden clasificar en dos categorías: **reemplazo global** y **reemplazo local**. Con una estrategia de reemplazo global se puede seleccionar para satisfacer el fallo de página de un proceso un marco que actualmente tenga asociada una página de otro proceso. Esto es, un proceso puede quitarle un marco de página a otro. Mientras que en la estrategia de reemplazo local sólo pueden usarse marcos de páginas libres o marcos ya asociados al proceso.

En los sistemas operativos actuales los procesos pueden estar compartiendo regiones de texto tanto de los programas como de las bibliotecas, esto hace difícil aplicar una política de reemplazo local.

El objetivo básico de cualquier algoritmo de reemplazo es minimizar la tasa de fallos de página, intentando que la sobrecarga asociada a la ejecución del algoritmo sea tolerable y que no se requiera una MMU con características específicas.

Algoritmo de reemplazo óptimo

El algoritmo óptimo, denominado también MIN, debe generar el mínimo número de fallos de página. Por ello, la página que se debe reemplazar es aquélla que tardará más tiempo en volverse a usar.

Evidentemente, este algoritmo es irrealizable, ya que no se puede predecir cuáles serán las siguientes páginas a las que se va a acceder. Este algoritmo sirve para comparar el rendimiento de los algoritmos que sí son factibles en la práctica.

Algoritmo FIFO (*First Input/First Output*, primera en entrar/primera en salir)

El algoritmo FIFO consiste en seleccionar para la sustitución la página que lleva más tiempo en memoria. Requiere mantener una lista con las páginas que están en memoria, ordenada por el tiempo que llevan residentes. Cada vez que se trae una nueva página a memoria, se pone al final de la lista. Cuando se necesita reemplazar, se usa la página que está al principio de la lista

La implementación es simple y no necesita ningún apoyo *hardware* especial. Sin embargo, el rendimiento del algoritmo en muchas ocasiones no es bueno, puesto que la página que lleva más tiempo en memoria puede contener instrucciones o datos a los que se accede con frecuencia.

Algoritmo LRU (*Least Recently Used*, menos recientemente usada)

El algoritmo LRU está basado en el principio de proximidad temporal de referencias: dado que es probable que se vuelvan a referenciar las páginas a las que se ha accedido recientemente, la página que se debe reemplazar es aquélla a la que no se ha hecho referencia desde hace más tiempo.

Hay un aspecto importante en este algoritmo cuando se considera su versión global. A la hora de seleccionar una página no habría que tener en cuenta el tiempo de acceso real, sino el tiempo lógico de cada proceso.

La implementación exacta en un sistema de memoria virtual es difícil, ya que requiere un considerable apoyo *hardware*. Una implementación del algoritmo podría basarse en lo siguiente:

- El procesador gestiona un contador que se incrementa en cada referencia a memoria. Cada posición de la tabla de páginas ha de tener un campo de tamaño suficiente para que quepa el contador.
- Cuando se hace referencia a una página, la MMU copia el valor actual del contador en la posición de la tabla correspondiente a esa página (realmente, debería ser en la TLB, para evitar un acceso a la tabla por cada referencia).
- Cuando se produce un fallo de página, el sistema operativo examina los contadores de todas las páginas residentes en memoria y selecciona como víctima aquélla que tiene el valor menor.

Algoritmo del reloj

El algoritmo de reemplazo del **reloj** (o de la **segunda oportunidad**) es una modificación sencilla del FIFO. Necesita que cada página tenga un bit de referenciada, pero un gran número de computadores incluyen este bit, gestionado directamente por la MMU, de forma que cada vez que la página es referenciada se pone a 1.

Como muestra la figura 4.57, para implementar este algoritmo se puede usar una lista circular de las páginas residentes en memoria. Existe un puntero que señala en cada instante al principio de la lista. Cuando llega a memoria una página, se coloca en el lugar donde señala el puntero y, a continuación, se avanza el puntero al siguiente elemento de la lista (creando una lista FIFO circular). Cuando se busca una página para reemplazar, se examina el bit de referencia de la página a la que señala el puntero. Si está activo, se desactiva y se avanza el puntero al siguiente elemento (de esta forma esta página queda como la más reciente). El puntero avanzará hasta que se encuentre una página con el bit de referencia desactivado. Debido a ello, esta estrategia se denomina **algoritmo del reloj**. Si todas las páginas tienen activado su bit de referencia, el algoritmo se convierte en FIFO.

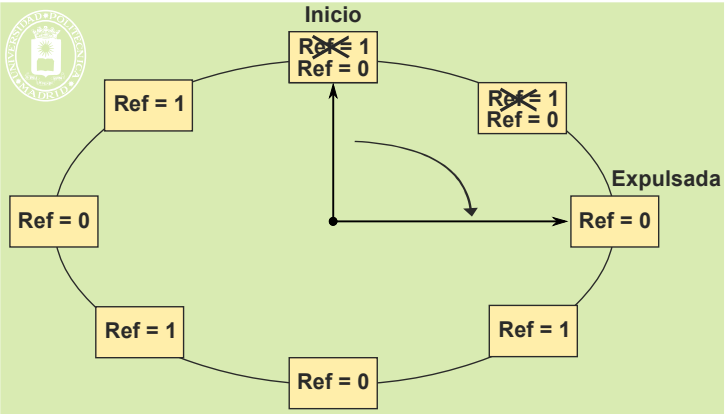


Figura 4.57 *Ejemplo del algoritmo del reloj.*

Este algoritmo es sencillo y evita el problema de eliminar páginas viejas pero muy utilizada, proporcionando unas prestaciones similares a las del algoritmo LRU, sin requerir un *hardware* específico (si el *hardware* no incluye el bit de referenciada, se puede simular por *software*). Esto ha hecho que, con pequeñas variaciones específicas, sea el algoritmo utilizado en la mayoría de los sistemas operativos actuales.

Algoritmo LFU (*Least Frequently Used*, menos frecuentemente usada)

La idea de este algoritmo es expulsar la página residente que menos accesos ha recibido. El problema de esta técnica es que una página que se utilice con mucha frecuencia en un intervalo de ejecución del programa obtendrá un valor del contador muy alto, no pudiendo ser expulsada en el resto de la ejecución del programa, por lo que no produce buenos resultados.

Su implementación, necesitaría una MMU específica que gestionase un contador de referencias, o bien usar una versión aproximada del mismo, que gestionara el contador mediante muestreos periódicos del bit de referencia.

buffering de páginas

Activar el algoritmo de reemplazo bajo demanda, es decir, en el momento que no queda más memoria libre, no es una buena solución, puesto que supone resolver el problema justo cuando el procesador está ocupado, teniendo en cuenta que, en el peor de los casos, se necesitarán dos accesos a memoria para resolver el fallo de página.

Por ello, la mayoría de los sistemas operativos utilizan la estrategia del ***buffering* de páginas,** que consiste en mantener una reserva mínima de marcos libres, realizando las operaciones de reemplazo de forma anticipada. De esta forma, el tratamiento del fallo de página es más rápido puesto que basta con coger un marco de página libre.

Cuando el sistema operativo detecta que el número de marcos de página disminuye por debajo de un cierto umbral, aplica repetidamente el algoritmo de reemplazo hasta que el número de marcos libres llegue a otro umbral que corresponda a una situación de estabilidad. Las páginas liberadas limpias pasan a una lista de marcos libres, mientras que las páginas sucias pasan a una lista de marcos modificados que deberán limpiarse (escribirse en memoria secundaria) antes de poder volver a utilizarse. Esta limpieza de marcos se puede intentar hacer cuando el sistema esté menos cargado y en lotes para obtener un mejor rendimiento del dispositivo de *swap*.

Dado que se requiere un flujo de ejecución independiente dentro del sistema operativo, para realizar estas operaciones, se suelo usar un proceso/*thread* de núcleo denominado **demonio de paginación**.

Las páginas que están en cualquiera de las dos listas pueden recuperarse si se vuelve a acceder a las mismas antes de reutilizarse. En este caso, la rutina de fallo de página recupera la página directamente de la lista y actualiza la entrada correspondiente de la tabla de páginas para conectarla. Este fallo de página no implicaría operaciones de entrada/salida. Esta estrategia puede mejorar el rendimiento de algoritmos de reemplazo que no sean muy efectivos. Así, si el algoritmo de reemplazo decide revocar una página que en realidad está siendo usada por un proceso, se producirá inmediatamente un fallo de página que la recuperará de las listas.

Este proceso de recuperación de la página plantea un reto: ¿cómo detectar de forma eficiente si la página requerida por un fallo de página está en una de estas listas? La solución es la **cache de páginas**.

cache de páginas

Con el uso de la técnica de memoria virtual, la memoria principal se convierte, a todos los efectos, en una cache de la memoria secundaria. Por otra parte, en diversas circunstancias, el sistema operativo debe buscar si una determinada página está residente en memoria (esa necesidad se acaba de identificar dentro de la técnica del *buffering* de páginas y volverá a aparecer cuando se estudie el compartimiento de páginas). Por tanto, parece lógico que ese comportamiento de cache se implemente como tal, es decir, que se habilite una estructura de información, la **cache de páginas**, que permita gestionar las páginas de los procesos que están residentes en memoria y pueda proporcionar una manera eficiente de buscar una determinada página.

La organización de la cache se realiza de manera que se pueda buscar eficientemente una página dado un identificador único de la misma. Generalmente, este identificador corresponde al número de bloque dentro del fichero (o dispositivo de *swap*) que contiene la página. Obsérvese que las páginas a rellenar a cero que todavía no están vinculadas con el *swap* no están incluidas en la cache de páginas. Cada vez que dentro de la rutina del tratamiento del fallo de página se copia una página de un bloque de un fichero o de un dispositivo de *swap* a un marco, se incluirá en la cache asociándola con dicho bloque.

Hay que resaltar que las páginas de la cache están incluidas, además, en otras listas, tales como las gestionadas por el algoritmo de reemplazo o la de marcos libres y modificados.

En el sistema de ficheros, como se analizará en el capítulo dedicado al mismo, existe también una cache de similares características, que se suele denominar cache de bloques. Aunque el estudio de la misma se realiza en dicho capítulo, se puede anticipar que, en los sistemas operativos actuales, la tendencia es fusionar ambas caches para evitar los problemas de coherencia y de mal aprovechamiento de la memoria, debido a la duplicidad de la información en las caches.

Retención de páginas en memoria

Para acabar esta sección en la que se han presentado diversos algoritmos de reemplazo, hay que resaltar que no todas las páginas residentes en memoria son candidatas al reemplazo. Se puede considerar que algunas páginas están *atornilladas* a la memoria principal.

En primer lugar, están las páginas del propio sistema operativo. La mayoría de los sistemas operativos tienen su mapa de memoria fijo en memoria principal. El diseño de un sistema operativo en el que las páginas de su propio mapa pudieran expulsarse a memoria secundaria resultaría complejo y, posiblemente, ineficiente. Tenga en cuenta, además, que el código de la rutina de tratamiento del fallo de página, así como los datos y otras partes de código usados desde la misma, deben siempre estar residentes para evitar el interbloqueo. Lo que sí proporcionan algunos sistemas operativos es la posibilidad de que un componente del propio sistema operativo reserve una zona de memoria que pueda ser expulsada, lo que le permite usar grandes cantidades de datos sin afectar directamente a la cantidad de memoria disponible en el sistema.

Además, si se permite que los dispositivos de entrada/salida que usan DMA realicen transferencias directas a la memoria de un proceso, será necesario marcar las páginas implicadas como no reemplazables hasta que termine la operación.

Por último, algunos sistemas operativos ofrecen servicios a las aplicaciones que les permiten solicitar que una o más páginas de su mapa queden retenidas en memoria (en UNIX existe el servicio `mlock` para este fin). Este servicio puede ser útil para procesos de tiempo real que necesitan evitar que se produzcan fallos de página imprevistos. Sin embargo, el uso indiscriminado de este servicio puede afectar gravemente al rendimiento del sistema.

4.10.7. Política de actualización

Dada la enorme diferencia entre la velocidad de transferencia de la memoria principal y la de la secundaria, no es factible usar una política de actualización inmediata, utilizándose, por tanto, una política de escritura diferida: sólo se escribirá una página a memoria secundaria cuando sea expulsada de la memoria principal estando, además, modificada.

Como se ha comentado en la sección de *buffering*, la actualización se intenta hacer por lotes y cuando el sistema está menos cargado.

4.10.8. Política de reparto de espacio entre los procesos

En un sistema con multiprogramación existen varios procesos activos simultáneamente que comparten la memoria del sistema. Es necesario, por tanto, determinar cuántos marcos de página se asignan a cada proceso. Existen dos tipos de estrategias de asignación: asignación fija o asignación dinámica.

Asignación fija

Con esta estrategia, se asigna a cada proceso un número fijo de marcos de página. Normalmente, este tipo de asignación lleva asociada una estrategia de reemplazo local. El número de marcos asignados no varía, ya que un proceso sólo usa para reemplazo los marcos que tiene asignados.

La principal desventaja de esta alternativa es que no se adapta a las diferentes necesidades de memoria de un proceso a lo largo de su ejecución. Habrá fases en la que el espacio asignado se le quedará pequeño, no permitiendo almacenar simultáneamente todas las páginas que está utilizando el proceso en ese intervalo de tiempo. En contraste, existirán fases en las que el proceso no usará realmente los marcos que tiene asignados. Una propiedad positiva de esta estrategia es que el comportamiento del proceso es relativamente predecible, puesto que siempre que se ejecute con los mismos parámetros va a provocar los mismos fallos de página.

Existen diferentes criterios para repartir los marcos de las páginas entre los procesos existentes. Puede depender de múltiples factores tales como el tamaño del proceso o su prioridad. Por otra parte, cuando se usa una estrategia de asignación fija, el sistema operativo decide cuál es el número máximo de marcos asignados al proceso. Sin embargo, la arquitectura de la máquina establece el número mínimo de marcos que deben asignarse a un proceso. Por ejemplo, si la ejecución de una única instrucción puede generar cuatro fallos de página y el sistema operativo asigna tres marcos de página a un proceso que incluya esta instrucción, el proceso podría no terminar de ejecutarla. Por tanto, el número mínimo de marcos de página para una arquitectura quedará fijado por la instrucción que pueda generar el máximo número de fallos de página.

Asignación dinámica

Usando esta estrategia, el número de marcos asignados a un proceso varía según las necesidades que tenga el mismo (y posiblemente el resto de procesos del sistema) en diferentes instantes de tiempo. Con este tipo de asignación se pueden usar estrategias de reemplazo locales y globales.

- Con reemplazo local, el proceso va aumentando o disminuyendo su conjunto residente dependiendo de sus necesidades en las distintas fases de ejecución del programa.
- Con reemplazo global, los procesos compiten en el uso de memoria quitándose entre sí las páginas.

La estrategia de reemplazo global hace que el comportamiento del proceso en tiempo de ejecución no sea predecible. El principal problema de este tipo asignación es que la tasa de fallos de página de un programa puede depender de las características de los otros procesos que estén activos en el sistema.

Hiperpaginación

Si el número de marcos de página asignados a un proceso no es suficiente para almacenar las páginas a las que hace referencia frecuentemente, se producirá un número elevado de fallos de página. Esta situación se denomina **hiperpaginación** (*thrashing*). Cuando se produce la hiperpaginación, el proceso pasa más tiempo en la cola de servicio del dispositivo de *swap* que en ejecución. Dependiendo del tipo de asignación usado, este problema puede afectar a procesos individuales o a todo el sistema.

En un sistema operativo que utiliza una estrategia de asignación fija, si el número de marcos asignados al proceso no es suficiente para albergar su conjunto de trabajo en una determinada fase de su ejecución, se producirá hiperpaginación en ese proceso, lo que causará un aumento considerable de su tiempo de ejecución. Sin embargo, el resto de los procesos del sistema no se verán afectados directamente.

Con una estrategia de asignación dinámica, el número de marcos asignados a un proceso se va adaptando a sus necesidades, por lo que, en principio, no debería presentarse este problema. No obstante, si el número de marcos de página en el sistema no es suficiente para almacenar los conjuntos de trabajo de todos los procesos, se producirán fallos de página frecuentes y, por tanto, el sistema sufrirá hiperpaginación. La utilización del procesador disminuirá, puesto que el tiempo que dedica al tratamiento de los fallos de página aumenta. Como se puede observar en la figura 4.58, no se trata de una disminución progresiva, sino drástica, que se debe a que al aumentar el número de procesos, por un lado, crece la tasa de fallos de página de cada proceso (hay menos marcos de página por proceso) y, por otro

lado, aumenta el tiempo de servicio del dispositivo de paginación (crece la longitud de la cola de servicio del dispositivo).

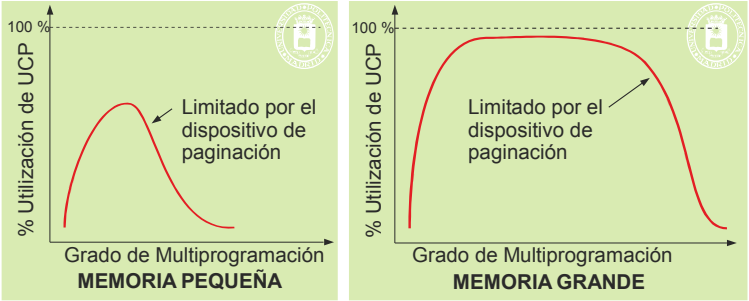


Figura 4.58 *Hiperpaginación.*

Cuando se produce esta situación se deben suspender uno o varios procesos liberando sus páginas. Es necesario establecer una estrategia de control de carga que ajuste el grado de multiprogramación en el sistema para evitar que se produzca hiperpaginación. Este mecanismo de suspensión tiene similitudes con la técnica del intercambio y, como en dicha técnica, habrá que establecer algún tipo de criterio para decidir qué procesos se deberían suspender (criterios tales como si el proceso está bloqueado, su prioridad, el número de páginas residentes, el tamaño de su mapa de memoria o el tiempo que lleva ejecutando). La reactivación de los procesos seleccionados sólo se realizará cuando haya suficientes marcos de página libres. La estrategia que decide cuándo suspender un proceso y cuándo reactivarlo se corresponde con la planificación a medio plazo presentada en el capítulo "3 Procesos". A continuación, se plantean algunas políticas de control de carga.

Estrategia del conjunto de trabajo

Como se comentó previamente, cuando un proceso tiene residente en memoria su conjunto de trabajo, se produce una baja tasa de fallos de página. Una posible estrategia consiste en determinar los conjuntos de trabajo de todos los procesos activos para intentar mantenerlos residentes en memoria principal.

Para poder determinar el conjunto de trabajo de un proceso es necesario dar una definición más formal de este término. El conjunto de trabajo de un proceso es el conjunto de páginas a las que ha accedido un proceso en las últimas n referencias. El número n se denomina la ventana del conjunto de trabajo. El valor de n es un factor crítico para el funcionamiento efectivo de esta estrategia. Si es demasiado grande, la ventana podría englobar varias fases de ejecución del proceso, llevando a una estimación excesiva de las necesidades del proceso. Si es demasiado pequeño, la ventana podría no englobar la situación actual del proceso, con lo que se generarían demasiados fallos de página.

Suponiendo que el sistema operativo es capaz de detectar cuál es el conjunto de trabajo de cada proceso, se puede especificar una estrategia de asignación dinámica con reemplazo local y control de carga.

- Si el conjunto de trabajo de un proceso decrece, se liberan los marcos asociados a las páginas que ya no están en el conjunto de trabajo.
- Si el conjunto de trabajo de un proceso crece, se asignan marcos para que puedan contener las nuevas páginas que han entrado a formar parte del conjunto de trabajo. Si no hay marcos libres, hay que realizar un control de carga, suspendiendo uno o más procesos y liberando sus páginas.

El problema de esta estrategia es cómo poder detectar cuál es el conjunto de trabajo de cada proceso. Al igual que ocurre con el algoritmo LRU, se necesitaría una MMU específica que fuera controlando las páginas a las que ha ido accediendo cada proceso durante las últimas n referencias.

Estrategia de administración basada en la frecuencia de fallos de página

Esta estrategia busca una solución más directa al problema de la hiperpaginación. Se basa en controlar la frecuencia de fallos de página de cada proceso. Como se ve en la figura 4.59, se establecen una cuota superior y otra inferior de la frecuencia de fallos de página de un proceso. Basándose en esa idea, a continuación se describe una estrategia de asignación dinámica con reemplazo local y control de carga.

- Si la frecuencia de fallos de un proceso supera el límite superior, se asignan marcos de página adicionales al proceso. Si la tasa de fallos crece por encima del límite y no hay marcos libres, se suspende algún proceso liberando sus páginas.
- Cuando el valor de la tasa de fallos es menor que el límite inferior, se liberan marcos asignados al proceso seleccionándolos mediante un algoritmo de reemplazo.

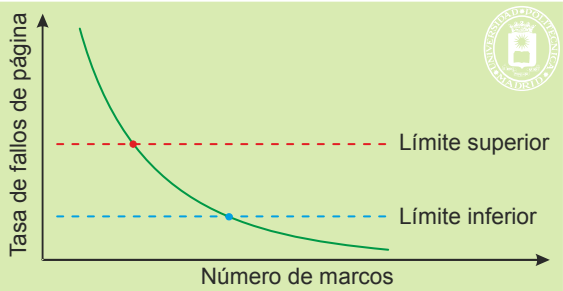


Figura 4.59 *Estrategia de administración basada en la frecuencia de fallos de página.*

Estrategia de control de carga para algoritmos de reemplazo globales

Los algoritmos de reemplazo globales no controlan la hiperpaginación. Incluso aunque se pudiera utilizar el algoritmo óptimo, el problema persistiría, puesto que dicho algoritmo seleccionaría la página menos útil, pero, en estas circunstancias, esa página también es útil. Necesitan trabajar conjuntamente con un algoritmo de control de carga. Normalmente, se usan soluciones de carácter empírico, que detectan síntomas de que el sistema está evolucionando hacia la hiperpaginación. Así, si la tasa de paginación en el sistema es demasiado alta y el número de marcos libres está frecuentemente por debajo del umbral mínimo, se considera que el sistema está en estado de hiperpaginación y se suspende uno o más procesos.

4.10.9. Gestión del espacio de *swap*

Un dispositivo de *swap* se implementa sobre una unidad de disco o una **partición** de la misma, sobre un **fichero** y, en algunos casos, sobre una **memoria RAM**. Normalmente, los sistemas operativos ofrecen la posibilidad de utilizar múltiples dispositivos de *swap*, permitiendo, incluso, añadir dispositivos de *swap* dinámicamente.

Sin embargo, hay que tener en cuenta que el acceso a los ficheros es más lento que el acceso directo a los dispositivos. En cualquier caso, esta posibilidad es interesante, ya que alivia al administrador de la responsabilidad de configurar correctamente a priori el dispositivo de *swap*, puesto que si hay necesidad, se puede añadir más espacio de *swap* en tiempo de ejecución. Habitualmente, también es posible que el administrador defina el modo de uso de los dispositivos de *swap*, pudiendo establecer políticas tales como no usar un dispositivo hasta que los otros estén llenos, o repartir cíclicamente las páginas expulsadas entre los dispositivos de *swap* existentes.

La estructura interna de un dispositivo de *swap* es muy sencilla: una cabecera y un conjunto de bloques. La cabecera incluye algún tipo de información de control, como, por ejemplo, si hay sectores de disco erróneos dentro de la misma. No es necesario que incluya información del estado de los bloques, puesto que el dispositivo de *swap* sólo se usa mientras el sistema está arrancado. Por tanto, no hay que mantener ninguna información cuando el sistema se apaga. El sistema operativo usa un mapa de bits en memoria para conocer si está libre u ocupado cada bloque del *swap*.

El sistema operativo debe gestionar el espacio de *swap* reservando y liberando zonas del mismo según evolucione el sistema. Existen básicamente dos alternativas a la hora de asignar espacio de *swap* durante la creación de una nueva región:

- **Con preasignación de *swap***. Cuando se crea una región privada o sin soporte, se reserva espacio de *swap* para la misma. Con esta estrategia, cuando se expulsa una página ya tiene reservado espacio en *swap* para almacenar su contenido. En algunos sistemas, más que realizar una reserva explícita de bloques de *swap*, se lleva una cuenta de cuántos hay disponibles, de manera que al crear una región que requiera el uso del *swap*, se descuenta la cantidad correspondiente al tamaño de la misma del total de espacio de *swap* disponible.
- **Sin preasignación de *swap***. Cuando se crea una región, no se hace ninguna reserva en el *swap*. Sólo se reserva espacio en el *swap* para una página cuando es expulsada por primera vez. En este caso, una página puede estar en una de las siguientes situaciones:
  - *swap*
  - un marco de memoria
  - un fichero proyectado
  - a rellenar a 0

La primera estrategia conlleva un peor aprovechamiento de la memoria secundaria, puesto que toda página debe tener reservado espacio en ella. Sin embargo, la preasignación presenta la ventaja de que con ella se detecta anticipadamente si no queda espacio en *swap*. Si al crear un proceso no hay espacio en *swap*, éste no se crea. Con un esquema sin preasignación, esta situación se detecta cuando se va a expulsar una página y no hay sitio para ella. En ese momento habría que abortar el proceso aunque ya hubiera realizado parte de su labor.

Sólo las páginas de las regiones privadas o sin soporte usan el *swap* para almacenarse cuando son expulsadas estando modificadas. En el caso de una página de una región compartida con soporte en un fichero, no se usa espacio de *swap* para almacenarla, sino que se utiliza directamente el fichero que la contiene como almacenamiento secundario.

Dado que puede haber múltiples entradas de tablas de páginas que hacen referencia al mismo bloque de *swap*, el sistema operativo gestionará un **contador de referencias** por cada bloque de *swap*, de manera que cuando el mismo valga cero, el bloque estará libre.

4.11. SERVICIOS DE GESTIÓN DE MEMORIA

Entre ese número relativamente reducido de servicios, en esta sección se ha considerado que los de mayor interés se pueden agrupar en tres categorías:

- **Servicios de proyección de ficheros**, que permiten incluir en el mapa de memoria de un proceso un fichero o parte del mismo. Bajo esta categoría existirán, básicamente, dos servicios:
  - **Proyección de un fichero**. Con esta operación se crea una región asociada al objeto de memoria almacenado en el fichero. Normalmente, se pueden especificar algunas propiedades de esta nueva región. Por ejemplo, el tipo de protección o si la región es privada o compartida.
  - **Desproyección de un fichero**. Este servicio elimina una proyección previa o parte de la misma.
- **Servicios de montaje explícito de bibliotecas**, que permiten que un programa cargue en tiempo de ejecución una biblioteca dinámica y use la funcionalidad proporcionada por la misma. En esta categoría se englobarían, básicamente, tres servicios:
  - **Carga de la biblioteca**. Este servicio realiza la carga de la biblioteca, llevando a cabo todas las operaciones de montaje requeridas.
  - **Acceso a un símbolo de la biblioteca**. Con esta operación, el programa puede tener acceso a uno de los símbolos exportados por la biblioteca, ya sea éste una función o una variable.
  - **Descarga de la biblioteca**. Este servicio elimina la biblioteca del mapa del proceso.
- **Servicios para bloquear páginas en memoria principal**. Permiten que el superusuario bloquee páginas en memoria principal.

En las siguientes secciones, se muestran los servicios proporcionados por UNIX y Windows dentro de estas categorías.

4.11.1. Servicios UNIX de proyección de ficheros

Los servicios de gestión de memoria más frecuentemente utilizados son los que permiten la proyección y desproyección de ficheros (`mmap`, `munmap`).

◙ caddr_t **mmap** (caddr_t direccion, size_t longitud, int protec, int indicadores, int descriptor, off_t despl);

Este servicio proyecta el fichero especificado creando una región con las características indicadas en la llamada. El primer parámetro indica la dirección del mapa donde se quiere que se proyecte el fichero. Generalmente, se especifica un valor nulo para indicar que se prefiere que sea el sistema el que decida donde proyectar el fichero. En cualquier caso, la función devolverá la dirección de proyección utilizada.

El parámetro `descriptor` corresponde con el descriptor del fichero que se pretende proyectar (que debe estar previamente abierto), y los parámetros `despl` y `longitud` establecen qué zona del fichero se proyecta: desde la posición `despl` hasta `desp` + `longitud`.

El argumento `protec` establece la protección de la región, que puede ser de lectura (`PROT_READ`), de escritura (`PROT_WRITE`), de ejecución (`PROT_EXEC`), o cualquier combinación de ellas. Esta protección debe ser compatible con el modo de apertura del fichero. Por último, el parámetro `indicadores` permite establecer ciertas propiedades de la región:

- `MAP_SHARED`. La región es compartida. Las modificaciones sobre la región afectarán al fichero. Un proceso hijo compartirá esta región con el padre.
- `MAP_PRIVATE`. La región es privada. Las modificaciones sobre la región no afectarán al fichero. Un proceso hijo no compartirá esta región con el padre, sino que obtendrá un duplicado de la misma.
- `MAP_FIXED`. El fichero debe proyectarse justo en la dirección especificada en el primer parámetro. Esta opción se utiliza, por ejemplo, para cargar el código de una biblioteca dinámica, si en el sistema se utiliza un esquema de gestión de bibliotecas dinámicas, tal que cada biblioteca tiene asignado un rango de direcciones fijo.

En el caso de que se quiera proyectar una región sin soporte (región anónima), en algunos sistemas se puede especificar el valor `MAP_ANON` en el parámetro `indicadores`. Otros sistemas UNIX no ofrecen esta opción, pero permiten proyectar el dispositivo `/dev/zero` para lograr el mismo objetivo. Esta opción se puede usar para cargar la región de datos sin valor inicial de una biblioteca dinámica.

◙ int **munmap**(caddr_t direccion, size_t longitud);

Este servicio elimina una proyección previa o parte de la misma. Los parámetros `direccion` y `longitud` definen la región (o la parte de la región) que se quiere eliminar del mapa del proceso.

Antes de presentar ejemplos del uso de estos servicios, hay que aclarar que se utilizan conjuntamente con los servicios de manejo de ficheros que se presentarán en el capítulo que trata este tema. Por ello, para una buena comprensión de los ejemplos, se deben estudiar también los servicios explicados en ese capítulo.

A continuación, se muestran dos ejemplos del uso de estas funciones. El primero es el programa 4.1 que cuenta cuántas veces aparece un determinado carácter en un fichero utilizando la técnica de proyección en memoria.

Programa 4.1 Programa que cuenta el número de apariciones de un carácter en un fichero.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
    int i, fd, contador=0;
    char caracter;
    char *org, *p;
    struct stat bstat;

    if (argc!=3) {
        fprintf (stderr, "Uso: %s caracter fichero\n", argv[0]);
        return 1;
    }
    /* Para simplificar, se supone que el carácter a contar corresponde con el primero del primer argumento */
    caracter=argv[1][0];
    /* Abre el fichero para lectura */
    if ((fd=open(argv[2], O_RDONLY))<0) {
        perror("No puede abrirse el fichero");
        return 1;
    }
    /* Averigua la longitud del fichero */
    if (fstat(fd, &bstat)<0) {
        perror("Error en fstat del fichero");
        close(fd);
        return 1;
    }
    /* Se proyecta el fichero */
    if ((org=mmap((caddr_t) 0, bstat.st_size, PROT_READ, MAP_SHARED, fd, 0)) == MAP_FAILED) {
        perror("Error en la proyeccion del fichero");
        close(fd);
        return 1;
    }
    /* Se cierra el fichero */
    close(fd);
    /* Bucle de acceso */
    p=org;
    for (i=0; i<bstat.st_size; i++)
        if (*p++==caracter) contador++;
    /* Se elimina la proyeccion */
    munmap(org, bstat.st_size);
    printf("%d\n", contador);
    return 0;
}
```

El segundo ejemplo se corresponde con el programa 4.2 que usa la técnica de proyección para realizar la copia de un fichero. Observe el uso del servicio `ftruncate` para asignar espacio al fichero destino.

Programa 4.2 Programa que copia un fichero.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <stdio.h>
```

```
#include <unistd.h>
void main(int argc, char *argv[]) {
    int i, fdo, fdd;
    char *org, *dst, *p, *q;
    struct stat bstat;

    if (argc!=3) {
        fprintf (stderr, "Uso: %s orig dest\n", argv[0]);
        return 1;
    }
    /* Abre el fichero origen para lectura */
    if ((fdo=open(argv[1], O_RDONLY))<0) {
        perror("No puede abrirse el fichero origen");
        return 1;
    }
    /* Crea el fichero destino */
    if ((fdd=open(argv[2], O_CREAT|O_TRUNC|O_RDWR, 0640))<0) {
        perror("No puede crearse el fichero destino");
        close(fdo);
        return 1;
    }
    /* Averigua la longitud del fichero origen */
    if (fstat(fdo, &bstat)<0) {
        perror("Error en fstat del fichero origen");
        close(fdo); close(fdd); unlink(argv[2]);
        return 1;
    }

    /* Establece que la longitud del fichero destino es igual a la del origen.*/
    if (ftruncate(fdd, bstat.st_size)<0) {
        perror("Error en ftruncate del fichero destino");
        close(fdo); close(fdd); unlink(argv[2]);
        return 1;
    }
    /* Se proyecta el fichero origen */
    if ((org=mmap((caddr_t) 0, bstat.st_size, PROT_READ, MAP_SHARED, fdo, 0)) == MAP_FAILED) {
        perror("Error en la proyeccion del fichero origen");
        close(fdo); close(fdd); unlink(argv[2]);
        return 1;
    }
    /* Se proyecta el fichero destino */
    if ((dst=mmap((caddr_t) 0, bstat.st_size, PROT_WRITE, MAP_SHARED, fdd, 0)) == MAP_FAILED) {
        perror("Error en la proyeccion del fichero destino");
        close(fdo); close(fdd); unlink(argv[2]);
        return 1;
    }
    /* Se cierran los ficheros */
    close(fdo);
    close(fdd);
    /* Bucle de copia */
    p=org; q=dst;
    for (i=0; i<bstat.st_size; i++)
        *q++= *p++;
    /* Se eliminan las proyecciones */
    munmap(org, bstat.st_size);
    munmap(dst, bstat.st_size);
    return 0;
}
```

4.11.2. Servicios UNIX de carga de bibliotecas

Por lo que se refiere a esta categoría, la mayoría de los sistemas UNIX ofrece las funciones `dlopen`, `dlsym` y `dlclose`.

- void ***dlopen**(const char *biblioteca, int indicadores);

La rutina `dlopen` realiza la carga y montaje de una biblioteca dinámica. Recibe como argumentos el nombre de la biblioteca y el valor `indicadores`, que determina diversos aspectos vinculados con la carga de la biblioteca. Como resultado, esta función devuelve un descriptor que identifica dicha biblioteca cargada. En cuanto al segundo parámetro, aunque permite especificar distintas posibilidades a la hora de cargarse la biblioteca, de forma obligatoria, sólo es necesario indicar uno de los dos siguientes valores: `RTLD_LAZY`, que indica que las referencias a símbolos que estén pendientes de resolver dentro de la biblioteca no se llevarán a cabo hasta que sea estrictamente necesario, o `RTLD_NOW`, que especifica que durante la propia llamada `dlopen` se resuelvan todas las referencias pendientes que haya dentro de la biblioteca que se desea cargar. Recuerde que estos dos modos de operación se analizaron cuando se estudiaron los aspectos de implementación de las bibliotecas dinámicas.

- void ***dlsym**(void *descriptor, char *simbolo);

La función `dlsym` permite acceder a uno de los símbolos exportados por la biblioteca. Recibe como parámetros el descriptor de una biblioteca dinámica previamente cargada y el nombre de un símbolo (una variable o una función). Este servicio se encarga de buscar ese símbolo dentro de la biblioteca especificada y devuelve la dirección de memoria donde se encuentra dicho símbolo. Como primer parámetro, en vez de un descriptor de biblioteca, se puede especificar la constante `RTLD_NEXT`. Si desde una biblioteca dinámica se invoca a la función `dlsym` especificando esa constante, el símbolo se buscará en las siguientes bibliotecas dinámicas del proceso a partir de la propia biblioteca que ha invocado la función `dlsym`, en vez de buscarlo empezando por la primera biblioteca dinámica especificada en el mandato de montaje. Esta característica suele usarse cuando se pretende incluir una función de interposición que, además, después invoque la función original.

- int **dlclose**(void *descriptor);

La rutina `dlclose` descarga la biblioteca especificada por el descriptor.

A continuación, se incluye el programa 4.3 que muestra un ejemplo del uso de estas funciones. El programa recibe como argumentos el nombre de una biblioteca dinámica, el nombre de una función y el parámetro que se le quiere pasar a la función. Esta función debe ser exportada por la biblioteca, debe tener un único parámetro de tipo cadena de caracteres y devolver un valor entero. El programa carga la biblioteca e invoca la función pasándole el argumento especificado, imprimiendo el resultado devuelto por la misma.

Programa 4.3 Programa que ejecuta una función exportada por una biblioteca dinámica.

```
#include <stdio.h>
#include <unistd.h>
#include <dlfcn.h>
#include <string.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    void *descriptor_bib;
    int (*procesar)(char *);
    int resultado;
    if (argc!=4) {
        fprintf(stderr, "Uso: %s biblioteca funcion argumento\n", argv[0]);
        return 1;
    }
    /* Se carga la biblioteca dinámica */
    if (!(descriptor_bib=dlopen(argv[1], RTLD_LAZY))) {
        fprintf(stderr, "Error cargando biblioteca: %s\n", dlerror());
        return 1;
    }
    /* Busca el símbolo */
    if (!(procesar=dlsym(descriptor_bib, argv[2]))) {
        fprintf(stderr, "Error: biblioteca no incluye la funcion\n");
        return 1;
    }
    /* Finalmente, llamamos a la función */
    resultado=procesar(argv[3]);
    printf("Resultado: %d\n", resultado);
    /* Se descarga la biblioteca */
    dlclose(descriptor_bib);
    return 0;
}
```

4.11.3. Servicios UNIX para bloquear páginas en memoria principal

En esta categoría, encontramos los servicios `mlock` y `munlock`, interesantes cuando no se puede tolerar el retardo producido por fallos de página, por ejemplo, en sistemas de tiempo real.

- int mlock(void *addr, int len);

Este servicio bloquea todas las páginas que tengan posiciones contenidas entre `addr` y `addr + len`, como se muestra en la figura 4.60.

El servicio está reservado al superusuario.

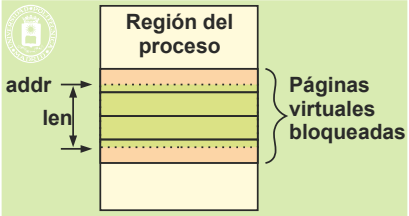


Figura 4.60 *El servicio* `mlock` *bloquea páginas en memoria principal.*

- int munlock(void *addr, int len);

Este servicio desbloquea las páginas que contengan parte del espacio de direcciones comprendido entre `addr` y `addr + len`.

4.11.4. Servicios Windows de proyección de ficheros

A diferencia de UNIX, en Windows, la proyección de un fichero se realiza en dos pasos. En primer lugar, hay que crear una proyección del fichero y, posteriormente, se debe crear una región en el proceso que esté asociada a la proyección.

- HANDLE **CreateFileMapping**(HANDLE fich, LPSECURITY_ATTRIBUTES segur, DWORD prot, DWORD tamaño_max_alta, DWORD tamaño_max_baja, LPCTSTR nombre_proy);

Esta función crea una proyección de un fichero. Como resultado de la misma, devuelve un identificador de la proyección. Recibe como parámetros el nombre del fichero, un valor de los atributos de seguridad, la protección, el tamaño del objeto a proyectar (especificando la parte alta y la parte baja de este valor en dos parámetros independientes) y un nombre para la proyección.

En cuanto a la protección, puede especificarse de sólo lectura (`PAGE_READONLY`), de lectura y escritura (`PAGE_READWRITE`) o privada (PAGE_WRITECOPY). Con respecto al tamaño, en el caso de que el fichero pueda crecer, se debe especificar el tamaño esperado para el fichero. Si se especifica un valor 0, se usa el tamaño actual del fichero. Por último, por lo que se refiere al nombre de la proyección, éste permite a otros procesos acceder a la misma. Si se especifica un valor nulo, no se asigna nombre a la proyección.

- LPVOID **MapViewOfFile**(HANDLE id_proy, DWORD acceso, DWORD desp_alta, DWORD desp_baja, DWORD tamaño);

Esta función crea una región en el mapa del proceso que queda asociada con una proyección previamente creada. Al completarse, esta rutina devuelve la dirección del mapa donde se ha proyectado la región. Recibe como parámetros el identificador de la proyección devuelto por `CreateFileMapping`, el tipo de acceso solicitado (`FILE_MAP_WRITE`, `FILE_MAP_READ` y `FILE_MAP_ALL_ACCESS`), que tiene que ser compatible con la protección especificada en la creación, el desplazamiento con respecto al inicio del fichero a partir del que se realiza la proyección, y el tamaño de la zona proyectada (el valor cero indica todo el fichero).

- BOOL **UnmapViewOfFile**(LPVOID dir);

Esta función elimina la proyección del fichero. El parámetro indica la dirección de comienzo de la región que se quiere eliminar.

A continuación, se muestran los dos mismos ejemplos que se plantearon en la sección dedicada a este tipo de servicios en UNIX. Como se comentó en dicha sección, es necesario conocer los servicios básicos de ficheros para entender completamente los siguientes programas. El primero es el programa 4.4 que cuenta cuántas veces aparece un determinado carácter en un fichero usando la técnica de proyección en memoria.

Programa 4.4 Programa que cuenta el número de apariciones de un carácter en un fichero.

```
#include <windows.h>
#include <stdio.h>

int main (int argc, char *argv[]) {
    HANDLE hFich, hProy;
```

```
    LPSTR base, puntero;
    DWORD tam;
    int contador=0;
    char caracter;

    if (argc!=3) {
        fprintf (stderr, "Uso: %s caracter fichero\n", argv[0]);
        return 1;
    }
    /* Para simplificar, se supone que el carácter a contar corresponde con el primero del primer argumento */
    caracter=argv[1][0];
    /* Abre el fichero para lectura */
    hFich = CreateFile(argv[2], GENERIC_READ, 0, NULL, OPEN_EXISTING,
            FILE_ATTRIBUTE_NORMAL, NULL);
    if (hFich == INVALID_HANDLE_VALUE) {
        fprintf(stderr, "No puede abrirse el fichero\n");
        return 1;
    }
    /* se crea la proyección del fichero */
    hProy = CreateFileMapping(hFich, NULL, PAGE_READONLY, 0, 0, NULL);
    if (hProy == INVALID_HANDLE_VALUE) {
        fprintf(stderr, "No puede crearse la proyección\n");
        return 1;
    }
    /* se realiza la proyección */
    base = MapViewOfFile(hProy, FILE_MAP_READ, 0, 0, 0);
    tam = GetFileSize(hFich, NULL);
    /* bucle de acceso */
    puntero = base;
    while (puntero < base + tam)
        if (*puntero++==caracter) contador++;
    printf("%d\n", contador);
    /* se elimina la proyección y se cierra el fichero */
    UnmapViewOfFile(base);
    CloseHandle(hProy);
    CloseHandle(hFich);
    return 0;
}
```

El segundo ejemplo se corresponde con el programa 4.5 que usa la técnica de proyección para realizar la copia de un fichero.

Programa 4.5 Programa que copia un fichero.

```
#include <windows.h>
#include <stdio.h>

int main (int argc, char *argv[]){
    HANDLE hEnt, hSal;
    HANDLE hProyEnt, hProySal;
    LPSTR base_orig, puntero_orig;
    LPSTR base_dest, puntero_dest;
    DWORD tam;

    if (argc!=3){
        fprintf (stderr, "Uso: %s origen destino\n", argv[0]);
        return 1;
    }

    /* se abre el fichero origen */
    hEnt = CreateFile (argv[1], GENERIC_READ, 0, NULL, OPEN_EXISTING,
            FILE_ATTRIBUTE_NORMAL, NULL);
    if (hEnt == INVALID_HANDLE_VALUE) {
        fprintf(stderr, "No puede abrirse el fichero origen\n");
        return 1;
```

```
    }

    /* se crea la proyección del fichero origen */
    hProyEnt = CreateFileMapping(hEnt, NULL, PAGE_READONLY, 0, 0,
                        NULL);
    if (hProyEnt == INVALID_HANDLE_VALUE) {
        fprintf(stderr, "No puede crearse proyección del fichero origen\n");
        return(1);
    }
    /* se proyecta el fichero origen */
    base_orig = MapViewOfFile(hProyEnt, FILE_MAP_READ, 0, 0, 0);
    tam = GetFileSize (hEnt, NULL);
    /* se crea el fichero destino */
    hSal = CreateFile(argv[2], GENERIC_READ | GENERIC_WRITE,
            0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    if (hSal == INVALID_HANDLE_VALUE) {
        fprintf(stderr, "No puede crearse el fichero destino\n");
        return 1;
    }
    /* se crea la proyección del fichero destino */
    hProySal = CreateFileMapping(hSal, NULL, PAGE_READWRITE, 0, tam, NULL);
    if (hProySal == INVALID_HANDLE_VALUE) {
        fprintf(stderr, "No puede crearse proyección del fichero destino");
        return 1;
    }
    /* se proyecta fichero destino */
    base_dest = MapViewOfFile(hProySal, FILE_MAP_WRITE, 0, 0, tam);
    /* bucle de copia */
    puntero_orig = base_orig;
    puntero_dest = base_dest;
    for ( ; puntero_orig < base_orig + tam; puntero_orig++, puntero_dest++)
        *puntero_dest = *puntero_orig;
    /* se eliminan proyecciones y se cierran ficheros */
    UnmapViewOfFile(base_dest);
    UnmapViewOfFile(base_orig);
    CloseHandle(hProyEnt);
    CloseHandle(hEnt);
    CloseHandle(hProySal);
    CloseHandle(hSal);
    return 0;
}
```

4.11.5. Servicios Windows de carga de bibliotecas

En esta categoría, Windows ofrece las funciones `LoadLibrary`, `GetProcAddress` y `FreeLibrary`.

- HINSTANCE **LoadLibrary**(LPCTSTR biblioteca);

La rutina `LoadLibrary` realiza la carga y montaje de una biblioteca dinámica. Recibe como argumento el nombre de la biblioteca.

- FARPROC **GetProcAddress**(HMODULE descriptor, LPCSTR simbolo);

La función `GetProcAddress` permite acceder a uno de los símbolos exportados por la biblioteca. Recibe como parámetros el descriptor de una biblioteca dinámica previamente cargada y el nombre de un símbolo. Este servicio se encarga de buscar ese símbolo dentro de la biblioteca especificada y devuelve la dirección de memoria donde se encuentra dicho símbolo.

- BOOL **FreeLibrary**(HINSTANCE descriptor);

La rutina `FreeLibrary` descarga la biblioteca especificada por el descriptor.

A continuación, en el programa 4.6 se muestra el mismo ejemplo que se planteó para UNIX en el programa 5.7.

Programa 4.6 Programa que ejecuta una función exportada por una biblioteca dinámica.

```
#include <windows.h>
```

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    HINSTANCE descriptor_bib;
    int (*procesar)(LPCTSTR);
    int resultado;

    if (argc!=4) {
        fprintf(stderr, "Uso: %s biblioteca funcion argumento\n", argv[0]);
        return 1;
    }
    /* Se carga la biblioteca dinámica */
    descriptor_bib = LoadLibrary(argv[1]);
    if (descriptor_bib == NULL)      {
        fprintf(stderr, "Error cargando biblioteca %s\n", argv[1]);
        return 1;
    }
    procesar = (int(*)(LPCTSTR))GetProcAddress(descriptor_bib, argv[2]);
    if (procesar == NULL) {
        /* La función no existe */
        fprintf(stderr, "Error: biblioteca no incluye la funcion\n");
        return 1;
    }
    /* Finalmente, llamamos a la función */
    resultado=procesar(argv[3]);
    printf("Resultado: %d\n", resultado);
    /* Se descarga la biblioteca */
    FreeLibrary(descriptor_bib);
    return 0;
}
```

4.11.6. Servicios Windows para bloquear páginas en memoria principal

Consideraremos los servicios `VirtualLock` y `VirtualLock`.

- BOOL WINAPI VirtualLock(_In_ LPVOID lpAddress, _In_ SIZE_T dwSize);

Este servicio bloquea todas las páginas que tengan posiciones contenidas entre `lpAddress` y `lpAddress + dwSize`. Sólo se permite bloquear unas pocas páginas, para bloquear un mayor número es necesario previamente llamar a `SetProcessWorkingSetSize`.

- BOOL WINAPI VirtualUnlock(_In_ LPVOID lpAddress, _In_ SIZE_T dwSize);

Este servicio desbloquea las páginas que contengan parte del espacio de direcciones comprendido entre `lpAddress` y `lpAddress + dwSize`.

4.12. LECTURAS RECOMENDADAS

Dada la variedad de temas abordados en este capítulo, al intentar dar una visión integral de la gestión de memoria, se recomiendan lecturas de muy diversa índole. Para profundizar en los aspectos relacionados con el ciclo de vida de un programa, se recomienda al lector interesado la consulta del libro [Levine, 1999]. En cuanto a la gestión del *heap* y, en general, el problema general de la asignación de espacio, es aconsejable el artículo [Wilson, 1995]. Dada la fuerte dependencia del *hardware* que existe en el sistema de memoria, es obligado hacer alguna referencia a documentación sobre los esquemas *hardware* de gestión de memoria ([Jacob, 1998]). Asimismo, se aconseja la revisión de documentación sobre aspectos menos convencionales, ya sea por su carácter novedoso o por ser avanzados, tales como nuevos algoritmos de reemplazo de memoria virtual ([Megiddo, 2004]) o sistemas basados en un espacio de direcciones único ([Chase, 1994]).

Todos los libros generales de sistemas operativos (como, por ejemplo, [Silberschatz, 2005], [Stallings, 2005], [Nutt, 2004] y [Tanenbaum, 2001]) incluyen uno o más capítulos dedicados a la gestión de memoria, aunque, en nuestra opinión, seguramente interesada, no usan el enfoque integral de la gestión de memoria que se ha utilizado en este capítulo, por lo que su extensión es menor que la del presente capítulo.

Para ampliar conocimientos sobre la implementación de la gestión de la memoria en diversos sistemas operativos, en el caso de Linux, se pueden consultar [Gorman, 2004], [Mosberger, 2002] y [Bovet, 2005], para UNIX BSD, [McKusick, 2004], y, por lo que se refiere a Windows, [Russinovich, 2005]. Por el impacto y la innovación que causaron en su momento en el campo de la gestión de memoria, son de especial interés los sistemas operativos MULTICS [Organick, 1972] y Mach [Rashid, 1988].

En cuanto a los servicios de gestión de memoria, en [Stevens, 1999] se presentan los servicios de UNIX y en [Hart, 2004] los de Windows.

4.13. EJERCICIOS

1. ¿Cuál de las siguientes técnicas favorece la proximidad de referencias?
- Un programa *multithread*.
- El uso de listas.
- La programación funcional.
- La programación estructurada.

2. Considere un sistema de paginación con un tamaño de página *P*. Especifique cuál sería la fórmula que determina la dirección de memoria física *F* a partir de la dirección virtual *D*, siendo *MARCO(X)* una función que devuelve qué número de marco está almacenado en la entrada *X* de la tabla de páginas.

3. ¿Es siempre el algoritmo LRU mejor que el FIFO? En caso de que sea así, plantee una demostración. En caso negativo, proponga un contraejemplo.

4. En el lenguaje C se define el calificador `volatile` aplicable a variables. La misión de este calificador es evitar problemas de coherencia en aquellas variables a las que se accede tanto desde el flujo de ejecución normal como desde flujos asíncronos, como por ejemplo una rutina asociada a una señal UNIX. Analice los tipos de problemas que podrían aparecer y proponga un método para resolver los problemas identificados para las variables etiquetadas con este calificador.

5. Algunas MMU no proporcionan un bit de referencia para la página. Proponga una manera de simularlo. Una pista: Se pueden forzar fallos de página para detectar accesos a una página.

6. Algunas MMU no proporcionan un bit de página modificada. Proponga una manera de simularlo.

7. Escriba un programa que use los servicios de proyección de ficheros de UNIX para comparar dos ficheros.

8. Escriba un programa que use los servicios de proyección de ficheros de Windows para comparar dos ficheros.

9. Determine qué número de fallos de página se producen al utilizar el algoritmo FIFO teniendo 3 marcos y cuántos con 4 marcos. Compárelo con el algoritmo LRU. ¿Qué caracteriza a los algoritmos de reemplazo de pila?

10. La secuencia que se utiliza habitualmente como ejemplo de la anomalía de Belady es la siguiente:

1 2 3 4 1 2 5 1 2 3 4 5

11. Suponiendo que se utiliza un sistema sin *buffering* de páginas, proponga ejemplos de las siguientes situaciones:
d) Fallo de página sin operaciones de E/S.
e) Fallo de página con sólo una operación de lectura.
f) Fallo de página con sólo una operación de escritura.
g) Fallo de página con una operación de lectura y una de escritura.

12. Repita el ejercicio anterior, suponiendo un sistema con *buffering* de páginas.

13. Considere un sistema de memoria virtual sin *buffering* de páginas. Realice un análisis de cómo evoluciona una página en este sistema dependiendo de la región a la que pertenece. Estudie los siguientes tipos:
a) Página de código.
b) Página de datos con valor inicial.
c) Página de datos sin valor inicial.
d) Página de un fichero proyectado.
e) Página de zona de memoria compartida.

14. Resuelva el ejercicio anterior suponiendo que hay *buffering* de páginas.

15. Como se comentó en la explicación del algoritmo de reemplazo LRU, el tiempo que se debe usar para seleccionar la página menos recientemente usada es el tiempo lógico de cada proceso y no el tiempo real. Modifique la implementación basada en contadores propuesta en el texto para que tenga en cuenta esta consideración.

16. En el texto sólo se ha planteado un bosquejo del algoritmo de reemplazo LFU. Explique cómo se podría diseñar una MMU que usara este algoritmo. Acto seguido, describa una aproximación del mismo usando sólo un bit de referencia.

17. Complete la rutina de tratamiento de fallo de página presentada en la sección "4.10.4 Política de extracción" con todos los aspectos que se han ido añadiendo en las secciones posteriores (cache de páginas y *buffering*, utilización de preasignación de *swap* o no, creación de tablas de páginas por demanda, expansión implícita de la pila, etc.).

18. Algunas versiones de UNIX realizan la carga de las bibliotecas dinámicas usando el servicio `mmap`. Explique qué parámetros deberían especificarse para cada una de las secciones de una biblioteca.

19. Acceda en un sistema Linux al fichero `/proc/self/maps` y analice cuál es su contenido. ¿A qué estructura de datos del sistema de gestión de memoria corresponde el contenido de este fichero?

20. En Windows se pueden crear múltiples *heaps*. Analice en qué situaciones puede ser interesante esta característica.

21. Algunas versiones de UNIX ofrecen una llamada denominada `vfork` que crea un hijo que utiliza directamente el mapa de memoria del proceso padre, que se queda bloqueado hasta que el hijo ejecuta una llamada `exec` o termina. En ese momento el padre recupera su mapa. Analice qué ventajas y desventajas presenta el uso de este nuevo servicio frente a la utilización del `fork` convencional. En este análisis suponga primero que el `fork` se implementa sin usar la técnica COW para, a continuación, considerar que sí se utiliza.

22. Analice qué puede ocurrir en un sistema que utiliza paginación por demanda si se recompila un programa mientras se ejecuta. Proponga soluciones a los problemas que pueden surgir en esta situación.

23. En UNIX se define el servicio `msync` que permite forzar la escritura inmediata de una región en su soporte. ¿En qué situaciones puede ser interesante usar esta función?

24. Analice qué situaciones se pueden producir en el tratamiento de un fallo de TLB en un sistema que tiene una gestión *software* de la TLB.

25. Con el uso de la técnica de proyección de ficheros se produce una cierta unificación entre el sistema de ficheros y la gestión de memoria. Puesto que, como se verá en el capítulo dedicado a los ficheros, el sistema de ficheros usa una cache de bloques con escritura diferida para acelerar el acceso al disco. Analice qué tipo de inconsistencias pueden producirse si se accede a un fichero utilizando una proyección y los servicios convencionales del sistema de ficheros.

26. Analice el uso de la técnica del COW para optimizar una función de paso de mensajes entre los procesos de una misma máquina.

27. Sea un proceso en cuyo mapa de memoria existe una región privada con soporte en fichero. El proceso va invocar la llamada `exec`. ¿En qué ubicaciones pueden estar las páginas de la región en el instante previo? ¿Qué tratamiento se realiza sobre esa región durante la llamada `exec`?

28. Basándose en la manera como se utilizan habitualmente las llamadas `fork` y `exec` en las aplicaciones, algunos proponen que sería más eficiente que después del `fork` se ejecutara primero el proceso hijo en vez del padre. Analice en qué puede basarse esta propuesta.

29. Considere un proceso *Pr1* en cuyo mapa de memoria se incluye una región privada con soporte en fichero, que está formada por 3 páginas. Suponga que se ejecuta la traza que se muestra a continuación y que después de ejecutar esta traza entran a ejecutar otros procesos que expulsan las páginas de estos procesos. Se debe calcular cuántos bloques de *swap* se dedican en total a esta región y, para cada proceso, cuál es la ubicación de cada página de la región, identificando en qué bloque del fichero o del *swap* está almacenada (numere los bloques del *swap* como considere oportuno).

a) Pr1: lee de la página 1.
b) Pr1: escribe en las páginas 2 y 3.
c) Pr1: `fork` (crea Pr2).
d) Pr1: escribe en las páginas 1 y 2.
e) Pr2: escribe en la página 2.
f) Pr2: `fork` (crea Pr3).
g) Pr2: escribe en las páginas 1 y 3.
h) Pr3: lee de las páginas 1, 2 y 3.

30. Muchos sistemas operativos mantienen una estadística de cuántos fallos de página se producen en el sistema que no implican una operación de lectura de disco (en Linux se denominan *minor faults* y en Windows *soft faults*). Suponga que se utiliza un sistema de memoria virtual basado en paginación por demanda, sin *buffering* de páginas. Analice si se produciría un fallo de página (ya sea convencional o el asociado al COW) y, en caso afirmativo, si implicaría una lectura de disco o no, para cada una de las situaciones que se plantean a continuación. En caso de que haya lectura, especifique si es de un fichero o de *swap*; en caso de que no la haya, explique si se requiere un marco libre para servir el fallo y qué información se almacena en el mismo.

a) Después de reservar memoria dinámica, el proceso escribe en la misma.
b) Después de llevar a cabo una llamada a procedimiento que hace que la región de pila se expanda, el proceso modifica una variable local.
c) El proceso lee una variable global sin valor inicial que modificó antes, pero cuya página no está actualmente presente.
d) Un proceso escribe en una variable global con valor inicial a la que ha accedido previamente sin modificarla, pero cuya página no está actualmente presente.
e) Inmediatamente a continuación de que un proceso modifique una variable global con valor inicial y, luego, cree un hijo (`fork`), el proceso hijo lee esa misma variable.
f) Inmediatamente a continuación de que un proceso lea una variable global con valor inicial y, luego, cree un hijo (`fork`), el hijo escribe en esa misma variable.
g) Inmediatamente a continuación de que un proceso modifique una variable global con valor inicial y, luego, cree un *thread*, el nuevo *thread* escribe en esa misma variable.
h) Analice cómo afectaría a esta estadística el uso del *buffering* de páginas. ¿Habría nuevas situaciones de fallos sin lectura de disco? ¿Dejaría de haber algunas de las situaciones de fallos sin lectura de disco que existen en un sistema que no usa *buffering*?

31. Considérese una TLB convencional, que no incluye información de proceso, tal que cada entrada tiene la información habitual: número de página y de marco, permisos de acceso, y bits de referencia y de modificado. Supóngase que el procesador incluye dos instrucciones para que el sistema operativo pueda manipular la TLB: una para invalidar la entrada vinculada a una página y otra para invalidar completamente el contenido de la TLB. Analice de forma razonada si en cada una de las siguientes operaciones, el sistema operativo haría uso de alguna de estas instrucciones, especificando, en el caso de que se invalide una entrada, a qué página correspondería:

a) Hay un cambio de contexto entre dos *threads* de distinto proceso.
b) Hay un cambio de contexto entre dos *threads* del mismo proceso.
c) Se crea una nueva región en el mapa del proceso.
d) Se elimina una región del mapa del proceso.
e) Cambia el tamaño de una región del mapa del proceso. Analice separadamente el caso del aumento de tamaño y el de la disminución.
f) Se marca como duplicada una región (por ejemplo, en el tratamiento de una región pri-

vada que forma parte de una llamada `fork`).

g) Hay un fallo de página que encuentra un marco libre.

h) Hay un fallo de página que no encuentra marcos libres. El sistema usa el algoritmo de reemplazo del reloj y encuentra que la primera página candidata a ser expulsada tiene tanto el bit de referencia como el de modificado a 1, mientras que la segunda tiene ambos bits a 0.

i) Se produce un fallo debido al COW y el contador de referencias de la página es mayor que 1.

j) Se produce un fallo debido al COW y el contador de Y es igual a 1.

32. Supóngase un programa que proyecta un fichero de forma privada y que, luego, crea un proceso mediante `fork` y este proceso hijo, a su vez, crea un *thread*. Analice, de manera independiente, qué ocurriría en las siguientes situaciones:

a) El *thread* modifica una determinada posición de memoria asociada al fichero proyectado, luego la modifica el proceso hijo y, por último, el padre.

b) El *thread* elimina la proyección del fichero y, acto seguido, intentan acceder a esa región el proceso hijo y el padre.

33. Sea un sistema con memoria virtual, en el que las direcciones lógicas que tienen un 0 en el bit de mayor peso son de usuario y las que tienen un 1 son de sistema. Considérese que el sistema operativo no tiene ningún error de programación y que se produce un fallo de página tal que la dirección que lo provoca no pertenece a ninguna región del proceso. Dependiendo de en qué modo se ejecutaba el proceso cuando ocurrió el fallo y del tipo de la dirección de fallo, se dan los siguientes casos, cada uno de los cuales se deberá analizar para ver si es posible esa situación y, en caso afirmativo, explicar qué tratamiento requiere analizando si influye en el mismo el valor del puntero de pila en el momento del fallo:

a) El proceso se estaba ejecutando en modo usuario y la dirección de fallo comienza con un 0.

b) El proceso se estaba ejecutando en modo usuario y la dirección de fallo comienza con un 1.

c) El proceso se estaba ejecutando en modo sistema y la dirección de fallo comienza con un 0. Distinga entre el caso de que el fallo se produzca durante una llamada al sistema o en una rutina de interrupción.

d) El proceso se estaba ejecutando en modo sistema y la dirección de fallo comienza con un 1. Distinga entre el caso de que el fallo se produzca durante una llamada al sistema o en una rutina de interrupción.

34. ¿Qué acciones sobre la TLB, la memoria cache y la tabla de páginas se realizan durante un cambio de contexto? Suponga distintos tipos de *hardware* de gestión de memoria.

35. En un sistema con preasignación de *swap* se implantan dos cuotas máximas de uso de recursos por parte de un proceso: el tamaño máximo del mapa del proceso y el consumo máximo del espacio de *swap* por el proceso. Analice razonadamente si en las siguientes operaciones es necesario comprobar alguna de estas cuotas:

a) Rutina de tratamiento del fallo de página.

b) Proyección compartida de un fichero.

c) Proyección privada de un fichero.

d) Proyección privada de tipo anónima.

e) Llamada al sistema `fork`.

f) Rutina de tratamiento del COW.

g) ¿Habría que comprobar alguna de estas cuotas en la gestión del *heap*? En caso afirmativo, ¿en qué rutina del sistema operativo?

h) ¿Habría que comprobar alguna de estas cuotas en la gestión de la pila? En caso afirmativo, ¿en qué rutina del sistema operativo?

36. Una técnica perezosa en la gestión de memoria es la paginación por demanda. Suponiendo que existe suficiente memoria física, explique cuándo es beneficiosa, en cuanto a una ejecución más eficiente del proceso si se compara con la solución no perezosa, y cuándo perjudicial y por qué motivo, en caso de que pueda serlo.

37. Responda a la misma cuestión sobre la técnica COW.

38. Durante el arranque, Linux almacena en un marco una página de sólo lectura llena de ceros, que se mantendrá así todo el tiempo, y que se usa para implementar una técnica perezosa en la gestión de páginas de una región anónima. Explique cómo sería esta técnica mostrando en qué situaciones puede ser beneficiosa y en cuáles perjudicial, si es que puede serlo.

5

E/S Y SISTEMA DE FICHEROS

La gestión de la entrada/salida (E/S) es una de las funciones principales del sistema operativo. De hecho, el origen de los sistemas operativos surgió de la necesidad de ocultar la complejidad y heterogeneidad de los diversos dispositivos de E/S, ofreciendo un modo de acceso a los mismos uniforme y de alto nivel. En la actualidad, el sistema de E/S sigue constituyendo una parte fundamental del sistema operativo, siendo el componente que, normalmente, comprende más código, dada la gran diversidad de dispositivos presentes en cualquier computador. En este capítulo se presentan los conceptos básicos de E/S, se describe brevemente el hardware de E/S y su visión lógica desde el punto de vista del sistema operativo, se muestra cómo se organizan los módulos de E/S en el sistema operativo y los servicios de E/S que proporciona éste.

En este capítulo también se presentan los conceptos relacionados con ficheros y directorios. El capítulo tiene tres objetivos básicos: mostrar al lector dichos conceptos desde el punto de vista de usuario, los servicios que da el sistema operativo y los aspectos de diseño de los sistemas de ficheros y del servidor de ficheros. De esta forma, se pueden adaptar los contenidos del tema a distintos niveles de conocimiento.

Desde el punto de vista de los usuarios y las aplicaciones, los ficheros y directorios son los elementos centrales del sistema. Cualquier usuario genera y usa información a través de las aplicaciones que ejecuta en el sistema. En todos los sistemas operativos de propósito general, las aplicaciones y sus datos se almacenan en ficheros no volátiles, lo que permite su posterior reutilización. Por ello, un objetivo fundamental de cualquier libro de sistemas operativos suele ser la dedicada a la gestión de archivos y directorios, tanto en lo que concierne a la visión externa y al uso de los mismos mediante los servicios del sistema, como en lo que concierne a la visión interna del sistema y a los aspectos de diseño de estos elementos.

Para alcanzar este objetivo el planteamiento que se va a realizar en este capítulo es plantear el problema inicialmente los aspectos externos de ficheros y directorios, con las distintas visiones que de estos elementos deben tener los usuarios, para continuar profundizando en los aspectos internos de descripción de ficheros y directorios, las estructuras que los representan y su situación en los dispositivos de almacenamiento usando sistemas de ficheros. Al final del capítulo se mostrarán ejemplos prácticos de programación de sistemas en UNIX y en Windows.

Para desarrollar el tema, el capítulo se estructura en los siguientes grandes apartados:

- *Nombrado de los dispositivos*
- *Manejadores de dispositivos*
- *Servicios de E/S bloqueantes y no bloqueantes*
- *Consideraciones de diseño de la E/S*
- *Concepto de fichero*
- *Directorios*
- *Sistema de ficheros*
- *Servidor de ficheros*
- *Protección*
- *Consideraciones de diseño del servidor de ficheros*
- *Servicios de E/S*
- *Servicios de ficheros y directorios*
- *Servicios de protección y seguridad*

5.1. INTRODUCCIÓN

En el capítulo "1 Conceptos arquitectónicos del computador" se ha visto que el **acceso físico** (o de bajo nivel) a los dispositivos de E/S es muy **complejo**, puesto que depende de detalles físicos de los dispositivos y de su comportamiento temporal. Además, es necesario conocer las direcciones físicas donde se ubican los registros de los controladores.

Por otro lado, los dispositivos **no incluyen mecanismos de protección**, por tanto, todo usuario que accede a nivel físico a un dispositivo lo hace sin ningún tipo de restricción. El controlador del dispositivo ejecuta las órdenes que recibe sin cuestionar si están o no están autorizadas.

Objetivos del sistema de E/S

El sistema de entrada/salida del sistema operativo consiste en un conjunto de capas de *software* que cubren las siguientes funcionalidades:

- Controlar el funcionamiento de los dispositivos de E/S. Para ello, ha de conocer todos los detalles de funcionamiento de cada dispositivo así como los tipos de errores que pueden generar y la forma de tratarlos.
- Presentar a los usuarios una interfaz lógica, sencilla y homogénea, con abstracción de los detalles físicos del acceso a los dispositivos de E/S, facilitando así el manejo de los mismos.
- Proporcionar mecanismos de protección, restringiendo los accesos que pueden realizar los usuarios en base a unos permisos y privilegios.
- Permitir la explotación de la concurrencia que existen entre la E/S y el procesador, optimizando el uso de este último.

Estructura y componentes del sistema de E/S

Como se puede observar en el ejemplo mostrado en la figura 5.1, el sistema de E/S se estructura en un conjunto de capas de *software*. Aunque el diseño concreto depende de cada sistema operativo, nos basaremos en este ejemplo para exponer las ideas fundamentales.

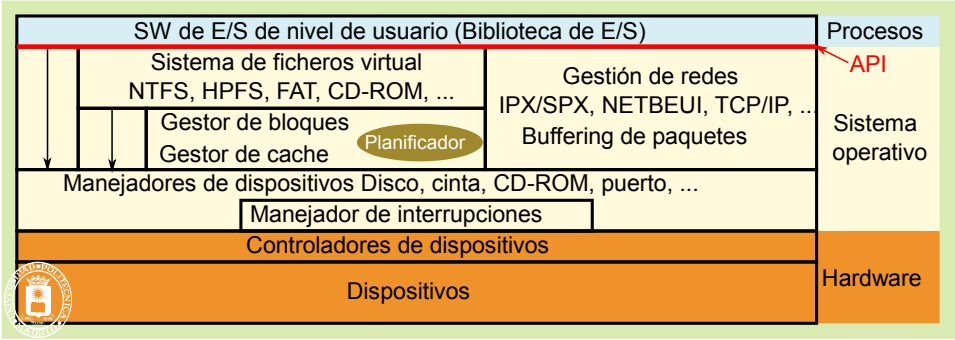


Figura 5.1 *Principales capas de un sistema de E/S.*

API (*Application Programming Interface*). El sistema operativo proporciona un conjunto de servicios que permiten a los procesos acceder a los periféricos de forma simple y sistemática, ocultando sus detalles de bajo nivel. Estos servicios vienen proporcionados fundamentalmente por el sistema de ficheros o por la gestión de redes, pero también hay servicios que permiten acceder directamente a los manejadores de dispositivo.

El **sistema de ficheros virtual** proporciona un conjunto de servicios homogéneos para acceder a todos los sistemas de ficheros que proporciona el sistema operativo (FFS, SV, NTFS, FAT, etc.). Permite acceder a los manejadores de los dispositivos de almacenamiento de forma transparente, incluyendo en muchos casos, como NFS o NTFS, accesos remotos a través de redes. Para los dispositivos de bloques convierte las peticiones en operaciones de bloque que solicita al gestor de bloques. Para dispositivos de caracteres interacciona directamente con los correspondientes manejadores de dispositivo.

El **gestor de redes** proporciona un conjunto de servicios homogéneos para acceder a los sistemas de red que incorpora el sistema operativo, implementando la pila de comunicaciones de protocolos como TCP/IP o Novell. Permite, además, acceder a los manejadores de cada tipo de red particular de forma transparente. El gestor de redes utiliza un *buffer* **de paquetes** que permite almacenar temporalmente tanto los paquetes que se envían y como los que se reciben.

El **gestor de bloques** se emplea para los dispositivos de bloques, encargándose de leer y escribir bloques. Interacciona con la **cache de bloques** para almacenar temporalmente bloques de información de forma que se optimice la E/S. El gestor de bloques incluye una **cola de peticiones** para cada dispositivo de bloques. También incluye las funciones de **planificación**, que determinan qué solicitud de la cola se atiende cuando queda libre el dispositivo afectado.

Cada tipo de dispositivo requiere un **manejador de dispositivo**. El manejador proporciona, a la capa superior, operaciones de alto nivel genéricas, que convierte en secuencias de control específicas para su tipo de dispositivo. Normalmente, su diseño se divide en una parte de alto nivel, que es independiente del tipo de dispositivo y que pro-

porciona una interfaz de alto nivel al resto del *software* de E/S, y una parte de bajo nivel, que recoge la idiosincrasia del dispositivo.

El **manejador de interrupciones** recibe cada interrupción y pone en ejecución el correspondiente código. En general, dicho código se encarga simplemente de comunicar el evento al correspondiente manejador, que es el encargado de tomar las acciones pertinentes. Para algunos dispositivos simples incluye parte del tratamiento. Por ejemplo, para el teclado incorpora un *buffer* para ir almacenando los caracteres recibidos.

Finalmente, destacaremos que la gran mayoría del *software* de E/S es independiente de los dispositivos, puesto que la parte específica de ellos queda limitada a la parte de bajo nivel de los manejadores.

5.2. NOMBRADO DE LOS DISPOSITIVOS

Linux

Internamente en Linux los dispositivos se identifican mediante dos números denominados *major* y *minor*, El *major* identifica el tipo de dispositivo, por ejemplo, en los discos SCSI el *major* vale 8, mientras que el *minor* especifica cada dispositivo concreto, por ejemplo, si se tienen dos discos SCSI, sus *minors* serán 0 y 1 respectivamente.

Adicionalmente, cada dispositivo tiene un **nombre de fichero** ubicado en el directorio `/dev`. Por ejemplo, el disco SCSI con minor 0 tiene el nombre `/dev/sda`. Los prefijos de nombre de fichero de los dispositivos más corrientes son los siguientes:

- hd: discos IDE.
  - hda: maestro primario (*major* = 3 y *minor* = 0).
  - hdb: esclavo primario.
  - hdc: maestro secundario (*major* = 22 y *minor* = 0).
  - hdd: esclavo secundario.
  - Las particiones se nombran añadiendo una cifra, por ejemplo, hda1 es la primera partición del disco hda.
- sd: disco SCSI.
- st: cinta magnética
- tty: terminales
  - ttyS: puerto serie
  - ttyUSB: convertidores serie USB, módems, etc.
- parport o pp: puerto paralelo.
- pt: pseudo-terminales (terminales virtuales).
- /dev/null – acepta y desecha cualquier escritura; no produce salida (siempre devuelve fin de fichero a toda lectura).
- /dev/zero – acepta y desecha cualquier escritura; en lectura produce una serie continua de bytes a cero.
- /dev/full – devuelve un “disco lleno” a cualquier escritura; en lectura produce una serie continua de bytes a cero.
- /dev/random and /dev/urandom – produce una cadena de número pseudo-aleatorios.

Windows

En Windows los dispositivos tienen un nombre interno dentro del “Windows NT's Object Manager”, ejemplos son: \Device\Harddisk0, \Device\CDRom0 y \Device\Serial0.

Sin embargo, los programas no pueden utilizar estos nombres. Por el contrario, deben utilizar los nombres de MS-DOS tales como «A:», «B:», «C:», «F:», «COM1», «COM2», «COM4», «LPT3», etc. Estos nombres son en realidad enlaces simbólicos al nombre interno, nombres que pueden ser cambiados por el usuario.

5.3. MANEJADORES DE DISPOSITIVOS

Cada clase de dispositivo de E/S tiene que incorporar un manejador en el sistema operativo, a través del cual poder utilizado. La tarea de un manejador de dispositivo es aceptar peticiones en formato abstracto y ejecutarlas. La figura 5.2 muestra un diagrama de flujo con las operaciones de un manejador.

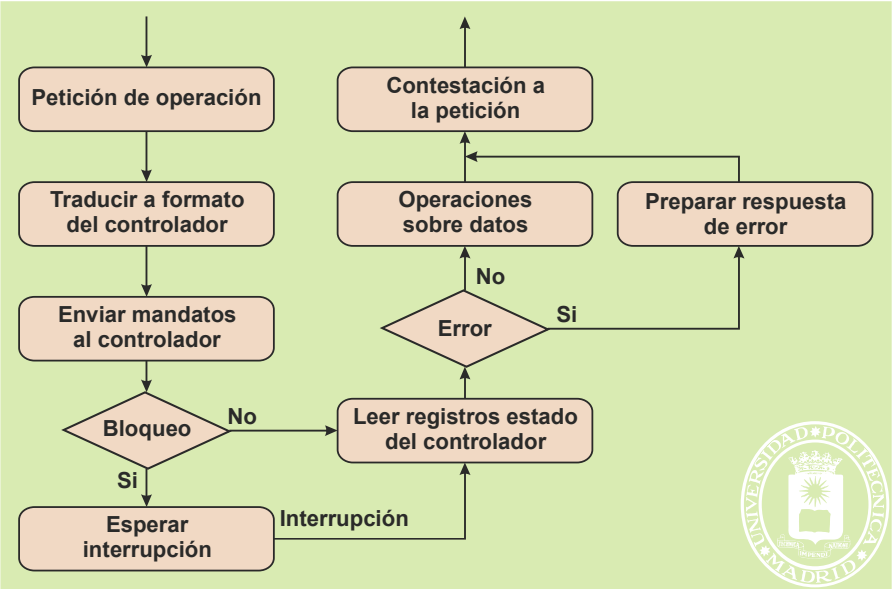


Figura 5.2 *Diagrama de flujo de las operaciones de un manejador de dispositivo.*

El primer paso del manejador es traducir la petición a términos que entienda el controlador. Seguidamente, envía al controlador las órdenes adecuadas en la secuencia correcta. Una vez enviada la petición, el manejador se bloquea, o no se bloquea, dependiendo de la velocidad del dispositivo. Para los lentos, que son la mayoría (e.g. disco, teclado, ratón, etc.), se **bloquea** esperando una interrupción. Se dice que tienen un funcionamiento **asíncrono**. Para los rápidos (e.g. pantalla o disco RAM) espera una respuesta inmediata **sin bloquearse**. Se dice que tienen un funcionamiento **síncrono**.

Después de recibir el fin de operación, el manejador controla la existencia de errores y devuelve al nivel superior el estado de terminación de la operación.

En los sistemas operativos modernos, como Windows, los manejadores se agrupan en clases. Para cada clase existe un manejador genérico que se encarga de las operaciones de E/S para una clase de dispositivos, tales como el CD-ROM, el disco, la cinta o un teclado. Cuando se instala un dispositivo particular, como por ejemplo el disco SEAGATE S300, se crea una instancia del manejador de clase con los parámetros específicos de ese objeto. Todas las funciones comunes al manejador de una clase se llevan a cabo en el manejador genérico y las particulares en el del objeto. De esta forma, se crea un apilamiento de manejadores que refleja muy bien qué operaciones son independientes del dispositivo y cuáles no.

5.4. SERVICIOS DE E/S BLOQUEANTES Y NO BLOQUEANTES

Hemos visto en la sección anterior que la mayoría de los dispositivos de E/S operan de forma asíncrona con el manejador. Sin embargo, la mayoría de los servicios de E/S funcionan con lógica bloqueante, lo que significa que el sistema operativo recibe la petición del proceso y lo bloquea hasta que ésta haya terminado (véase figura 5.3a), momento en que desbloquea al proceso y le suministra el estado del resultado de la operación. El proceso puede acceder al resultado de la operación en cuanto ha sido desbloqueado (para él inmediatamente). Este modelo de programación es claro y sencillo, por lo que las principales llamadas al sistema de E/S, como `read` o `write` en Linux y `ReadFile` y `WriteFile` en Windows, bloquean al proceso y completan la operación antes de devolver el control al proceso.

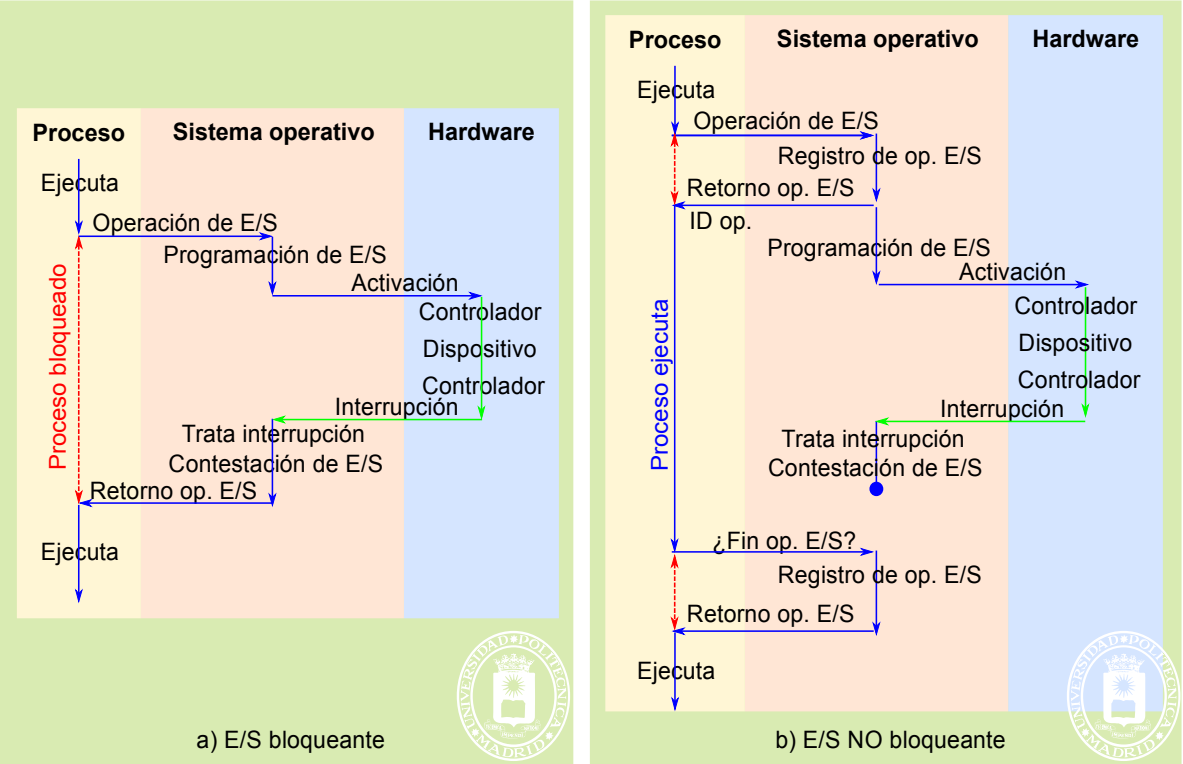


Figura 5.3 *Flujo de las operaciones de E/S a) bloqueantes y b) no bloqueantes.*

Las llamadas de E/S no bloqueantes se comportan de forma muy distinta, reflejando mejor la propia naturaleza del comportamiento de los dispositivos de E/S. Estas llamadas permiten al proceso seguir su ejecución, sin bloquearlo, después de hacer una petición de E/S (véase figura 5.3b). El procesamiento de la llamada de E/S consiste en recoger los parámetros de la misma, asignar un identificador de operación de E/S pendiente de ejecución y devolver a la aplicación este identificador. A continuación, el sistema operativo ejecuta la operación de E/S en concurrencia con la aplicación, que sigue ejecutando su código. Es responsabilidad de la aplicación preguntar por el estado de la operación de E/S, y cancelarla si ya no le interesa o tarda demasiado.

Las llamadas de UNIX `aioread` y `aiowrite` permiten realizar operaciones no bloqueantes. La consulta se realiza con `aiowait` y la cancelación con `aiocancel`.

En Windows se puede conseguir este mismo efecto indicando, cuando se crea el fichero, que se desea E/S no bloqueante (`FILE_FLAG_OVERLAPPED`) y usando las llamadas `ReadFileEx` y `WriteFileEx`.

Este modelo no bloqueante es más complejo, pero se ajusta muy bien al modelo de algunos sistemas que emiten peticiones y reciben la respuesta después de un cierto tiempo. Un programa que esté leyendo datos de varios ficheros, por ejemplo, puede usarlo para hacer lectura adelantada de datos y tener los datos de un fichero listos en memoria en el momento de procesarlos. Un programa que escuche por varios canales de comunicaciones podría usar también este modelo (véase el consejo 8.1).

Consejo 5.1 El modelo de E/S no bloqueante, o asíncrono, es complejo y no apto para programadores o usuarios noveles del sistema operativo. Si lo usa, debe tener estructuras de datos para almacenar los descriptores de las operaciones que devuelve el sistema y procesar todos ellos, con espera o cancelación. Tenga en cuenta que almacenar el estado de estas operaciones en el sistema tiene un coste en recursos y que el espacio es finito. Por ello, todos los sistemas operativos definen un máximo para el número de operaciones de E/S no bloqueantes que pueden estar pendientes de solución. A partir de este límite, las llamadas no bloqueantes devuelven un error.

Es interesante resaltar que, independientemente del formato elegido por el programador, el sistema operativo procesa internamente las llamadas de E/S de forma no bloqueante, o asíncrona, para ejecutar otros procesos durante los tiempos de espera de los periféricos.

5.5. CONSIDERACIONES DE DISEÑO DE LA E/S

Analizaremos en esta sección los siguientes temas:

- El manejador del terminal.
- Almacenamiento secundario.
- Gestión de reloj.
- Ahorro de energía.

5.5.1. El manejador del terminal

El terminal se compone fundamentalmente de dos dispositivos el teclado y la pantalla. En esta sección se presentarán primero los aspectos relacionados con el *software* que maneja la entrada del terminal, es decir, el teclado, para, a continuación, estudiar aquéllos vinculados al *software* que gestiona la salida.

Software de entrada

La lectura del terminal está dirigida por interrupciones, cada tecla pulsada por el usuario genera una interrupción, se trata, por tanto, de un dispositivo con un modo de entrada asíncrono. Esto significa que pueden llegar caracteres aunque no haya ningún proceso esperando por los mismos, es decir, leyendo del teclado. El manejador mantiene un *buffer* de entrada para guardar los caracteres que va tecleando el usuario. Si hay un proceso leyendo del teclado se le pasan los caracteres, pero si no lo hay se mantienen en el *buffer*. Posteriormente, el proceso puede leer dichos caracteres, dando lugar al **tecleado anticipado** (*type ahead*) o borrarlos simplemente. Esta característica permite que el usuario teclee información antes de que el programa la solicite, lo que proporciona una interfaz más amigable.

La lectura del *buffer* puede estar **orientada a líneas** o caracteres. En el primer caso, solamente se suministra la información cuando llega un fin de línea (carácter LF en UNIX o combinación de caracteres CR-LF en Windows).

En la mayoría de los sistemas operativos, el manejador tiene dos modos de funcionamiento, que en UNIX se llaman elaborado y crudo.

En el modo **crudo** el manejador se limita a introducir en el *buffer* de entrada y a enviar a la aplicación las teclas pulsada por el usuario, sin procesar nada. Dado que las teclas corresponden, según el país, con caracteres distintos se produce normalmente una traducción de tecla a carácter.

En el modo **elaborado** el manejador procesa directamente ciertas teclas, tales como:

- **Caracteres de edición**. Teclas que tienen asociadas funciones de edición tales como borrar el último carácter tecleado, borrar la línea en curso o indicar el fin de la entrada de datos.
- **Caracteres para el control de procesos**. Todos los sistemas proporcionan al usuario algún carácter para abortar la ejecución de un proceso o detenerla temporalmente.
- **Caracteres de control de flujo**. El usuario puede desear detener momentáneamente la salida que genera un programa para poder revisarla y, posteriormente, dejar que continúe apareciendo en la pantalla. El manejador gestiona caracteres especiales que permiten realizar estas operaciones.
- **Caracteres de protección**. A veces el usuario quiere introducir como entrada de datos un carácter que está definido como especial. Se necesita un mecanismo para indicar al manejador que no trate dicho carácter, sino que lo pase directamente a la aplicación. Para ello, generalmente, se define un carácter de protección cuya misión es indicar que el carácter que viene a continuación no debe procesarse. Evidentemente, para introducir el propio carácter de protección habrá que teclear otro carácter de protección justo antes.

En la mayoría de los sistemas se ofrece la posibilidad de cambiar qué carácter está asociado a cada una de estas funciones o, incluso, desactivar dichas funciones si se considera oportuno.

Por último, hay que resaltar que, dado que el procesamiento de cada carácter conlleva una cierta complejidad, sobre todo en modo elaborado, habitualmente, éste se realiza dentro de una rutina diferida activada mediante una interrupción *software*.

Software de salida

La salida en un terminal no es algo totalmente independiente de la entrada. Por defecto, el manejador hace **eco** de todos los caracteres que va recibiendo en las sucesivas interrupciones del teclado, enviándolos a la pantalla. Así, la salida que aparece en el terminal es una mezcla de lo que escriben los programas y del eco de los datos introducidos por el usuario. La opción de eco del manejador se puede desactivar, encargándose entonces la aplicación de hacer o no hacer el eco. En la figura 5.4, se puede apreciar esta conexión entre la entrada y la salida. Además del *buffer* de entrada ya presentado previamente, en la figura se puede observar un *buffer* de salida.

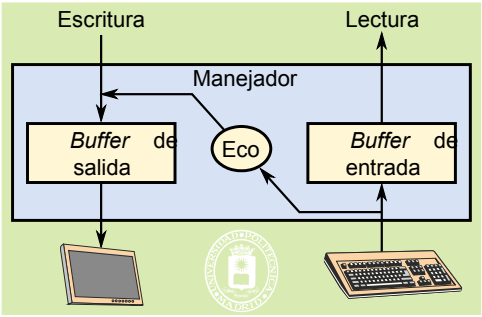


Figura 5.4 *Flujo de datos en el manejador del terminal.*

A diferencia de la entrada, la salida no está orientada a líneas de texto, sino que se escriben directamente los caracteres que solicita el programa, aunque no constituyan una línea (véase la aclaración 5.1).

Aclaración 5.1. Un programador en C que trabaja en un sistema UNIX puede pensar que la salida en el terminal está orientada a líneas, ya que, cuando usa en la sentencia `printf` una cadena de caracteres que no termina en un carácter de nueva línea, ésta no aparece inmediatamente en la pantalla. Sin embargo, este efecto no es debido al manejador, sino a que se usa un *buffer* en la biblioteca estándar de entrada/salida del lenguaje C.

5.5.2. Almacenamiento secundario

Planificación del disco

El disco es el periférico más importante, siendo en muchos casos el cuello de botella en las prestaciones del sistema. Por ello, la planificación de los accesos al disco es de gran importancia, especialmente cuando se trata de **discos magnéticos**, por el elevado tiempo de acceso de los mismos así como porque este tiempo depende fuertemente de la posición actual del brazo y de la posición del o de los sectores a leer.

Los criterios de planificación son básicamente los dos siguientes:

- Optimizar los tiempos de búsqueda. Esto es relevante en los discos magnéticos, existiendo algoritmos específicos para alcanzar esta optimización.
- Dar servicio determinista, es decir, garantizar unos plazos determinados en los accesos al disco. Esto es especialmente importante en aplicaciones de multimedia, para que la imagen o el sonido no tenga saltos, y sistemas de tiempo real.

Los algoritmos principales de planificación, algunos de los cuales ya se han visto en la planificación de procesos, son los siguientes:

- Primero en llegar primero en ejecutar FCFS (*First Come First Served*). Las peticiones se sirven en orden de llegada
- Menor tiempo de búsqueda SSF (*Shortest Seek First*). Relevante para discos magnéticos. Se emplea en discos magnéticos y se selecciona la petición que está más cerca de la posición actual del brazo. Este algoritmo reduce el tiempo total de los accesos, pero presenta el problema de la inanición. Una solicitud relativa a un sector alejado puede no llegar a ser servida nunca si siguen apareciendo solicitudes cercanas a la posición del brazo.
- Política del ascensor o Scan. Relevante para discos magnéticos. En esta política se mueve el brazo de un extremo a otro del disco, sirviendo todas las peticiones que van encontrándose. Seguidamente, se mueve en sentido inverso, sirviendo las peticiones. El tiempo total de búsqueda es mayor que para el algoritmo SSF, pero evita la inanición.
- Política del **ascensor circular** o CScan (*Circular Scan*). Relevante para discos magnéticos. Es una modificación del Scan en el que las peticiones solamente se atienden en un sentido del movimiento del brazo. Si bien se pierde el tiempo de retorno del brazo, es un algoritmo más equitativo que el Scan que da prioridad a las pistas centrales. Es el algoritmo más empleado en la actualidad.
- Para sistemas con plazos temporales (multimedia o tiempo real), en los que hay que garantizar el tiempo de respuesta, se utilizan otros algoritmos tales como:
  - Plazo más temprano EDF (*Earliest Deadline First*). Se selecciona la solicitud más urgente. Para ello, cada petición lleva asociado un plazo límite de ejecución.
  - Scan-EDF. Es un algoritmo EDF en el que cuando los plazos de varias solicitudes se pueden satisfacer se utiliza la técnica Scan entre ellas.

Gestión de errores de disco

Los errores del disco pueden ser transitorios o permanentes.

Errores transitorios

Los errores transitorios se deben a causas tales como: existencia de partículas de polvo en la superficie del disco magnético que interfieren con el cabezal, pequeñas variaciones eléctricas en la transmisión de datos, fallos de calibración de los cabezales, etc.

Estos errores se detectan porque el ECC (*Error Correcting Code*), que incluye cada sector del disco, que se lee no coincide con el que se calcula de los datos del sector leído.

Los errores se resuelven repitiendo la operación de E/S. Si, después de un cierto número de repeticiones, no se resuelve el problema, el manejador concluye que la superficie del disco está dañada y lo comunica al nivel superior de E/S, que lo trata como un error permanente.

Errores permanentes

Los errores permanentes pueden deberse a las siguientes causas:

- Errores de aplicación, por ejemplo, petición para un dispositivo o sector que no existe. Se comunica el error a la aplicación.
- Errores del controlador, por ejemplo, errores al aceptar peticiones o parada del controlador. Se puede tratar de reiniciar el controlador para ver si desaparece el error. Si, al cabo de un cierto número de repeticiones, no se resuelve el problema, se reporta el error.

- Errores del medio de almacenamiento. Se sustituye el bloque por uno de repuesto. Para ello, tanto los discos magnéticos como los de estado sólido, incluyen sectores de repuesto. Evidentemente, la posible información almacenada en el sector dañado se pierde.

Discos RAM

Un disco RAM es una porción de memoria principal que el sistema operativo trata como un dispositivo de bloques (`tmpfs` y `ramfs` en Linux). La porción de memoria se considera dividida en bloques y el sistema operativo ofrece una interfaz de disco similar a la de cualquier disco. El manejador de esos dispositivos incluye llamadas `open`, `read`, `write`, etc., a nivel de bloque y se encarga de traducir los accesos del sistema de ficheros a posiciones de memoria dentro del disco RAM. Las operaciones de transferencia de datos son copias de memoria a memoria. El disco RAM no conlleva ningún *hardware* especial asociado y se implementan de forma muy sencilla. Pero tienen un problema básico: si falla la alimentación se pierden todos los datos almacenados.

Los discos RAM son una forma popular de optimizar el almacenamiento secundario en sistemas operativos convencionales y de proporcionar almacenamiento en sistemas operativos de tiempo real, donde las prestaciones del sistema exigen dispositivos más rápidos que un disco convencional.

Particiones

Una **partición** es una porción de un disco a la que se la dota de una identidad propia y que puede ser manipulada por el sistema operativo como una entidad lógica independiente. Las particiones se definen en la tabla de particiones. A veces volumen y partición se consideran sinónimos, más adelante veremos la diferencia entre ellos.

MBR (*Master Boot Record*)

El diseño clásico de la arquitectura PC incluye el MBR (*Master Boot Record*) almacenado en el sector 0 del disco. El MBR permite definir **4 particiones** de acuerdo a la siguiente estructura:

Dirección	Descripción	Tamaño
000h	Código ejecutable (Arranca el computador)	446 Bytes
1BEh	Definición de la 1ª partición	16 Bytes
1CEh	Definición de la 2ª partición	16 Bytes
1DEh	Definición de la 3ª partición	16 Bytes
1EEh	Definición de la 4ª partición	16 Bytes
1FEh	Boot Record Signature (55h AAh)	2 Bytes

Cada definición de partición tiene los siguientes campos:

Dirección	Descripción	Tamaño
00h	Situación de la partición (00h=inactiva, 80h=activa)	1 Byte
01h	Principio de la partición - Cabeza	1 Byte
02h	Principio de la partición - Cilindro/Sector	1 Palabra
04h	Tipo de partición (Fat-12, Fat-16, Fat-32, XENIX, NTFS, AIX, Linux, MINIX, BSD ...)	1 Byte
05h	Fin de la partición - Cabeza	1 Byte
06h	Fin de la partición - Cilindro/Sector	1 Palabra
08h	Número de sectores desde el MBR y el primer sector de la partición	1 Doble Palabra
0Ch	Número de sectores de la partición	1 Doble Palabra

El tipo de partición 0x05 se utiliza para definir una partición extendida, que puede a su vez ser dividida en varias particiones adicionales, que no pueden contener un sistema operativo.

Cada partición tiene un **tamaño máximo de 2,2 TB**, por lo que tamaño máximo de disco es de 8,8 TB.

GPT (*GUID Partition Table*)

Para superar las limitaciones del diseño MBR se ha desarrollado el estándar GTP como parte del estándar UEFI (*Unified Extensible Firmware Interface*). La figura 5.5 muestra la estructura de la GPT, así como de cada una de las entradas que define una partición.

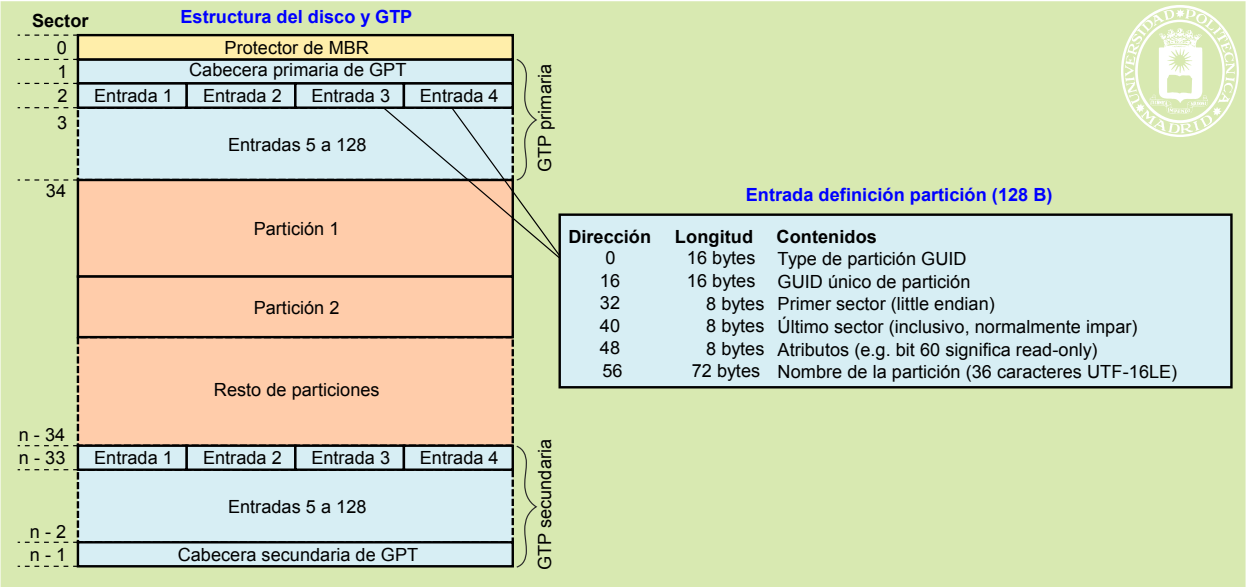


Figura 5.5 *Estructura del disco y de la GPT y de cada entrada de partición.*

Las características de este sistema son las siguientes:

- Se pueden tener hasta 128 particiones.
- La GTP está duplicada para mayor seguridad.
- Permite discos y particiones de hasta 9.4 ZB (9.4×10^{21} bytes)

Volumen

Un volumen es una entidad de almacenamiento definida sobre una o sobre varias particiones. Si las particiones que forman el volumen corresponden a varios discos físicos el volumen es multidisco, como se muestra en la figura 5.6. Como se verá más adelante, sobre un volumen se construye un sistema de ficheros mediante una operación de dar formato.

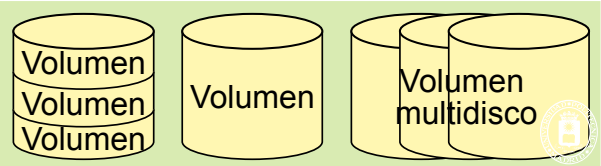


Figura 5.6 Un volumen puede ser una partición de un disco, un disco completo o un conjunto de discos.

Lo más frecuente es que el volumen se defina sobre una sola partición, por lo cual estos términos de consideran muchas veces **sinónimos**. Solamente en el caso de que el volumen sea de tipo multidisco se diferencian los dos términos.

Dispositivos RAID

Los dispositivos **RAID** (*Redundant Array of Independent Disks*) se basan en un conjunto de discos para almacenar la información neta más información de paridad de la información neta. La información de paridad sirve para detectar y corregir la información neta frente al fallo de uno de los discos. Con ello, se consigue un **almacenamiento permanente**, es decir, un almacenamiento que no pierde datos aunque fallen algunos de sus elementos.

La función RAID se puede hacer por *hardware* o por *software*. En el primer caso, el RAID aparece como un solo dispositivo, mientras que, en el segundo, aparecen los N discos individualmente. Es responsabilidad del *software* el presentarlos como un solo dispositivo y el realizar todas las funciones del RAID.

Los RAID permiten seguir operando aunque falle un disco, pero si se produce un segundo fallo la mayoría de ellos pierde la información. Por ello, es necesario sustituir de forma rápida el disco fallido. Esta operación se suele poder hacer **en caliente**, es decir, sin apagar el sistema. Adicionalmente, hay que **reconstruir** la información del disco fallido, lo que representa una carga importante de trabajo para el RAID, por lo que sus prestaciones durante la reconstrucción pueden verse seriamente afectadas.

Existen distintos niveles de RAID que se muestran en la figura 5.7 y que se comparan en la tabla 5.1. Sus características más importantes son las siguientes:

- RAID 0. No es un RAID propiamente dicho, puesto que no incluye redundancia. Reparte la información de los ficheros de forma entrelazada entre los distintos discos, por lo que se alcanzan grandes velocidades de transferencia de información.
- RAID 1. Son discos espejo en los cuales se tiene la información duplicada. Es una solución simple, pero que requiere el doble de discos.

- RAID 2. Distribuye los datos por los discos, repartiéndolos de acuerdo con una unidad de distribución definida por el sistema o la aplicación. El grupo de discos se usa como un disco lógico, en el que se almacenan bloques lógicos distribuidos según la unidad de reparto.
- RAID 3. Reparte los datos a nivel de byte por todos los discos. Se puede añadir bytes con códigos correctores de error. Este dispositivo exige que las cabezas de todos los discos estén sincronizadas, es decir que un único controlador controle sus movimientos.
- RAID 4. Reparto de bloques y código de paridad para cada franja de bloques. La paridad se almacena en un disco fijo. En un grupo de 5 discos, por ejemplo, los 4 primeros serían de datos y el 5º de paridad. Este arreglo tiene el problema de que el disco de paridad se convierte en un cuello de botella.
- RAID 5. Reparto de bloques y paridad por todos los discos de forma cíclica. Tiene la ventaja de la tolerancia a fallos sin los inconvenientes del RAID 4. Existen múltiples dispositivos comerciales de este tipo y son muy populares en aplicaciones que necesitan fiabilidad. Existe el RAID 5E que incluye un disco de reserva, de forma que si un disco falla se utiliza el de reserva, permitiendo que la sustitución se realice más tarde.
- RAID 6. Igual que el RAID 5 pero incluye el doble de redundancia por lo que ofrece una mejor disponibilidad. También existe el RAID 6E.

Tabla 5.1 *Comparación de los distintas soluciones RAID*

Categoría	Nivel	Descripción	Discos necesarios	Disponibilidad de datos	Capacidad de transferencia de grandes operaciones de E/S	Capacidad de transferencia de pequeñas operaciones de E/S
Striping	0	Entrelazado a nivel de bloque sin redundancia.	N	Menor que un solo disco.	Muy alta.	Muy alta tanto para lecturas como escrituras.
Espejo	1	Espejo.	2N	Mayor que RAID 2, 3, 4 o 5, pero menos que RAID 6.	Para lectura más alta que un solo disco, para escritura como un solo disco.	Para lecturas el doble que un solo disco, para escrituras como un solo disco
Acceso paralelo	2	Redundante mediante código Hamming.	N + logN	Mucho mayor que un disco. Comparable a RAID 3, 4 o 5.	La mas alta de todas las alternativas listadas.	Aproximadamente el doble que un solo disco.
	3	Entrelazado a nivel de byte con disco de paridad.	N + 1	Mucho mayor que un disco. Comparable a RAID 2, 4 o 5.	La mas alta de todas las alternativas listadas.	Aproximadamente el doble que un solo disco.
Acceso independiente	4	Entrelazado a nivel de bloque con disco de paridad.	N + 1	Mucho mayor que un disco. Comparable a RAID 2, 3 o 5.	Para lecturas similar al RAID 0, para escrituras significativamente menor que un solo disco.	Para lecturas similar al RAID 0, para escrituras significativamente menor que un solo disco.
	5	Entrelazado a nivel de bloque con paridad distribuida.	N + 1	Mucho mayor que un disco. Comparable a RAID 2, 3 o 4.	Para lecturas similar al RAID 0, para escrituras menor que un solo disco.	Para lecturas similar al RAID 0, para escrituras generalmente menor que un solo disco.
	6	Entrelazado a nivel de bloque con doble paridad distribuida.	N + 2	La mayor de todas las alternativas listadas.	Para lecturas similar al RAID 0, para escrituras menor que RAID 5.	Para lecturas similar al RAID 0, para escrituras significativamente menor que RAID 5.

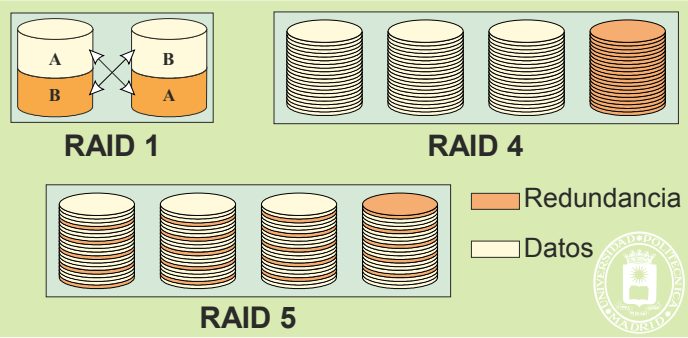


Figura 5.7 *Algunas configuraciones de RAID.*

Tal y como muestra la figura 5.8, se pueden construir RAID utilizando otros RAID como dispositivos. En ese caso, el tipo de RAID viene definido por el tipo del RAID de los dispositivo seguido del tipo del RAID global (a veces estos números se separan por un signo +). En el ejemplo de la figura se utiliza un RAID 0 con dispositivos RAID 5, por lo que su denominación es RAID 50, o bien RAID 5+0.

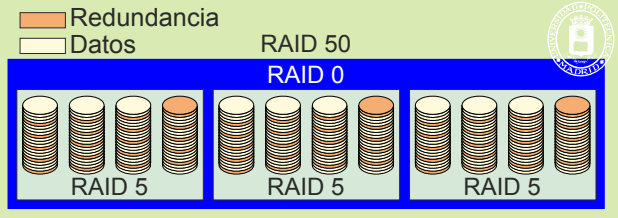


Figura 5.8 *RAID tipo 0 compuesto por tres dispositivos tipo RAID 5, dando lugar a un RAID 50, también denominado RAID 5+0.*

Los conjuntos de almacenamiento RAID se comercializan en unidades enracables en armarios de 12” y tienen alturas de 2U (88,9 mm), 3U (133,35 mm) o 4U (177.8 mm). Sus características más importantes son las siguientes:

- Dependiendo del tamaño la unidad puede albergar entre 12 y 48 discos de 3,5” o 2,5”.
- Suelen incluir varias conexiones de tipo Fiber Channel, iSCSI o AoE, lo que permite su conexión a varios computadores o conmutadores.
- Permiten configurar varios tipos de RAID tales como: 0, 1, 10, 1E, 5, 6, 50, 5EE, 60.
- Permiten la sustitución de discos en caliente (*hot-swappable*).
- La reconstrucción del RAID una vez sustituido el disco defectuoso se hace por *hardware*.

Sistemas de almacenamiento SAN y NAS

Existen dos filosofías en el diseño de los sistemas de almacenamiento, la denominada SAN (*Storage Area Network*) y la denominada NAS (*Network-Attached Storage*). La diferencia entre una SAN y una NAS estriba en el tipo de conexión entre los dispositivos de almacenamiento y los nodos clientes del mismo.

En los sistemas SAN la conexión está orientada a bloques, lo que ocurre con las conexiones de tipo Fiber Channel, iSCSI o AoE. El sistema se comporta como un gran dispositivo de bloques.

Sin embargo, en los sistemas NAS la conexión está orientada a ficheros, por lo que la conexión está basada en un protocolo de sistema de ficheros distribuido, como SMB, NFS o AFS. En este caso, el sistema se comporta como un gran repositorio de ficheros.

Evidentemente, con un SAN se puede construir un NAS. Para ello hay que añadir unos nodos, que ejecuten un servidor de ficheros distribuido, así como una red adicional para conectar los nodos servidores de ficheros con los nodos clientes del NAS, tal y como se muestra en la figura 5.9.

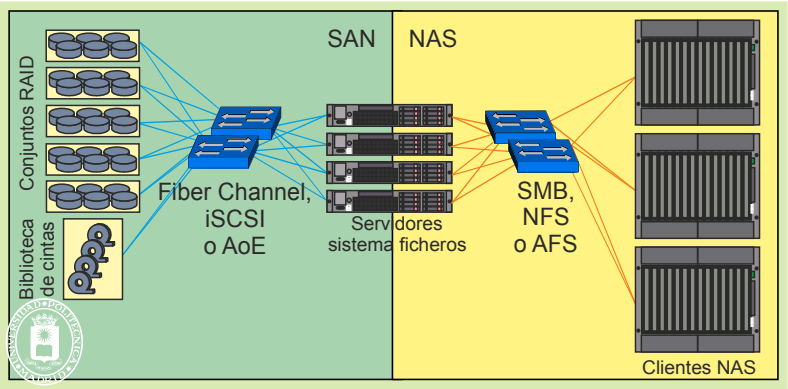


Figura 5.9 *Sistema híbrido SAN-NAS*

5.5.3. Gestión de reloj

Según se ha visto en la sección “ 1.4 El reloj”, los computadores actuales incluyen un RTC (*Real Time clock*) que mantiene la fecha y hora con resolución de 1 segundo y que genera las interrupciones denominadas tic con un periodo de entre 1 a 60 ms. Adicionalmente, los procesadores suelen incluir un TSC (*Time Stamp Counter*) que cuenta los ciclos de reloj de procesador.

Sincronización del reloj

El servicio NTP de Internet está diseñado para sincronizar los relojes de los computadores, de forma que marquen la misma hora. Los sistemas operativos utilizan sistemáticamente este servicio para mantener sincronizado el reloj de los computadores conectados a Internet, por ejemplo, Windows, por defecto, realiza una operación de sincronización cada semana. EL NTP tiene una precisión 20 ms y está basado en un esquema de servidores primarios y secundarios.

La Sección de Hora del Real Instituto y Observatorio de la Armada en San Fernando, que tiene como misión principal el mantenimiento de la unidad básica de Tiempo, difunde, en colaboración con el CSIC, el tiempo mediante el protocolo NTP en la dirección hora.roa.es.

Tratamiento de la interrupción del reloj

Las interrupciones del reloj son de alta prioridad, puesto que si se superponen dos interrupciones se pierde un tic de reloj. Además, hay que minimizar la duración de la rutina de tratamiento, para asegurar que no se pierdan otras interrupciones. Para evitar esta pérdida, normalmente, se aplica la misma técnica que se usa en la mayoría de los manejadores, y que consiste, tal como se analizó en la sección “3.10.2 Detalle del tratamiento de interrupciones”, en dividir las operaciones asociadas a una determinada interrupción en las dos partes siguientes:

- Operaciones no aplazables, que se ejecutan en el ámbito de la rutina de interrupción, manteniendo inhabilitadas las interrupciones de ese nivel y de niveles inferiores.
- Operaciones aplazables, que lleva a cabo una rutina que ejecuta con las interrupciones habilitadas. Esta función es activada por la propia rutina de interrupción de reloj mediante un mecanismo de invocación diferida, que en algunos sistemas operativos se denomina interrupción *software*.

Funciones del manejador del reloj

Se pueden identificar las siguientes operaciones como las funciones principales del *software* de manejo del reloj:

- Mantenimiento de la fecha y de la hora.
- Gestión de temporizadores.
- Soporte para la planificación de procesos.
- Contabilidad y estadísticas.

Hay que resaltar que en algunos sistemas operativos sólo la primera de estas cuatro funciones la realiza directamente la rutina de tratamiento de la interrupción de reloj, delegando las restantes a una rutina diferida, que ejecuta con todas las interrupciones habilitadas.

Mantenimiento de la fecha y de la hora

Dado que ni la resolución de 1 segundo ofrecida por el RTC ni el periodo de milisegundos de los tics ofrecen suficiente precisión para algunas situaciones (téngase en cuenta que un procesador actual ejecuta cientos de millones de instrucciones por segundo), el sistema operativo mantiene una hora con resolución de décimas o centésimas de segundo, para lo cual se basa en el TSC.

El sistema operativo almacena la hora en el sistema de tiempo estándar UTC (Tiempo Universal Coordinado), con independencia de las peculiaridades del país donde reside la máquina. La conversión al horario local no la realiza el sistema operativo, sino las bibliotecas del sistema.

Gestión de temporizadores

El sistema operativo permite crear temporizadores para que los programas establezcan plazos de espera. Para ello, mantiene una o varias listas de temporizadores activos, cada uno con indicación del número de tics hasta su vencimiento y de la función que se invocará cuando éste venza.

La gestión de temporizadores es una operación aplazable que ejecuta en una rutina diferida con todas las interrupciones habilitadas.

Soporte para la planificación de procesos

La mayoría de los algoritmos de planificación de procesos tienen en cuenta de una u otra forma el tiempo y, por tanto, implican la ejecución de ciertas acciones de planificación dentro de la rutina de interrupción.

En el caso de un algoritmo *round-robin*, en cada interrupción de reloj se le descuenta el tiempo correspondiente a la porción de tiempo asignada al proceso. Cuando se alcanza el cero, se activa el planificador de procesos para seleccionar otro proceso.

Otros algoritmos requieren recalcular cada cierto tiempo la prioridad de los procesos, teniendo en cuenta el uso del procesador en el último intervalo. Nuevamente, estas acciones estarán asociadas con la interrupción tic de reloj.

Contabilidad y estadísticas

Puesto que la rutina de interrupción se ejecuta periódicamente, desde ella se puede realizar un muestreo de diversos aspectos del estado del sistema, tales como:

- Contabilidad del uso del procesador por parte de cada proceso.
- Obtención de perfiles de ejecución.

5.5.4. Ahorro de energía

Teniendo en cuenta que el consumo eléctrico de los sistemas informáticos se estima que es del orden del 10% del consumo total de energía eléctrica a nivel mundial, el ahorro de energía es un tema de alta relevancia y, no solamente, para los sistemas alimentados por baterías.

ACPI (*Advanced Configuration and Power Interface*) es una especificación abierta actual de control de alimentación, que establece los mecanismos por los cuales el sistema operativo gestiona la energía, no sólo de los portátiles, sino también de equipos de sobremesa y servidores. ACPI es la evolución de APM (*Advanced Power Management*) presentada inicialmente en 1996. Básicamente, permiten acceder a los datos de información de las baterías, controlar la temperatura del procesador aumentando o reduciendo su velocidad, apagar la pantalla, apagar los discos duros y suspender el sistema. Antes de suspender el sistema es necesario volcar la información de estado del mismo al disco duro para poder rearrancar en el punto exacto en que se suspendió la actividad del mismo.

5.6. CONCEPTO DE FICHERO

Un **fichero** es una unidad de almacenamiento lógico no volátil que agrupa un conjunto de informaciones relacionada entre sí bajo un mismo nombre. El **servidor de ficheros** es la parte del sistema operativo que gestiona estas unidades de almacenamiento lógico, ocultando al usuario los detalles del sistema físico de almacenamiento secundario donde se albergan.

5.6.1. Visión lógica del fichero

Tal y como se muestra en la figura 5.10, el sistema operativo ofrece a los usuarios una **visión lógica** del fichero formada por una **cadena ordenada de bytes** que tiene asociado un **puntero**. Las operaciones de escritura y lectura se realizan a partir de dicho puntero, que queda incrementado en el número de bytes de la operación. De esta forma, lecturas o escrituras sucesivas afectan a zonas consecutivas del fichero. Observe que el puntero se utiliza para indicar la posición sobre la que se lee o escribe, por lo tanto es una información que solamente existe en memoria, no existe en el disco.

Figura 5.10 *Visión lógica de un fichero.*

La ventaja de la visión lógica como cadena de bytes tiene la ventaja de ser muy simple y permite a las aplicaciones acomodar cualquier estructura interna de fichero que se desee, entre las que se pueden destacar la estructura en registros de tamaño fijo o variable, o la estructura en árbol, como se muestra en la figura 5.11.

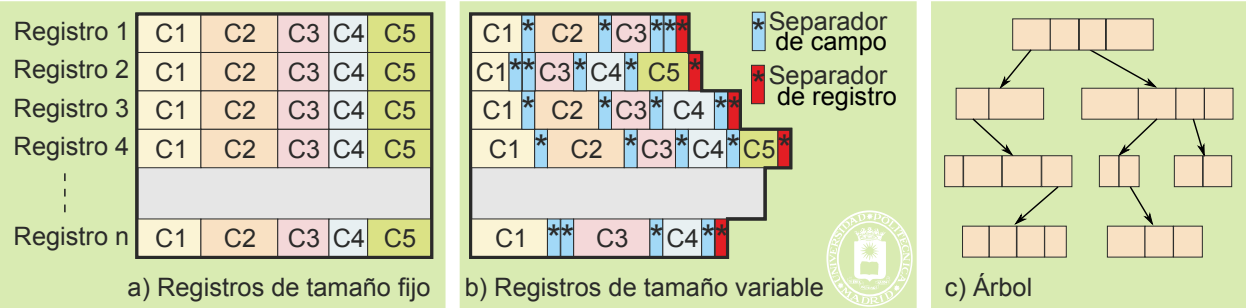


Figura 5.11 *Distintas estructuras internas de los ficheros*

Otras estructuras de fichero

Además de la visión lógica de cadena de bytes el sistema operativo soporta otras estructuras de fichero. En concreto podemos destacar las siguientes:

- Los ficheros directorios tienen una estructura de registros, que en las versiones más simples son registros de tamaño fijo, pero que suele ser de registros de tamaño variable, para adaptarse al diverso tamaño que tienen los nombres de los ficheros.
- Los ficheros ejecutables tienen una estructura que interpreta directamente el sistema operativo y que se muestra en la figura 4.27, página 166.

Ficheros indexados

Un problema de los ficheros de registros es la búsqueda de un determinado registro, operación muy común en la utilización de estos ficheros. Para una búsqueda rápida se deben mantener los registros ordenados. Esta solución presenta dos graves inconvenientes. Por un lado, la inserción y eliminación de registros es muy costosa, puesto que requiere mover un gran número de registros. Por otro lado, sólo se pueden ordenar los registros por un campo, lo que da lugar a un único criterio de búsqueda optimizado.

Para resolver estos problemas se pueden utilizar ficheros indexados que son ficheros de registros de tamaño variable a los que se añade uno o varios índices para hacer las búsquedas, como se muestra en la figura 5.12.

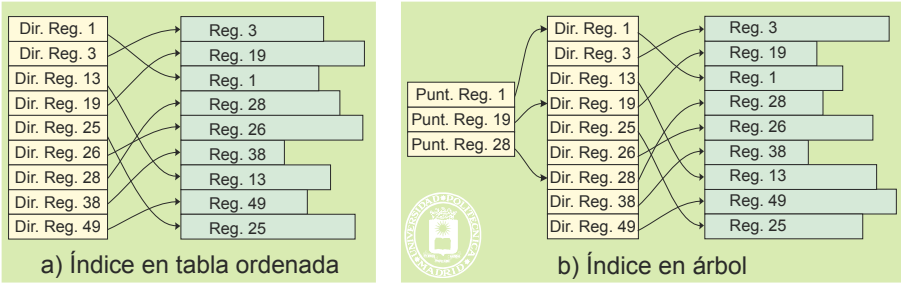


Figura 5.12 *Ficheros indexados. El índice se puede construir como una tabla ordenada o bien como un árbol, lo que acelera las búsquedas, inserciones y eliminaciones.*

Los sistemas operativos actuales no ofrecen la funcionalidad de ficheros indexados. Pero esta funcionalidad se puede añadir mediante una capa *software* adicional, montada sobre el gestor de ficheros del sistema operativo. Existe un estándar de X-OPEN denominado ISAM (Indexed Sequential Access Method) muy frecuentemente usado por estas capas de *software* adicional.

5.6.2. Unidades de información del disco

En el disco se definen las tres unidades de información siguientes:

- **Sector**: Unidad mínima de transferencia que puede manejar el controlador del disco tiene un tamaño de 2^m bytes, siendo normalmente M = 9, por lo que el sector es de 512 B. Esta unidad está definida por el *hardware*. Una operación de lectura o escritura afecta, como mínimo, a todo un sector. Por tanto, si, por ejemplo, solamente se quieren escribir 4B, hay que leer a memoria primero el sector afectado, modificar en memoria los 4 B mencionados y, finalmente, escribir en disco el sector.
- **Bloque**: Es un conjunto de sectores de disco y es la unidad de transferencia mínima que usa el sistema de ficheros al leer o escribir en el disco. Un bloque tiene un tamaño de 2^n sectores. En los sistemas con memoria virtual el bloque suele tener el tamaño de la página, por lo que viene determinado por la unidad de gestión de memoria o MMU. Los bloques se puede direccionar de manera independiente
- **Agrupación**: Es un conjunto de bloques que tiene un tamaño de 2^p bloques. La agrupación tiene un identificador o número de agrupación único y se utiliza como una unidad lógica de gestión de almacenamiento. Al fichero se le asignan agrupaciones, pero se accede siempre en bloques.
  La agrupación suele ser **definible por el administrador** cuando da formato a un volumen. En algunos sistemas Bloque = Agrupación.

No siempre se utilizan los términos bloque y agrupación con el sentido que hemos indicado. Por ejemplo, en el entorno UNIX se denomina bloque a lo que aquí llamamos agrupación. Por tanto, hay que tener cuidado a la hora de leer documentación relativa a UNIX.

El fichero está almacenado en disco, ocupando determinadas agrupaciones. La **estructura física** de un fichero viene dado por el conjunto ordenado de las agrupaciones en las que está almacenado, como se muestra en la figura 5.13. Para poder acceder a un fichero es, por tanto, imprescindible conocer los números de las agrupaciones que lo forman así como el orden en que lo forman, denominándose **mapa del fichero** a esta información (en la figura el mapa es: 7, 24, 72 y 32).

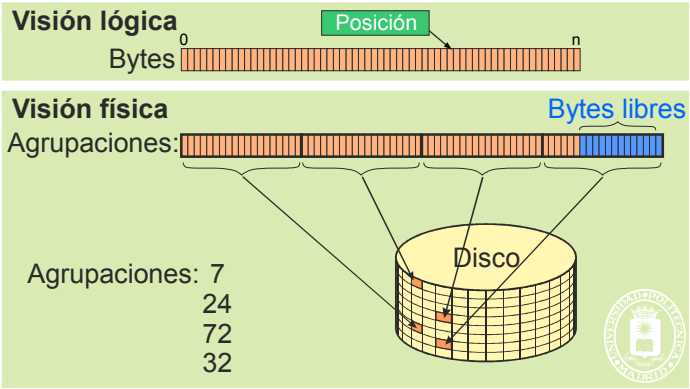


Figura 5.13 *Visión física y lógica de un fichero.*

Dado que los ficheros tienen el tamaño que corresponde a la información que el usuario ha almacenado, la última agrupación tiene, generalmente, espacio sin ocupar. En una primera aproximación, si los tamaños de los ficheros siguiesen una distribución uniforme, se tendría que, por término medio, queda libre media agrupación por fichero. Por tanto, si se define una agrupación muy grande se pierde bastante espacio de almacenamiento por lo que se denomina **fragmentación interna**.

5.6.3. Otros tipos de ficheros

Además de los ficheros **regulares**, mantenidos en almacenamiento secundario, el servidor de ficheros maneja los denominado **ficheros especiales**, que permiten modelar cualquier dispositivo de E/S como un fichero más del sistema. Los ficheros especiales pueden serlo de caracteres (para modelar terminales, impresoras, etcétera) o de bloques (para modelar discos y cintas).

Adicionalmente, el servidor de ficheros incluye los **mecanismos de comunicación** tales como pipes, FIFOS o *sockets*, ofreciendo a los usuarios una interfaz similar a la de los ficheros.

5.6.4. Metainformación del fichero

Para poder gestionar adecuadamente los ficheros, el servidor de ficheros utiliza un conjunto de informaciones, asociadas a cada fichero, que constituyen la metainformación del fichero. Elementos de esta metainformación son los siguientes:

- **Nombre(s)**: Es el nombre textual dado por su creador. Un fichero puede tener más de un nombre, pero un mismo nombre no puede referirse a más de un fichero, puesto que el servidor de ficheros no sabría determinar qué fichero es el realmente referenciado. En la sección de directorio se analizará la estructura jerárquica que tienen los nombres de ficheros. Como se pueden añadir y eliminar nombres de un fichero, es necesario saber el **número de nombres** que tiene un fichero en cada momento, puesto que solamente se puede eliminar el fichero cuando su número de nombres llegue cero. En ese momento, los usuarios ya no pueden referenciar el fichero, por lo que se pueden marcar como libres sus recursos (agrupaciones y nodo-i en UNIX o registro MFT en Windows).
- **Identificador único**: este identificador es fijado por el servidor de ficheros y suele ser un número. Habitualmente es desconocido por los usuarios, que utilizan el nombre textual para referenciar un fichero. Un fichero tiene un único identificador y, por supuesto, el identificador es distinto para cada fichero (relación biunívoca identificador-fichero).
- **Tipo de fichero**: permite diferenciar entre ficheros directorio, ficheros regulares, ficheros especiales o mecanismos de comunicación.
- **Mapa del fichero**: determina la estructura física del fichero, especificando las agrupaciones asignadas al fichero.
- **Dueño**: el fichero tiene un dueño que se identifica por su UID y GID. El dueño es, en principio, el usuario que crea el fichero, pero se puede cambiar.
- **Protección**: información de control de acceso que define quién puede hacer qué sobre el fichero.
- **Tamaño del fichero**: expresa el número de bytes que ocupa el fichero. Dado que la última agrupación no suele estar llena, el tamaño del fichero es menor que el espacio de disco ocupado.
- **Marcas de tiempo**: tiempo de creación, del último acceso, de la última actualización, etc. Esta información es muy útil para gestionar, monitorizar y proteger los sistemas de ficheros.

Esta metainformación se almacena en los ficheros directorio, que analizaremos más adelante, y en una estructura de información asociada a cada fichero, que denominaremos de forma genérica **descriptor físico de fichero o DFF**. La figura 5.14 muestra tres formas de describir un fichero: nodo-i de UNIX, registro de MFT en Windows y entrada de directorio de MS-DOS.

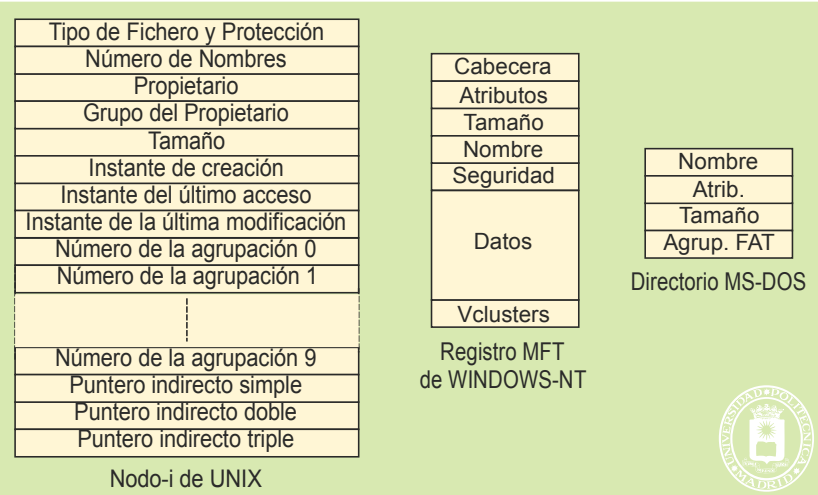


Figura 5.14 *Distintas formas de almacenar la información asociada al fichero.*

El **nodo-i** de UNIX contiene información acerca del propietario del fichero, de su grupo, del modo de protección aplicable al fichero, del número de enlaces al fichero, de valores de fechas de creación y actualización, el tamaño del fichero y el tipo del mismo. Además, incluye un mapa del fichero mediante los números de las agrupaciones que contienen al fichero. A través del mapa de agrupaciones del nodo-i se puede acceder a cualquiera de sus agrupaciones con un número muy pequeño de accesos a disco.

Cuando se abre un fichero, su nodo-i se trae a memoria. En este proceso, se incrementa la información del nodo-i almacenada en el disco con datos referentes al uso dinámico del mismo, tales como el dispositivo en que está almacenado y el número de veces que el fichero ha sido abierto por los procesos que lo están usando.

El **registro de MFT** de Windows describe el fichero mediante los siguientes atributos: cabecera, información estándar, descriptor de seguridad, nombre de fichero y datos (véase figura 5.14). A diferencia del nodo-i de UNIX, un registro de Windows permite almacenar hasta 1,5 KiB de datos del fichero en el propio registro, de forma que cualquier fichero menor de ese tamaño debería caber en el registro. Si el fichero es mayor, dentro del registro se almacena información del mapa físico del fichero incluyendo punteros a grupos de bloques de datos (*Vclusters*), cada uno de los cuales incluye a su vez datos y punteros a los siguientes grupos de bloques. Cuando se abre el fichero, se trae el registro a memoria. Si es pequeño, ya se tienen los datos del fichero. Si es grande hay que acceder al disco para traer bloques sucesivos. Es interesante resaltar que en este sistema todos los accesos a disco proporcionan datos del fichero, cosa que no pasa en UNIX, donde algunos accesos son sólo para leer metainformación.

En el caso de **MS-DOS**, la representación del fichero es bastante más sencilla, debido principalmente a que es un sistema operativo monoproceso y monousuario. En este caso, la información de protección no existe y se limita a unos atributos mínimos que permiten ocultar el fichero o ponerlo como de sólo lectura. El nombre se incluye dentro de la descripción, así como los atributos básicos y el tamaño del fichero en bytes. Además, se especifica la posición del inicio del fichero en la tabla FAT (*file allocation table*), donde se almacena el mapa físico del fichero.

Extensiones al nombre del fichero

Muchos sistemas operativos permiten añadir una o más **extensiones** al nombre de un fichero. Dichas extensiones se suelen separar del nombre con un punto (e.g. `txt`, `pdf`, `gif`, `exe`, `etc.`) y sirven para indicar al sistema operativo, a las aplicaciones, o a los usuarios, características del contenido del fichero.

Habitualmente, las extensiones son significativas sólo para las aplicaciones de usuario. **UNIX no reconoce las extensiones**. Únicamente distingue algunos formatos, como los ficheros ejecutables, porque en el propio fichero existe una cabecera donde se indica el tipo del mismo mediante un número, al que se denomina **número mágico**. Sin embargo, un compilador de lenguaje C puede necesitar nombres de ficheros con la extensión `.c` y la aplicación `compress` puede necesitar nombres con la extensión `.Z`. **Windows tampoco es sensible a las extensiones** de ficheros, pero sobre él existen aplicaciones del sistema (como el explorador o el escritorio) que permiten asociar dichas extensiones con la ejecución de aplicaciones. De esta forma, cuando se activa un fichero con el ratón, se lanza automáticamente la aplicación que permite trabajar con ese tipo de ficheros.

Tamaño máximo de un fichero

El tamaño máximo que puede tener un fichero depende de las limitaciones de los recursos físicos disponible y de la capacidad de direccionamiento de la metainformación.

Las **agrupaciones disponibles** para un fichero pueden ser las agrupaciones libres del sistema de ficheros (todas si solamente se establece un fichero, menos las dedicadas a directorio). Sin embargo, el administrador puede establecer cuotas para los usuarios, de forma que el máximo fichero viene limitado por el tamaño de la cuota que tenga el usuario que crea el fichero.

La **metainformación** también limita el máximo tamaño de los ficheros. Por ejemplo:

- El campo que almacena el tamaño real del fichero limita su tamaño máximo. Si se reservan 4 B para dicho campo el tamaño máximo será de 2^{32} B = 4 GiB.
- En los sistemas FAT12 originales se utilizaban 12 bits para identificar una agrupación, por lo que el máximo número de agrupaciones era de 2^{12} = 4.096 agrupaciones. Con una agrupación de, por ejemplo, 4 KiB

se obtiene un tamaño máximo de volumen (y, por tanto, de fichero) de 16 MiB. Por ello, se ha pasado de la FAT12 a la FAT16 y a la FAT32, actualmente en uso y que permite hasta 2^{28} agrupaciones.

Algunos valores máximos típicos de ficheros y volúmenes son los siguientes:

Sistema de ficheros	Máximo tamaño del nombre del fichero	Máximo tamaño de fichero	Máximo tamaño de volumen
FAT32	255 caracteres UTF-16	4 GiB	2 TiB
NTFS (Windows)	255 caracteres UTF-16	16 TiB	256 TiB
ext2 (UNIX)	255 bytes	2 TiB	32 TiB
ext3 (UNIX)	255 bytes	2 TiB	32 TiB
ext4 (UNIX)	255 bytes	16 TiB	1 EiB

5.7. DIRECTORIOS

Un directorio es un objeto que relaciona de forma unívoca un **nombre de fichero** (dado por el usuario) con su **descriptor interno** DFF. Es, por tanto, necesario garantizar que no se repita el mismo nombre en el directorio, puesto que no quedaría identificado de forma unívoca un fichero.

El directorio es una **unidad de organización** que proporciona el SO. Por lo tanto, son datos con un formato que el propio SO utiliza para localizar ficheros.

Aunque en algunos sistemas de ficheros se han utilizado directorios planos de un solo nivel, la solución que se utiliza de forma casi exclusiva es un esquema jerárquico de nombrado en forma de árbol de nombres con múltiples niveles como muestra en la figura 5.15. Las **hojas** del árbol identifican **ficheros** de usuario, mientras que el resto de los **nodos** identifican **directorios**. El nodo raíz identifica al **directorio raíz**. Todos los directorios tienen un padre (por lo que a veces se denominan subdirectorios) menos el directorio raíz.

En un directorio jerárquico se diferencia entre:

- **Nombre local**: Es el nombre que tiene el fichero en su directorio. Por ejemplo el fichero Yuit de la figura 5.15.
- **Nombre absoluto**: Está compuesto por los nombres de todos los directorios desde el raíz que hasta llegar al directorio donde está incluido el fichero, más su nombre local. Por ejemplo el fichero Raíz-Prog-Voit-Buit-Yuit de la figura 5.15. Ejemplos de nombres absolutos son:
  - Ejemplo UNIX: /usr/include/stdio.h
  - Ejemplo Windows: C:\DocenciaSO\sos2\transparencias\pipes.ppt

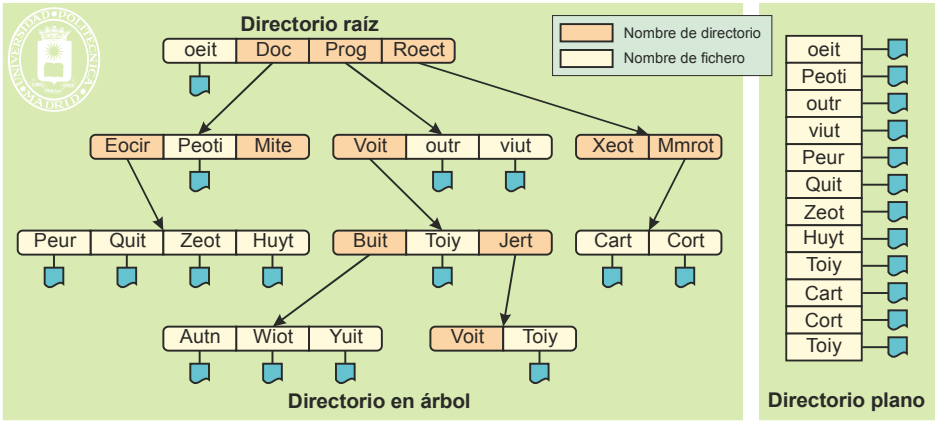


Figura 5.15 *Direcotrio en árbol y plano.*

Se emplea el término de **ruta** para indicar el conjunto de directorios, desde el raíz, que hay que atravesar hasta que se llega a un fichero.

La organización jerárquica de un directorio presenta las siguientes ventajas:

- **Simplifica** el nombrado de ficheros, puesto que, para garantizar que los nombres sean únicos, basta con garantizar que no se repiten nombres locales dentro de un directorio. En efecto, con esta condición, aunque se dé el mismo nombre local a un fichero, su nombre absoluto se diferenciará por tener distinta ruta. Así, en la figura 5.15 hay dos fichero Toiy, pero uno es Raíz-Prog-Voit-Jert-Toiy, mientras que el otro es Raíz-Prog-Voit-Toiy.
- Permite **agrupar** ficheros de forma lógica (mismo usuario, misma aplicación)
- Proporciona una **gestión distribuida** de los nombres. En efecto un usuario puede dar los nombres que quiera dentro de su directorio *home* sin interferir con los nombres que den otros usuarios y sin tener que comprobar si sus nombres están permitidos, puesto que se diferenciarán justamente en su ruta por el nombre de ese directorio *home*.

5.7.1. Directorio de trabajo o actual

El SO mantiene en el BCP el nombre del directorio actual o de trabajo y, además, mantiene en memoria el DFF del directorio de trabajo. De esta forma se consiguen las dos ventajas siguientes:

- El usuario puede utilizar **nombres relativos**. Es decir nombres que empiezan a partir del directorio de trabajo. Estos nombres son más cortos que los absolutos, al no tener que especificar la ruta desde el directorio raíz, por lo que son mucho más cómodos.
- Es más eficiente puesto que la operación de apertura de un fichero no requiere recorrer la ruta desde el raíz al de trabajo, lo que permite ahorrar accesos a disco.

5.7.2. Nombrado de ficheros y directorios

Existen distintas formas de establecer los nombres de los ficheros y directorios, en concreto nos centraremos en los sistemas operativos UNIX y Windows.

Nombres de fichero y directorio en UNIX

En UNIX se establece un único árbol en el que se incluyen los nombres de todos los dispositivos de almacenamiento que utilice el sistema. Esto se consigue mediante la operación de montado de un dispositivo en un punto del árbol de nombres, operación que se analizará más adelante. Las características más destacables son las siguientes

- El nombre absoluto empieza por `/`, por ejemplo: `/usr/include/stdio.h`
- El nombre relativo al directorio actual o de trabajo no empieza por `/`. Por ejemplo: `stdio.h` asumiendo que `/usr/include` es el directorio actual.
- Todo directorio incluye las dos entradas siguientes: «.» (propio directorio) y «..» (directorio padre). Para el directorio raíz, que no tiene padre, se pone «..» = «..». Estos dos nombres «.» y «..» pueden utilizarse para formar rutas de acceso. Por ejemplo, si `/usr/include` es el directorio actual, que contiene el fichero `stdio.h`, se puede llegar a éste con los siguientes nombres:

  stdio.h
  ../include/stdio.h
  ./../include/stdio.h
  /usr/./include/../include/stdio.h

Nombres de fichero y directorio en Windows

En Windows se utiliza un árbol por dispositivo. Sin embargo, el sistema NTFS permite la operación de montado, por lo que se puede establecer un árbol único si se desea. Las características más destacables son las siguientes.

- En lugar de utilizar `/`, se utiliza \
- Todo directorio incluye las dos entradas siguientes: «.» (propio directorio) y «..» (directorio padre), que se pueden utilizar de igual modo que en UNIX.

5.7.3. Implementación de los directorios

La función básica de un directorio es relacionar un nombre de fichero con un DFF, por tanto la implementación de los directorios se realiza estableciendo una tabla nombre/IDFF por cada nodo del árbol de nombres que no sea hoja, tal como muestra la figura 5.16.

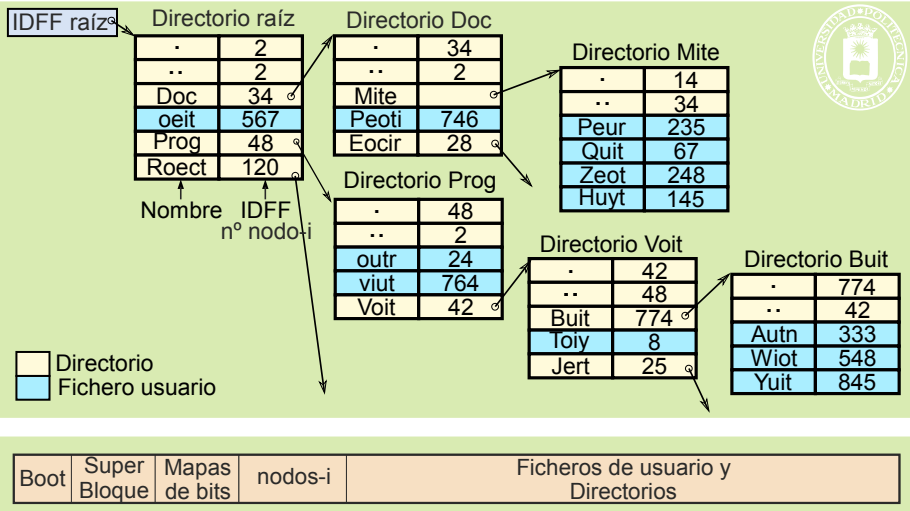


Figura 5.16 *El árbol de nombres de un sistema de ficheros se establece como un árbol de tablas directorio.*

Para poder utilizar un fichero hay que conocer su IDFF (n.º de nodo-i en UNIX), lo que obliga a recorrer el árbol de directorios (desde el raíz o directorio de trabajo si el nombre es relativo) comprobando los permisos en cada paso.

5.7.4. Enlaces

Un enlace es un nuevo nombre que se da a un fichero o directorio existente. El nuevo nombre no implica cambios en el fichero, que conserva sus atributos sin modificar, tales como: permisos, dueño, instantes relevantes, etc.

Existen dos tipos de enlaces, los enlaces físicos y los simbólicos, que analizaremos seguidamente.

Enlaces físicos

Los enlaces físicos, llamados *hard links* en Windows, se caracterizan por tener las siguientes propiedades:

- Solamente se pueden establecer dentro de un sistema de ficheros, por tanto, en UNIX no se puede establecer a un fichero perteneciente a otro sistema de ficheros montado, ni en Windows se puede hacer entre unidades.
- Dependiendo del sistema de ficheros se pueden o no enlazar directorios. En algunos casos se permite enlazar directorios, pero comprobando que el nombre enlazado no esté más arriba en la ruta del nuevo nombre. De esta forma se evita que se produzcan ciclos en el árbol de nombres, lo cual tiene el grave problema de hacer el recorrido del árbol de directorios infinito, repitiendo el ciclo.
- Es necesario llevar la cuenta de nombres que tiene el fichero o directorio para poder eliminarlo cuando llegue a 0 dicha cuenta.
- No se guarda constancia de cuál es el nombre original. Aunque se elimine el nombre original del fichero o directorio, éste seguirá existiendo con el nuevo nombre. Simplemente se decrementa la cuenta de nombres.
- Requiere que se tengan permiso de acceso al fichero existente y que se tenga permiso de escritura en el directorio en el que se incluya el nuevo nombre.
- Al abrir el fichero con el nuevo nombre se comprueban los permisos del nuevo nombre. En el ejemplo de la figura 5.17 se comprobarán los permisos de la ruta: `/user/pedro/dat2.txt`. Por tanto, aunque se cambien los permisos de la ruta original `/user/luis/dat.txt` se mantienen los permisos de la nueva ruta.
- Es de destacar que los nombres «.» y «..» son enlaces físicos, por lo que los enlaces de un directorio son los siguientes: Su nombre en el directorio padre, «..» en su propio directorio y «.» en cada directorio hijo. Por ejemplo, en la figura 5.16, el directorio `Prog` tiene 3 enlaces.

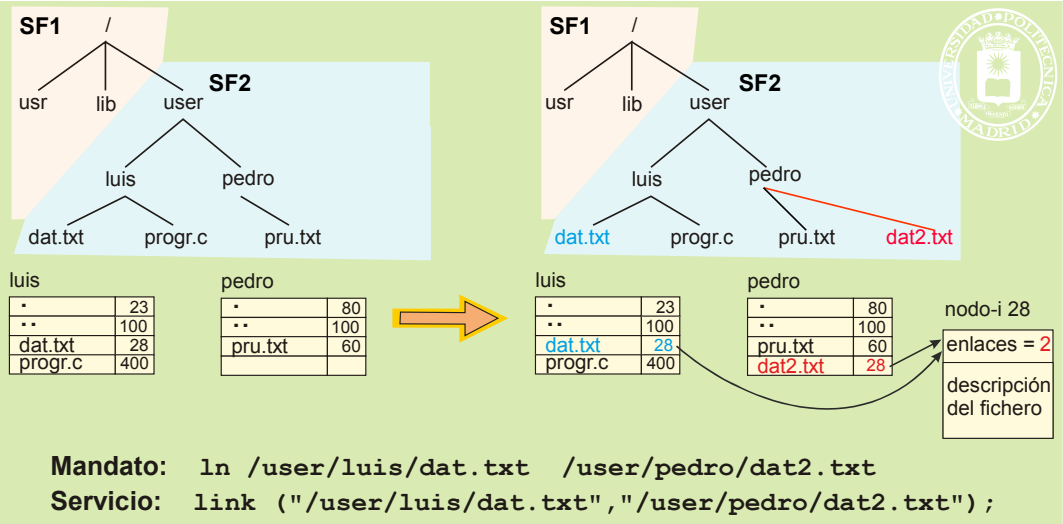


Figura 5.17 *Implementación de un enlace físico en UNIX. Solamente puede hacerse sobre un elemento del mismo sistema de fichero, el SF2 en la figura.*

La implementación de los enlaces físicos en sistemas UNIX se muestra en la figura 5.17. Se puede observar que lo que se hace es utilizar el mismo número de nodo-i para el nuevo nombre e incrementar el número de enlaces del fichero. Dado que los números de nodo-i se repiten en cada sistema de fichero (por ejemplo, el número de nodo-i del directorio raíz suele ser el 2) es evidente que no se puede aplicar esta técnica entre sistemas de ficheros diferentes.

En laces simbólicos

Los enlaces simbólicos, tanto en UNIX como en Windows tienen las siguientes características:

- Se pueden enlazar tanto ficheros como directorios.

- Los ficheros o directorios pueden pertenecer a distintos sistemas de ficheros.
- Requiere que se tengan permisos de acceso al fichero existente y que se tengan permisos de escritura en el directorio en el que se incluya el nuevo nombre.
- Requiere que se tengan permisos de acceso al fichero existente y que se tengan permisos de escritura en el directorio en el que se incluya el nuevo nombre.
- Al abrir el fichero con nuevo nombre se comprueban, primero, los permisos del nuevo nombre hasta el subdirectorio donde se encuentre. A continuación, se comprueban los permisos de la nueva ruta. En el ejemplo de la figura 5.18 se comprobarán primero los permisos de la ruta: `/user/pedro`, siguiendo con los permisos de la ruta: `/user/luis/dat.txt`. Por tanto, si se modifican los permisos de esa ruta, quedarán modificados los permisos de acceso del nuevo nombre.
- Si se elimina un enlace físico del fichero y su contador de enlaces llega a 0, el fichero se elimina. Si el fichero tuviese un enlace simbólico, éste permanece, pero ocurrirá un error de fichero no existente si tratar de abrirlo.

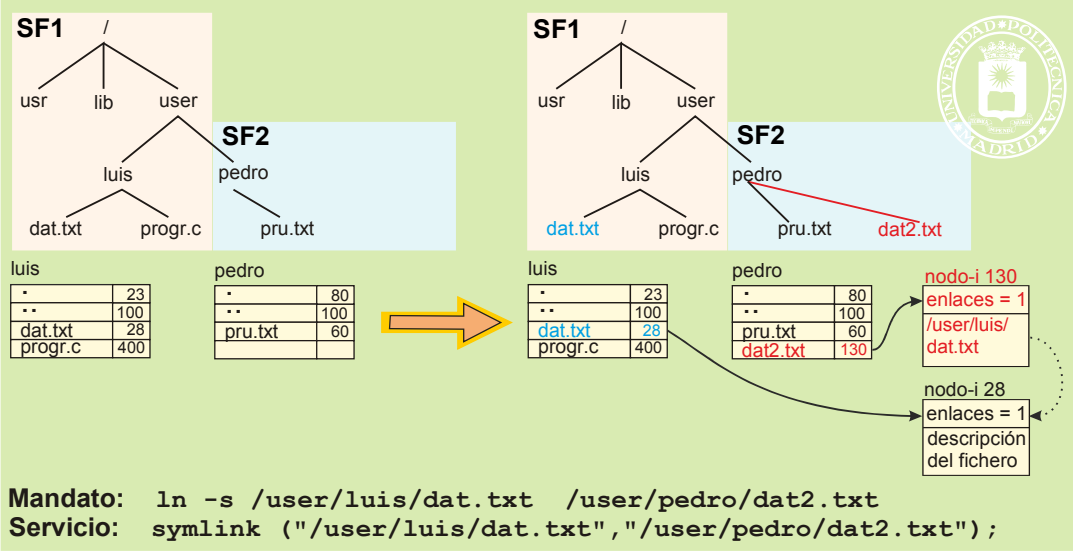


Figura 5.18 Implementación de un enlace simbólico en UNIX. Puede hacerse sobre un elemento de otro sistema de fichero, el SF1 en la figura.

La figura 5.18 muestra la implementación de un enlace simbólico en UNIX sobre un fichero. El esquema sobre un directorio es similar.

Se puede observar que al nuevo nombre `dat2.txt` se le asigna un nodo-i libre, en el que se almacena el nombre textual del fichero existente, es decir `/user/luis/dat.txt`. Observe que **no se incrementa** el número de enlaces del fichero original `dat.txt`. Si se borrase `dat.txt`, llegaría a cero su contador de referencias, por lo que el fichero se elimina. Sin embargo, no se actúa sobre `dat2.txt`, quedando 'colgado' el enlace simbólico. Si se intenta abrir ahora `dat2.txt`, el servicio devolverá un error, al no poder encontrar el fichero `/user/luis/dat.txt`.

5.8. SISTEMA DE FICHEROS

Un sistema de ficheros es un **conjunto autónomo** de informaciones incluidas en una unidad de almacenamiento (volumen) que permiten su explotación. Con la operación de **dar formato** se construye un sistema de ficheros sobre un volumen (mandato **FORMAT** en Windows y **mkfs** en UNIX). Cada tipo de servidor de ficheros organiza el formato del sistema a su manera. La figura 5.19 presenta algunos formatos básicos de sistemas de ficheros.

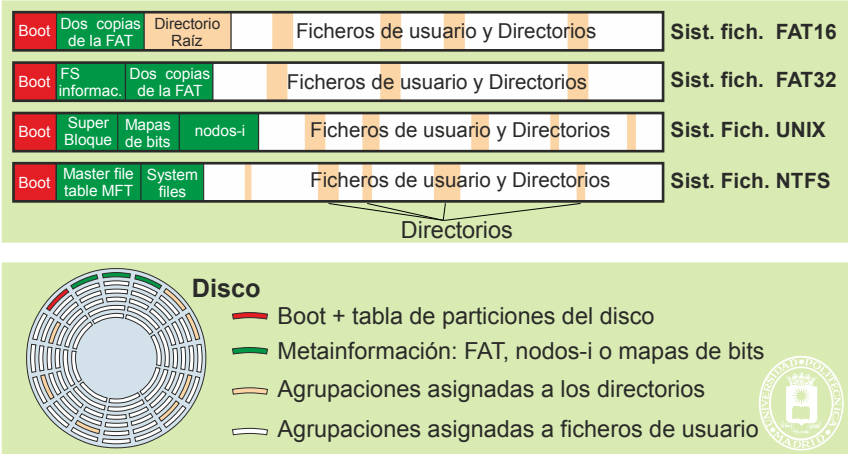


Figura 5.19 *Formato básicos de varios sistemas de ficheros.*

El sistema de ficheros incorpora los siguientes elementos:

- **Información neta**: Ficheros regulares, programas y datos, constituyen la información neta que el usuario almacena en el sistema de ficheros.
- **Metainformación**, compuesta por:
  - Mapa del fichero, que determina la estructura física de los ficheros.
  - Atributos de los ficheros.
  - Directorios, que incluyen la información relativa a los nombres de los ficheros.

Se puede observar en la figura 5.19 que hay una parte del volumen directamente asignada a la metainformación como las copias de la FAT, los mapas de bits o los nodos-i, y una zona, que está representada con fondo blanco en la figura, que se organiza en agrupaciones y en la que se almacenan los ficheros de usuario, además de la información de directorio.

Sistema de ficheros tipo UNIX

El sistema de ficheros tipo UNIX incluye el superbloque, dos mapas de bits, los nodos-i y los datos.

El **superbloque** incluye informaciones tales como las siguientes:

- Tamaño de la agrupación.
- Tamaños del propio superbloque, de los mapas de bits y de nodos-i.
- Número total de agrupaciones disponibles.
- Número total de nodos-i disponibles.
- Número del primer nodo-i. Suele ser el número 2 y se utiliza para el directorio raíz.

Existen dos **mapas de bits** uno para nodos-i y otro para agrupaciones. El mapa de bits se analiza en la siguiente sección.

El **nodo-i** clásico UNIX tiene un tamaño de 128 B y utiliza palabras de 4 B. Cada fichero tiene asociado un nodo-i, siendo el número de nodo-i el identificador interno del fichero. Su estructura es la de la figura 5.20.

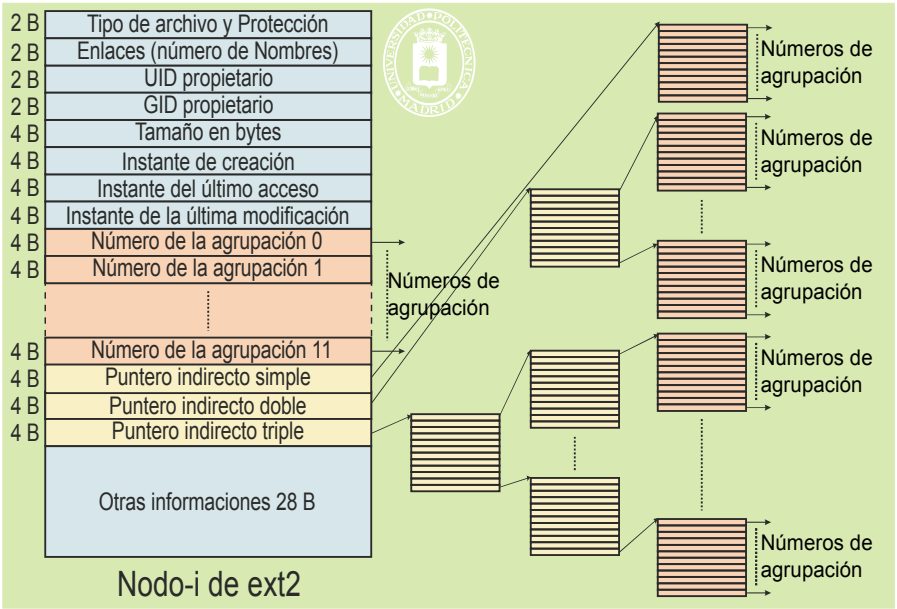


Figura 5.20 *Formato del nodo-i clásico de UNIX, concretamente del sistema de ficheros ext2, y las agrupaciones adicionales de indirectos.*

Destacamos el campo de **enlaces**, que indica el número de nombres que tiene el fichero. Cuando dicho número llega a 0 significa que el fichero se puede eliminar.

El nodo-i incluye espacio para identificar directamente las primeras agrupaciones del fichero (12 en el caso del ext2). Si esto no es suficiente, se utiliza el puntero indirecto simple que identifica una agrupación que contiene los números de las agrupaciones 11, 12, 13, y siguientes. Si esto no es suficiente, se utiliza el puntero indirecto doble, que identifica una agrupación que contiene punteros indirectos simples. Finalmente, el puntero indirecto triple apunta a una agrupación que contiene punteros indirectos dobles. Estas agrupaciones necesarias para establecer el mapa del fichero se almacenan en el espacio de datos, al igual que los directorios.

De acuerdo a este esquema de punteros, el tamaño máximo que se permite, suponiendo una agrupación de 4X KiB, es el siguiente:

- nodo-i 12 agrupaciones = 48X KiB
- Indirecto simple: X Ki-agrupaciones = $4X^2$ MiB
- Indirecto doble: X^2 Mi-agrupaciones = $4X^3$ GiB
- Indirecto triple: X^3 Gi-agrupaciones = $4X^4$ TiB

Dando un total de: 48X KiB + $4X^2$ MiB + $4X^3$ GiB + $4X^4$ TiB

Para un valor de agrupación de 2 KiB, X = ½, y vemos que el mayor fichero podría llegar a tener algo más de ¼ TiB. Sin embargo, como el tamaño real del fichero se almacena en una palabra de 4 B, el mayor valor que puede tener será de 2^{32} B = 4 GiB.

5.8.1. Gestión del espacio libre

El sistema de ficheros contiene una estructura de información con los recursos libres (DFF y agrupaciones) que se utiliza en la creación de un fichero, en la asignación de espacio al fichero y en la eliminación de un fichero. Existen básicamente dos alternativas a la hora de diseñar la estructura de información que permite determinar los elementos libres, que son el mapa de bits o la lista de recursos libres.

El **mapa de bits**, o vector de bits contiene un bit por recurso existente (DFF o agrupación). Si el recurso está libre, el valor del bit asociado al mismo es 1, pero si está ocupado es 0. Ejemplo, sea un disco en el que las agrupaciones de número 2, 3, 4, 8, 9 y 10 están ocupados y el resto libres, y en el que los descriptores DFF de número 3, 4 y 5 están ocupados. Sus mapas de bits de serían:

MB de agrupaciones: `1100011100011....`

MB de descriptores DFF: `1110001...`

El mapa de bits es fácil de implementar y sencillo de usar. Además, es eficiente si el dispositivo no está muy lleno o muy fragmentado.

La **lista de recursos libres** consiste en mantener enlazados en una lista todos los recursos disponibles (DDFs o agrupaciones) manteniendo un apuntador al primer elemento de la lista. Este método no es eficiente, excepto para dispositivos muy llenos y fragmentados. En el caso de las agrupaciones la lista puede construirse dentro de las propias agrupaciones libres, no necesitando espacio de disco adicional.

Asignación de recursos

Cuando se requiere un DFF (al crear un nuevo fichero) o una nueva agrupación (al aumentar el tamaño de un fichero) se ha de buscar en la estructura de recursos libres, un DFF o agrupación libre, que se marca como ocupada y se asigna al fichero.

Eliminación de un fichero

Cuando el número de nombres de un fichero llega a cero se puede eliminar, lo que se hace marcando en las estructuras de información de recursos libres el DFF y sus agrupaciones como libres (e.g. marcando en el mapa de bits el nodo-i y las agrupaciones como libres). Observe que el fichero realmente **no se borra**, las agrupaciones seguirán con su contenido, que se considera basura, pero quedan disponibles para volver a ser asignadas a otro fichero.

Esto puede plantear un problema de seguridad, puesto que, en principio, parece que el usuario que recibe alguna de esas agrupaciones podría leer su contenido. Para evitar que esto ocurra, el DFF incluye el tamaño real del fichero y el servidor de ficheros no permitirá nunca leer más allá de este tamaño, evitando que se pueda leer la basura dejada por un fichero anterior.

5.9. SERVIDOR DE FICHEROS

El servidor de ficheros es la capa de *software* del sistema operativo que se sitúa entre los dispositivos y los usuarios con el objetivo de alcanzar las siguientes metas:

- Debe presentar una visión lógica simplificada de los ficheros almacenados en los dispositivos de almacenamiento secundario. Visión que se ha presentado en la página 217.
- Debe establecer un esquema de nombrado lógico para que los usuarios identifiquen cómodamente a los ficheros. Esquema que se ha presentado en la página 221.

- Debe suministrar una visión lógica uniforme de los demás dispositivos y mecanismos de comunicación, presentándolos como ficheros especiales, tal y como muestra la figura 5.21.
- Debe ofrecer primitivas de acceso cómodas e independientes de los detalles físicos de los dispositivos y de la estructura del sistema de ficheros.
- Debe incorporar mecanismos de protección que garanticen que los usuarios solamente puedan utilizar los ficheros a los que tenga permiso.

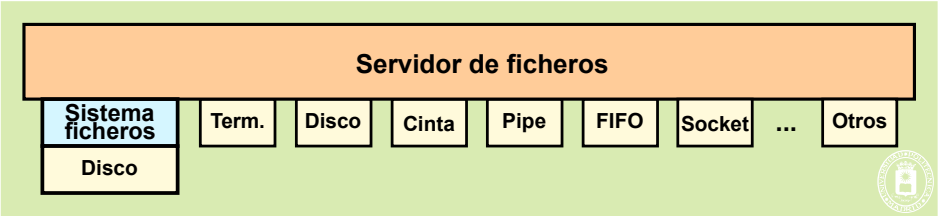


Figura 5.21 *El servidor de ficheros permite tratar de forma homogénea tanto a los ficheros de los sistemas de ficheros ubicados en dispositivos de almacenamiento secundario como a otros dispositivos y mecanismos de comunicación.*

El servidor de ficheros tiene dos tipos de funciones muy distintos entre sí. Por un lado, debe definir e implementar la **visión lógica** de usuario del sistema de entrada/salida, incluyendo servicios, archivos, directorios, sistemas de archivos, etc. Por otro lado, debe definir e implementar los algoritmos y estructuras de datos a utilizar para hacer corresponder la visión del usuario con el sistema **físico** de almacenamiento secundario y resto de dispositivos.

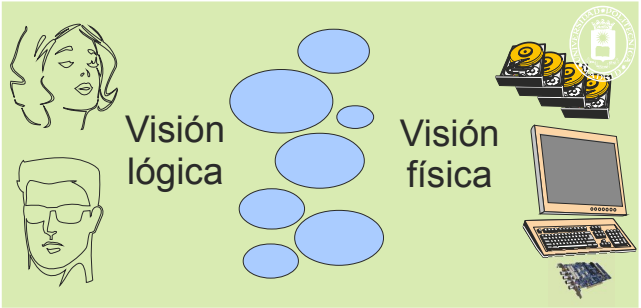


Figura 5.22 *El sistema de ficheros abstrae al usuario de la realidad física de los dispositivos y presenta una visión lógica que permite identificar ficheros y dispositivos mediante un esquema de nombrado y acceder a ellos mediante unas primitivas homogéneas.*

5.9.1. Vida de un fichero

Como se muestra en la figura 5.23, para trabajar con un fichero hay que definir una **sesión** delimitada por los servicios de apertura de fichero y de cierre del descriptor de fichero.

- Se crea el fichero
  - Se abre → se obtiene un fd (descriptor de fichero)
    - Se opera (escribe, lee, ...) a través del fd
  - Se cierra el fd
- Se elimina

Sesión

Figura 5.23 *Vida de un fichero.*

La **creación** de un fichero requiere permisos de acceso al subdirectorio en el que incluye el fichero, así como permisos de escritura sobre dicho directorio. Esta operación supone los siguientes pasos:

- Asignarle al fichero una estructura DFF (en UNIX un nodo-i) libre y rellenarla.
- Incluir el nombre local en el subdirectorio correspondiente.

La **apertura** del fichero es una operación compleja que se analizará más adelante y que devuelve un descriptor de fichero o manejador. La apertura marca el comienzo de una sesión.

Sobre un fichero abierto se pueden realizar operaciones tales como **lectura** o **escritura**, utilizando el descriptor de fichero y no el nombre lógico. La escritura puede suponer el crecimiento del fichero.

Para terminar la sesión se ha de **cerrar** el descriptor de fichero. Si bien esta operación se hace automáticamente cuando un proceso finaliza, conviene siempre cerrar los descriptores cuando ya no se van a utilizar. En algunos casos, que veremos más adelante, es fundamental cerrar los descriptores cuando ya no se utilizan, para que el programa funcione correctamente.

El fichero se **elimina** cuando se han eliminado todos sus nombres lógicos. A partir de ese momento no se puede acceder al fichero por lo que se marcan como libres todos sus recursos, es decir, su DFF y sus agrupaciones. Es de destacar que el fichero **no se borra**, el sistema operativo no pierde tiempo borrando las agrupaciones, por lo que quedan con su contenido, que pasa a ser basura.

Aclaración 5.2. Una vez que un proceso abre un fichero, sigue pudiendo acceder al mismo aunque se cambien los permisos de forma que el proceso ya no tenga derecho de acceso. Esta situación se produce puesto que los permisos solamente se comprueban en la operación de apertura.

Adicionalmente, si el fichero es eliminado, el proceso puede seguir leyendo y escribiendo del mismo. Los recursos del fichero eliminado solamente se liberan cuando no hay ningún proceso que lo tenga abierto.

5.9.2. Descriptores de fichero

En los sistemas UNIX, el descriptor de fichero es un número entero no negativo que identifica un fichero abierto y que se obtiene al ejecutar el servicio de apertura de fichero (en Windows se obtiene un manejador y la funcionalidad es similar a la descrita para UNIX). Los descriptores de fichero se almacenan en el BCP del proceso en forma de vector, como se muestra en la figura 5.24. El índice del vector es el descriptor de fichero, mientras que el contenido es una referencia interna para que el servidor de ficheros pueda alcanzar al fichero.

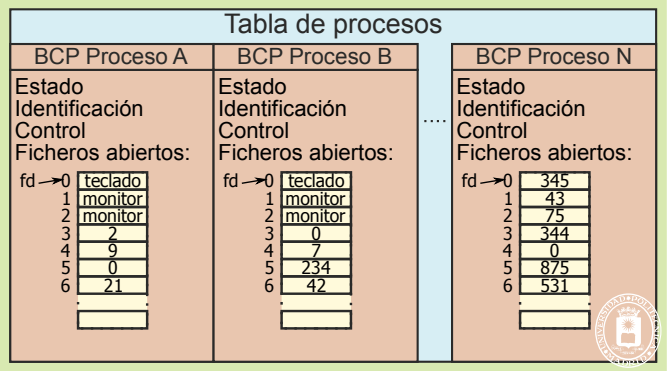


Figura 5.24 *El BCP del proceso incluye una tabla de descriptores de fichero.*

Los descriptores de fichero, que se suelen representar con **fd** (*file descriptor*), se asignan por orden, empezando por el 0 y buscando el primero que esté cerrado. Para indicar que el descriptor está cerrado se pone su contenido a 0, por tanto el descriptor 5 del proceso A de la figura 5.24 está cerrado.

Los procesos tienen abiertos al menos los tres primeros descriptores que tienen una utilización especial. Estos descriptores se denominan estándar y tienen la siguiente asignación:

fd = **0 entrada estándar** (STDIN_FILENO)
fd = **1 salida estándar** (STDOUT_FILENO)
fd = **2 salida de error** (STDERR_FILENO)

En unistd.h están definidas las constantes simbólicas STDIN_FILENO, STDOUT_FILENO y STDERR_FILENO con valor 0, 1 y 2 respectivamente.

Tal y como se muestra en la figura 5.25, en un proceso interactivo la entrada estándar está asociada al teclado, mientras que la salida estándar y la salida de error lo están a la pantalla.

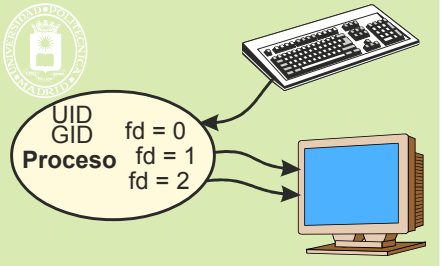


Figura 5.25 *Los descriptores estándar de un proceso interactivo están asociados al terminal.*

En UNIX el **proceso hijo hereda los descriptores** de fichero del padre. Además, todos los ficheros abiertos siguen abiertos después del servicio `exec`.

En Windows el proceso hijo puede heredar los manejadores de fichero, lo que se selecciona mediante el parámetro `fInheritHandles` del servicio `CreateProcess`.

Redirección de descriptores estándar

Los descriptores estándar, al igual que cualquier descriptor, se pueden cerrar y volver a reasignar seguidamente a otro fichero, regular o especial. Esta operación de cambiar el fichero asociado a un descriptor estándar se denomina redirección y es utilizada, por ejemplo, por el *shell* cuando ejecuta un mandato como el `ls > fich`. El *shell* crea un proceso hijo, redirecciona la salida estándar del hijo al fichero `fich` y cambia el programa del hijo al `ls` mediante un servicio `exec`. El `ls` produce su resultado por la salida estándar, por lo que la escribe en `fich`.

5.9.3. Semántica de coutilización

El sistema operativo permite que varios procesos accedan de forma simultánea a un mismo fichero. La **semántica de coutilización** especifica el efecto de varios procesos accediendo de forma simultánea al mismo archivo.

Aclaración 5.3. En algunos sistemas operativos, como UNIX, los ficheros se pueden coutilizar siempre. Sin embargo, en otros sistemas operativos, como Windows, hay que especificar, al abrir el fichero, que se autoriza a compartirlo para operaciones lectura, escritura o borrado.

Dado que existen grandes diferencias entre los accesos a ficheros locales y los accesos a ficheros remotos, los cuales presentan unas latencias apreciables en las operaciones de lectura y escritura, existen diversas semánticas de coutilización. El usuario debe ser consciente de la semántica que se aplica a los ficheros que accede.

Un aspecto muy importante de la coutilización es si **se comparte o no el puntero del fichero**. Veamos dos situaciones. Si hay dos usuarios que están accediendo, por ejemplo, a un fichero de ayuda, está claro que cada uno desea leer el fichero a su ritmo y que las operaciones de lectura que hace el otro usuario no le afecten. Para ello, cada uno ha de tener un puntero distinto, que avance según él va leyendo. Consideremos ahora una aplicación dividida en varios procesos en la que se desea mantener un fichero de bitácora o log, en el que los procesos van escribiendo eventos de cómo procede su ejecución. Si cada proceso tiene su puntero, unos sobrescribirán las anotaciones de otros. En este caso, es fundamental que los procesos compartan el puntero. El servidor de ficheros deberá posibilitar las dos situaciones, que los procesos compartan o no compartan el puntero. Veremos más adelante que, cuando un proceso hereda un descriptor de fichero, comparte el puntero, sin embargo, cuando un proceso abre un fichero se crea un nuevo puntero para ese fichero.

Semántica UNIX

Una semántica aplicable especialmente a los sistemas de ficheros locales. Sus características principales son las siguientes:

- Los procesos pueden compartir ficheros, es decir, varios procesos pueden tener abierto el mismo fichero.
- Los procesos pueden compartir el puntero cuando han heredado el descriptor (existe relación de parentesco).
- Los procesos pueden tener punteros distintos cuando abren el fichero de forma independiente.
- Las escrituras son inmediatamente visibles para todos los procesos con el fichero abierto. Si el proceso A escribe e inmediatamente después el proceso B lee, leerá lo que escribió A.
- La coutilización afecta, lógicamente, también a los metadatos.

Problema: El sistema operativo serializa las operaciones de escritura y lectura de forma que no comienza una operación hasta que no haya terminado la anterior, evitando que se entremezclen. Sin embargo, no garantiza el orden en que se ejecutan dichas operaciones (su orden de ejecución puede incluso venir afectado por el algoritmo de optimización del disco). Si varios procesos escriben sobre la misma porción del fichero, el resultado final es indeterminado.

Solución si una aplicación dividida en varios procesos requiere un orden específico en los accesos a un fichero compartido, ha de usar algún mecanismo de sincronización entre sus procesos, como puede ser usar cerrojos sobre la zona del fichero sobre la que se está escribiendo.

Semántica de sesión

Es una semántica más relajada que la anterior por lo que es aplicable a sistemas de ficheros remotos. Sus características principales son las siguientes:

- Las escrituras que hace un proceso no son visibles para los demás procesos con el archivo abierto.
- Las escrituras solamente se hacen visibles cuando el proceso que escribe cierra su sesión sobre el fichero y otro proceso abre otra sesión. Los cambios, por tanto, se hacen visibles para las futuras sesiones.

Problema: Un fichero puede asociarse temporalmente a varias imágenes si varios procesos abren sesiones de escritura. Cada proceso tiene su propia versión del fichero, que no incorpora o incluso es incompatible con las modificaciones realizadas por otros procesos. ¿Qué copia es la válida?

Solución: Se hace necesario sincronizar los procesos explícitamente mediante algún mecanismo de sincronización proporcionado por el sistema operativo.

Semántica de versiones

Es una semántica parecida a la anterior, con la peculiaridad de que las actualizaciones del fichero se hacen sobre copias distintas, diferenciadas mediante un número de versión.

- Las modificaciones sólo son visibles cuando se consolidan las versiones al cerrar la sesión de modificación.
- Se requiere una sincronización explícita si se desea tener actualización inmediata.

Semántica de ficheros inmutables

Esta semántica es adecuada en servicios particulares, como los de *back-up* y los repositorios de información con gestión de versiones. Las características de esta semántica son las siguientes:

- Una vez creado el fichero sólo puede ser compartido para lectura y no cambia nunca.
- Una modificación supone crear un nuevo fichero con otro nombre, a menos que se borre el anterior.
- Para optimizar su implementado se puede utilizar una técnica tipo *copy on write*, como la vista en la página 171. Para ello es necesario establecer atributos de lectura y escritura por bloque como en la memoria virtual.

5.9.4. Servicio de apertura de fichero

La apertura de un fichero es una operación que parte del nombre lógico de un fichero y obtiene el descriptor interno IDFF que permite utilizar el fichero. La operación devuelve un descriptor de fichero (UNIX) o manejador (Windows) que es utilizado por el usuario para realizar operaciones sobre el mismo. Ejemplos de este servicio son `open` y `creat` en UNIX, y `CreateFile` en Windows.

Como se indicó en la sección "3.4.8 Privilegios del proceso UNIX", el proceso UNIX tiene dos identidades la real y la efectiva (UID y GID efectivos). El servicio de apertura de fichero utiliza la identidad efectiva para comprobar si el proceso tiene derechos de apertura del fichero. En el resto de la sección nos centraremos en la apertura de ficheros para el caso de UNIX.

En la operación de apertura se solicitan determinados **permisos de acceso** al fichero, tales como lectura y escritura. En algunos casos, como en Windows, también hay que especificar los **permisos de coutilización** que se solicitan, que pueden ser de lectura, escritura o borrado, siendo la opción por defecto la **no** coutilización. En UNIX no se especifican permisos de coutilización, puesto que existen siempre. Es recomendable solicitar los menores permisos posibles, así, aunque se tenga permiso de escritura sobre un fichero, si solamente vamos a leerlo, se debe abrir solamente para lectura.

Recorrer el directorio

La apertura de un fichero exige recorrer el árbol de directorios hasta encontrar el nombre buscado, es decir, recorrer la ruta de acceso, comprobando, al mismo tiempo, los permisos en cada directorio de la búsqueda. Este recorrido comienza en el directorio raíz cuando se parte de un nombre absoluto, pero comienza en el directorio de trabajo para los nombres relativos.

Para comprender lo que supone este recorrido vamos a plantear que se quiere abrir para lectura y escritura el fichero /Prog/Voit/Buit/Yuit de la figura 5.26, suponiendo que es un sistema de ficheros de tipo UNIX. Para otros sistemas de ficheros el proceso es similar, pero utilizando el correspondiente IDFF, por ejemplo, el identificador de registro MFT de NTFS.

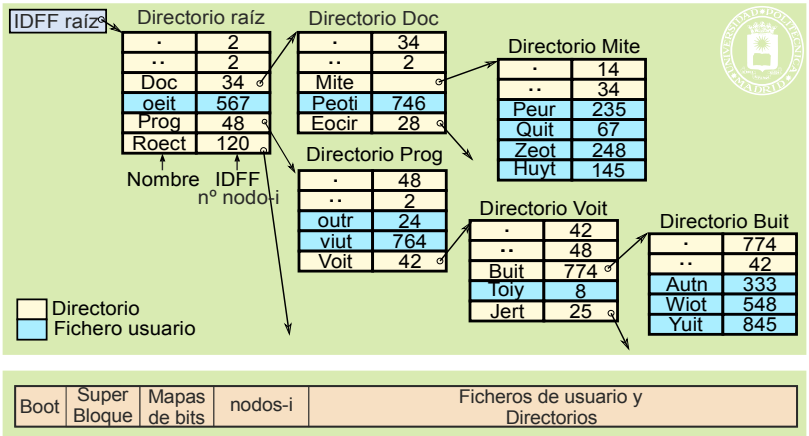


Figura 5.26 *Parte de las tablas directorio de un árbol de directorios.*

Suponiendo que no hay ninguna información del sistema de ficheros en memoria, las operaciones a realizar son las siguientes.

1. Hay que determinar el número que tiene el nodo-i del directorio raíz, que, según la figura, es el 2. Esta información se encuentra en el superbloque. Por lo tanto, habría que leer esta información del disco (en realidad al montar el dispositivo se trae el superbloque a memoria, por lo que esta lectura del disco no es necesaria).
2. Hay que leer del disco el nodo-i 2. Se comprueba que el tipo de fichero es directorio. También se comprueba con el UID y DID efectivos del proceso, que éste tiene los permisos para atravesar el directorio. Si todo es correcto se pasa al punto siguiente.
3. Se leen del disco las agrupaciones que contienen el directorio. Esto puede ser una o más accesos al disco, dependiendo del tamaño y fragmentación del directorio.
4. Se busca el nombre Prog dentro del raíz y se encuentra su n.º de nodo-i, que es 48.

5. Se lee del disco el nodo-i 48. Se comprueba que es de tipo directorio y que se tienen permisos para atravesar con el UID y DID efectivos del proceso. Si todo es correcto se pasa al punto siguiente.
6. Se leen del disco las agrupaciones que contienen el directorio Prog.
7. Se busca el nombre Voit dentro de Prog y se encuentra su n.º de nodo-i, que es 42.
8. Se lee del disco el nodo-i 42. Se comprueba que es de tipo directorio y que se tienen permisos para atravesar. Si todo es correcto se pasa al punto siguiente.
9. Se leen del disco las agrupaciones que contienen el directorio Voit.
10. Se busca el nombre Buit dentro de Voit y se encuentra su n.º de nodo-i, que es 25.
11. Se lee del disco el nodo-i 25. Se comprueba que es de tipo directorio y que se tienen permisos para atravesar. Si todo es correcto se pasa al punto siguiente.
12. Se leen del disco las agrupaciones que contienen el directorio.
13. Se busca el nombre Yuit y se encuentra su n.º de nodo-i, que es 845.
14. Se lee del disco el nodo-i 845. Se comprueba que es de tipo regular y que se tienen los permisos de lectura y/o escritura con el UID y GID efectivos del proceso, por lo que se puede abrir el fichero.
15. Si todo es correcto el nodo-i queda almacenado en memoria, en la tabla de nodos-i en memoria, puesto que se seguirá utilizando hasta que se cierre el fichero.

Se puede observar que por cada directorio es necesario acceder al disco por un lado para leer su nodo-i y seguidamente, al menos un acceso, para leer las agrupaciones que componen el directorio.

Para mejorar las prestaciones, el servidor de ficheros puede mantener en memoria información de los directorios recientemente utilizados, de forma que se minimizan los accesos a disco si parte de la ruta ya está en memoria.

Descriptor de fichero

La operación de apertura devuelve el descriptor de fichero. Para ello, la operación de apertura sigue con el siguiente paso:

16. El servidor de ficheros ha de buscar, en la tabla de ficheros abiertos (tabla de fd), del proceso que ha solicitado el servicio, la primera entrada de descriptor cerrado, es decir, la primera entrada que contenga 0. El descriptor de fichero abierto no es más que la posición de la mencionada entrada. Por ejemplo, en la tabla de descriptores del proceso N, mostrada en la figura 5.27, se seleccionaría la posición sexta, por lo que el descriptor de fichero devuelto será el 5.

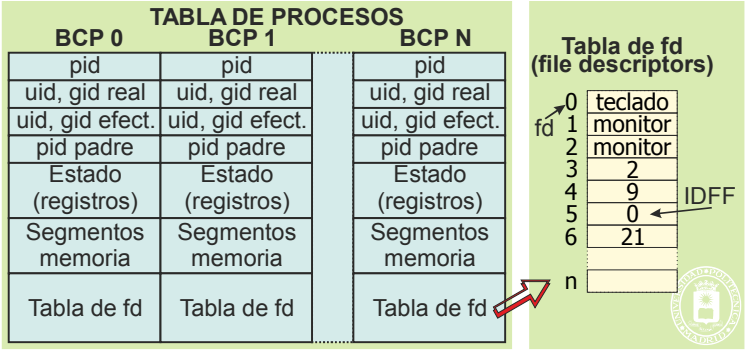


Figura 5.27 *Tabla de descriptores de ficheros abiertos del proceso N.*

Contenido de la tabla de descriptores de fichero

En una primera solución podríamos pensar que dentro de la tabla de descriptores se introduce el número de nodo-i, lo que en nuestro ejemplo sería el valor 845.

Sin embargo, esta solución tiene el siguiente problema ¿dónde se ubica el puntero de posición para el fichero que se acaba de abrir?

Se podría plantear el añadir una columna a la tabla de descriptores para contener los punteros. Esta solución es adecuada cuando se quiere que el proceso tenga su propio puntero para el fichero. Sin embargo, no sirve cuando se desea compartir el puntero por varios procesos. Para resolver esta situación y permitir que el puntero sea compartido o no compartido se establece una tabla común a todos los procesos, que se denomina tabla intermedia y que se analiza seguidamente.

Tabla intermedia

La figura 5.28 muestra una visión simplificada de la tabla intermedia, conteniendo los dos campos de número de nodo-i y puntero de posición del fichero. Para completar la operación de apertura se realiza el siguiente paso:

17. Se busca una entrada libre en la tabla intermedia y se rellena con el número de nodo-i del fichero abierto y se pone el puntero a 0. El número de esa entrada, al que llamaremos identificador intermedio o II, se introduce en la entrada de la tabla de descriptores. Esta situación queda reflejada en la mencionada figura 5.28, en la que se pueden observar los siguientes puntos:
    - El contenido de la tabla fd de descriptores apunta a la entrada de la tabla intermedia.
    - La entrada 0 de la tabla intermedia no existe, puesto que el valor 0 en la tabla de descriptores indica que el descriptor está cerrado.

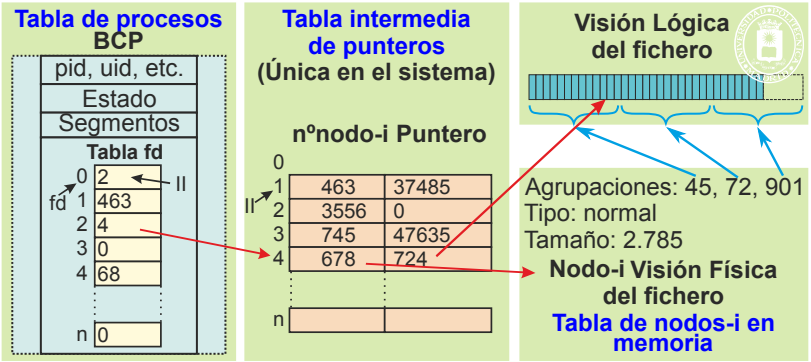


Figura 5.28 *Visión simplificada de la tabla intermedia y su relación con la tabla de descriptores de fichero, así como con la tabla de nodos-i en memoria.*

En la mencionada figura también se puede apreciar que el DFF del fichero abierto está copiado en memoria.

Sin embargo, a la hora de solicitar la apertura de un fichero se puede especificar si se desea abrir sólo para lectura, sólo para escritura o para lectura y escritura. Esta información es necesaria mantenerla para autorizar o no los accesos que se soliciten sobre el fichero. Esto se resuelve añadiendo en la tabla intermedia un campo para los bytes rw que indiquen cómo se abrió el fichero, como se muestra en la figura 5.29. Es de destacar que, aunque el usuario tuviese permisos de lectura y escritura, si abre el fichero solamente para lectura, las operaciones de escritura devolverán error. Por tanto, no hay que confundir los permisos del fichero, almacenados en su DFF con los permisos de apertura almacenados en la tabla intermedia.

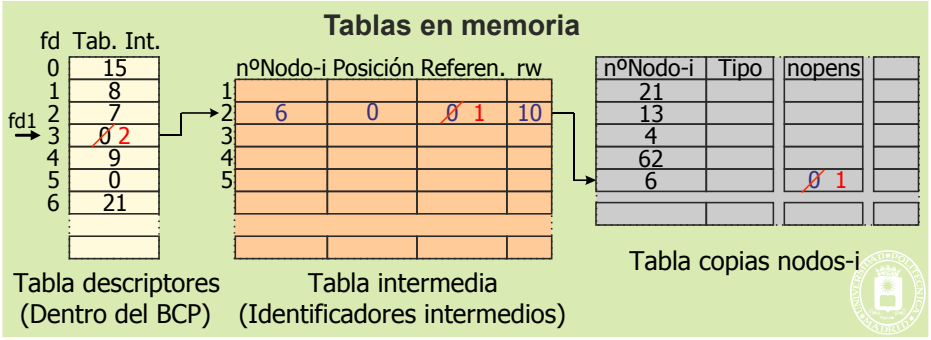


Figura 5.29 *Estructuras de información del servidor de ficheros para un sistema UNIX con las modificaciones producidas por un open. Se aprecia la tabla de descriptores en el BCP del proceso, la tabla intermedia y la tabla de nodos-i en memoria.*

Destacamos los campos de referencias en la tabla intermedia y de nopens en la tabla de nodos-i mostrados en la mencionada figura y que analizamos seguidamente.

Gestión de los campos de referencias y de nopens.

Una entrada de la tabla intermedia puede estar referenciada en más de una tabla de descriptores. En efecto, al crear un proceso hijo se copia la tabla de descriptores, esto ocurre siempre en UNIX y también en Windows si se solicita, por tanto, se crean nuevas referencias a las correspondientes entradas de la tabla intermedia. Lo mismo ocurre con el servicio de duplicar descriptor. Como una entrada de la tabla intermedia solamente quedará libre si no hay ninguna referencia a ella, es necesario llevar la cuenta del número de referencias. Por ello, se añade en la tabla intermedia un nuevo campo que actúe de contador de referencias.

Observación 5.1. Es de observar que, siempre que un recurso pueda tener un número variable de usuarios, es necesario tener un mecanismo para determinar si el recurso queda libre. Una solución sencilla y eficaz es mantener un contador de usuarios, de forma que cuando dicho contador llegue a cero signifique que el recurso queda libre.

Algo similar ocurre con la copia en memoria de los nodos-i. En efecto, cuando se abre un fichero que ya está abierto no se hace una nueva copia del nodo-i en memoria, sino que se comparte. Por tanto, es necesario llevar la cuenta del número de veces que el fichero se abre, lo que se hace en el campo nopens.

La operativa es la siguiente:

- Cuando se abre un fichero se rellena la entrada de la tabla intermedia, poniendo referencias = 1. Seguidamente:
  - Si el nodo-i ya está en memoria, se incrementa la variable nopens.
  - En caso contrario, se busca una entrada libre en la tabla de nodos-i de memoria, donde se guarda el nodo-i, poniendo nopens = 1.
- Cuando se crea un proceso hijo, se incrementan las referencias de todos los descriptores copiados.
- Cuando se ejecuta un servicio de duplicar un descriptor, se incremente el correspondiente valor de referencias.

5.9.5. Servicio de duplicar un descriptor de fichero

El proceso que ejecuta este servicio obtiene un nuevo descriptor o manejador del fichero. Este descriptor o manejador referencia el mismo fichero con el mismo puntero y los mismos permisos de acceso.

Ejemplos de este servicio son `dup` y `dup2` en UNIX y `DuplicateHandle` en Windows.

Es de destacar la creación un proceso conlleva en UNIX la copia de sus descriptores de fichero, lo que supone una duplicación de los mismos, como se muestra en la figura 5.30 para el descriptor 2. En Windows en la creación de un proceso hijo se puede especificar si hereda o no los manejadores de fichero.

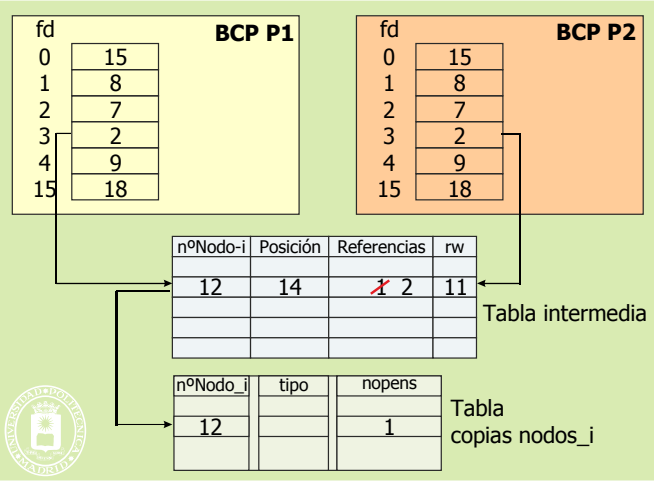


Figura 5.30 *La copia de la tabla de descriptores de ficheros que se realiza con el servicio fork implica incrementar las referencias. Se muestra en la figura solamente el caso del descriptor 3.*

5.9.6. Servicio de creación de un fichero

La creación de un nuevo fichero es una operación bastante parecida a la apertura, puesto que, entre otras cosas, deja abierto el fichero para poder escribir en él. El servicio devuelve, por tanto, un descriptor de fichero.

Hay que recorrer el árbol hasta al subdirectorio en el que se quiere insertar el nuevo fichero. Dado que hay que añadir el nuevo nombre en ese subdirectorio hay que tener **permisos de escritura** en el mismo.

Hay que buscar un DFF libre para asignárselo al fichero. Además, como el fichero queda abierto, se selecciona una entrada libre de la tabla de DFF en memoria y se rellenan sus campos, entre los que destacamos los siguientes:

- Tipo de fichero = **regular**.
- Permisos: Los solicitados en el servicio, afectados por los permisos por defecto del proceso.
- Dueño y grupo: Los del proceso que pide la creación del fichero.
- Instante de creación.

Ejemplos de este servicio son `open` y `creat` en UNIX y `CreateFile` en Windows.

5.9.7. Servicio de lectura de un fichero

El servicio de lectura de un fichero incluye los siguientes argumentos:

- Descriptor de fichero o manejador que permite identificar el fichero del que se lee. No se utiliza el nombre lógico del fichero.
- Número N de bytes que se quieren leer.
- *buffer* B del espacio de memoria del proceso en el que se quiere el resultado. El argumento que realmente se pasa es la dirección de comienzo del *buffer*.

El servicio puede devolver menos bytes de los solicitados si desde la posición del puntero al final del fichero hay menos de N bytes. Si el puntero ya se encuentra al final del fichero devuelve 0 bytes leídos, pero no da error de lectura.

Después de la lectura se incrementa el puntero del fichero con el número de bytes realmente transferidos.

Un aspecto importante es que el servidor de ficheros no conoce el tamaño del *buffer* del proceso que solicita la lectura, por tanto, escribirá a partir de la dirección de comienzo de B los N bytes solicitados. Si resulta que N es mayor que el tamaño de B, lo que ocurrirá es que se sobrescribirán las variables que tenga declaradas el proceso a continuación de B, como ocurre con el ejemplo de la figura 5.31. La MMU generaría una excepción de violación de memoria solamente en caso de que la escritura se salga de la región de memoria en la que está ubicado el *buffer*.

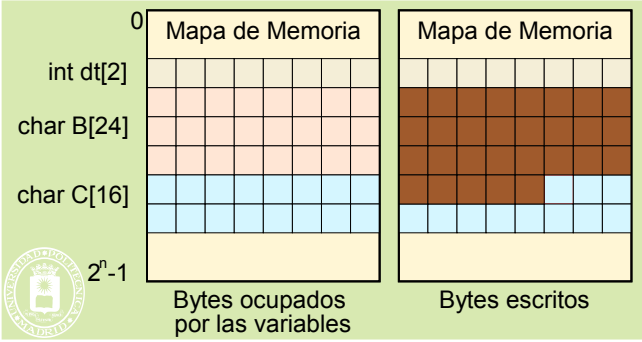


Figura 5.31 *Ejemplo de desbordamiento de buffer en una máquina de 64 bits, al ejecutar:* `read(fd1, B, 30);`.
Se escriben los 30 bytes marcados en marrón en la figura. Pero, al tener B un tamaño de 24 bytes se sobrescriben también los 5 primeros bytes de C.

La semántica de la lectura es distinta para los ficheros especiales, por ejemplo, para el teclado y los mecanismos de comunicación si no existen bytes disponibles el servicio se bloquea hasta que se pulse una tecla o llegue un mensaje.

Ejemplos de este servicio son `read` en UNIX y `ReadFile` en Windows.

5.9.8. Servicio de escritura de un fichero

El servicio de escritura de un fichero incluye los siguientes argumentos:

- Descriptor de fichero o manejador que permite identificar el fichero del que se lee. No se utiliza el nombre lógico del fichero.
- Número N de bytes que se quieren escribir.
- *buffer* B del espacio de memoria del proceso que se quiere escribir. El argumento que realmente se pasa es la dirección de comienzo del *buffer*.

El servicio devuelve el número de bytes realmente escritos. Dado que el servidor de ficheros asigna automáticamente nuevas agrupaciones a medida que el fichero las va necesitando, la operación de escritura casi siempre escribe los N bytes solicitados. Solamente si existe un error o si se acaba el espacio físico del disco o la cuota de disco del usuario se podrían escribir menos bytes de los pedidos.

Para asignar una nueva agrupación el servidor de ficheros debe buscar una libre en la estructura de información de agrupaciones libres. En general, esta agrupación no estará pegada a la última del fichero, por lo que el fichero queda **fragmentado**.

De forma similar al caso anterior, si N es mayor que el tamaño de B, se escribirán en el disco valores correspondientes a otras variables.

Ejemplos de este servicio son `write` en UNIX y `WriteFile` en Windows.

5.9.9. Servicio de cierre de fichero

Este servicio se ejecuta cuando se solicita. Al morir un proceso el sistema operativo cierra automáticamente todos sus descriptores de fichero. El servicio realiza las siguientes funciones:

- Cuando se cierra un descriptor se decrementa el campo de referencias en la tabla intermedia. Si el valor llega a 0:
  - La entrada queda libre (su contenido no se borra, queda como basura, que se sobrescribirá la siguiente vez que se utilice).
  - Hay que decrementar en 1 el campo nopens del nodo-i afectado. Observe que dicho campo indica el número de entradas de la tabla intermedia que referencian al nodo-i.
- Cuando nopens llega a 0, significa que ningún proceso tiene abierto dicho fichero, por lo que se libera la correspondiente entrada en la tabla de nodos-i en memoria. Su contenido tampoco se borra, se deja como basura.

Observe que un valor de referencias = 0 indica que la entrada de la tabla intermedia está libre y un valor nopens = 0 indica que la entrada de la tabla de nodos-i está libre. Por tanto, el servidor de ficheros analiza dichos campos cuando busca entradas libres.

La figura 5.32 muestra los cambios que se producen en las estructuras de información del sistema de ficheros al producirse varios servicios de cierre de fichero.

Ejemplos de este servicio son `close` en UNIX y `CloseHandle` en Windows.

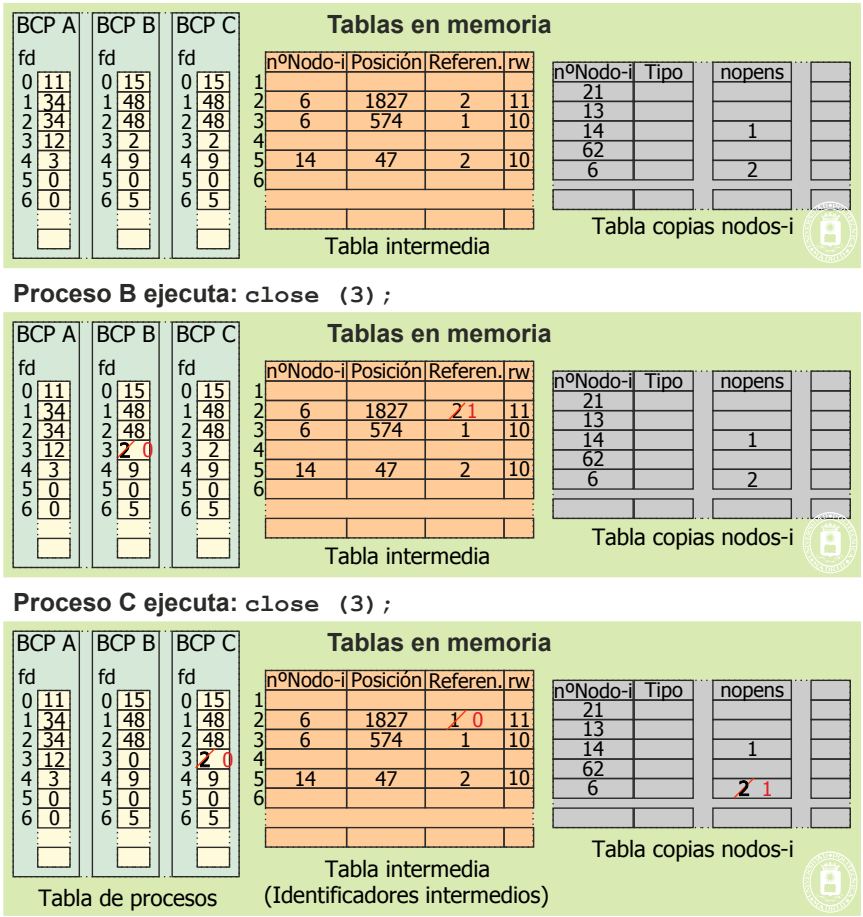


Figura 5.32 *Cambios en las tablas de descriptores de ficheros, intermedia y nodos-i al ejecutar varios servicios de cierre de descriptor de ficheros.*

5.9.10. Servicio de posicionar el puntero del fichero

Este servicio permite mover el puntero a cualquier posición del fichero. Las características principales del servicio son las siguientes:

- El servicio suele incluir un origen y un desplazamiento. El origen puede ser el principio del fichero, el final del fichero o la posición actual del puntero. El desplazamiento puede ser positivo o negativo.
- No se admiten valores negativos del puntero.
- Sin embargo, nos podemos **salir del tamaño** del fichero. Esto no implica que no se aumente el tamaño del fichero. Pero, como muestra la figura 5.33, si, una vez posicionado el puntero fuera del fichero, se escribe, el fichero crece y se genera un hueco en el mismo. En posteriores operaciones de lectura el servidor de ficheros garantizará que se lean nulos (0x00) del hueco.

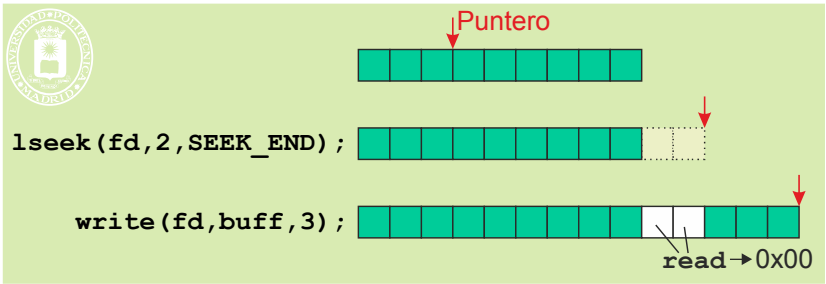


Figura 5.33 *Creación de un hueco en un fichero en UNIX.*

Cuando el hueco ocupa agrupaciones enteras, el servidor de ficheros no asigna dichas agrupaciones, puesto que es una pérdida de espacio que, además, habría que rellenar con nulos. Solamente si, posteriormente, se escribe en una zona del hueco, se asignará la o las agrupaciones necesarias. Esto lleva a una situación singular en la que el tamaño real del fichero llega a ser mayor que el espacio físico asignado al mismo.

Las técnica de los huecos permite construir, con ahorro de disco, **ficheros ralos** (*sparse file*), es decir, ficheros poco poblados (con muchos espacios en blanco).

Ejemplos de este servicio son `lseek` en UNIX y `SetFilePointer` en Windows.

5.9.11. Servicios sobre directorios

A diferencia de los ficheros regulares, la visión lógica que presenta el servidor de ficheros de un fichero directorio es de una tabla de registros, en la que cada registro es una entrada del directorio. Además, se añade un puntero que indica la entrada por la que se va leyendo. Los registros quedan definidos en la estructura `dirent` en Linux o la `WIN32_FIND_DATA` en Windows. Los elementos que más nos interesan de estas estructuras son dos: el nombre del fichero y el IDFF.

Los servicios más importantes son los siguientes:

Apertura del directorio

Este servicio devuelve un manejador que se utiliza en el resto de los servicios.

El servicio requiere permisos de lectura del directorio

Ejemplos de este servicio son `opendir` en UNIX y `FindFirstFile` en Windows, que además de abrir el fichero devuelve su primera entrada, es decir, la entrada «.».

Lectura entrada del directorio

Este servicio permite leer la siguiente entrada del directorio. Además, avanza el puntero al siguiente registro.

El servicio fracasa, entre otras causas, si se lee llegado al final de directorio. Es necesario analizar la variable de error (`errno` en UNIX o `GetLastError` en Windows) para determinar si se trata de un error o de fin de directorio.

Ejemplos de este servicio son `readdir` en UNIX y `FindNextFile` en Windows.

Cerrar directorio

Servicio que cierra la asociación entre el manejador y la secuencia de entradas de directorio.

Ejemplos de este servicio son `closedir` en UNIX y `FindClose` en Windows.

Crear directorio

La creación de un nuevo directorio es una operación bastante parecida a la creación de un fichero.

Hay que recorrer la ruta hasta al subdirectorio en el que se quiere insertar el nuevo directorio. Dado que hay que añadir el nuevo nombre en ese subdirectorio hay que tener **permisos de escritura** en el mismo.

Hay que buscar un DFF libre para asignárselo al directorio. Seguidamente, hay que rellenar los campos del mismo, entre los que destacamos los siguientes:

- Tipo de fichero = **directorio**.
- Permisos: Los solicitados en el servicio, afectados por los permisos por defecto del proceso.
- Dueño y grupo: Los del proceso que pide la creación del directorio.
- Instante de creación.
- A diferencia de cuando se crea un fichero regular, en la creación de un directorio siempre es necesario asignar una agrupación, para poder incluir los dos nombres «.» y «..».

Ejemplos de este servicio son `mkdir` en UNIX y `CreateDirectory` en Windows.

Eliminar directorio

El servicio de eliminar un directorio se ha de realizar sobre un directorio vacío, por tanto, previamente hay que eliminar su contenido. Se tiene que tener permiso de escritura en el directorio que contiene el directorio a eliminar.

Ejemplos de este servicio son `rmdir` en UNIX y `RemoveDirectory` en Windows.

Eliminar fichero

Se borra el nombre del correspondiente directorio y se decrementa el número de enlaces. Si éste llega a 0 es cuando se elimina realmente el fichero y se marcan como libres sus recursos (DFF y agrupaciones). Se deben tener permisos de escritura en el correspondiente directorio. Si el fichero está abierto, en UNIX el servicio se ejecuta, pero el fichero realmente no se elimina hasta que se cierre el fichero, por lo que puede seguir escribiendo y leyendo de él. En Windows el servicio da error si el fichero está abierto.

Ejemplos de este servicio son `unlink` en UNIX y `DeleteFile` en Windows.

Cambiar directorio de trabajo o actual

Permite establecer un nuevo directorio como directorio de trabajo. Requiere permisos de búsqueda para el nuevo directorio.

Ejemplos de este servicio son `chdir` en UNIX y `SetCurrentDirectory` en Windows.

Cambiar el nombre de un fichero o directorio

Con este servicio se cambia el nombre de un fichero o directorio. Se puede cambiar el nombre local o se puede llevar el fichero o directorio a otra posición dentro del árbol de directorios.

Ejemplos de este servicio son `rename` en UNIX y `MoveFile` en Windows.

5.9.12. Servicios sobre atributos

El servidor de ficheros permite obtener los atributos de los ficheros y directorios, en concreto, en UNIX se puede obtener la estructura `stat`, mientras que en Windows se obtienen por separado determinados atributos.

Ejemplos de este servicio son `stat` y `fstat` en UNIX y `GetFileAttributes`, `GetFileSize`, `GetFileTime`, `GetFileType` y en Windows.

5.10. PROTECCIÓN

5.10.1. Listas de control de accesos ACL (*Access Control List*)

Una lista de control de accesos o ACL es una estructura de datos (por lo general una tabla) que contiene entradas que especifican los derechos de grupos o de usuarios individuales a objetos específicos del sistema, tales como programas, procesos o ficheros. Estas entradas son conocidas generalmente como entradas de control de acceso o ACE (*Access Control Entries*). Cada objeto accesible tiene asociada una ACL, que es una lista ordenada de varios ACE. Los privilegios o permisos establecidos en cada ACE determinan los derechos de acceso específicos, por ejemplo, si un usuario puede leer, escribir o ejecutar un objeto. En algunas implementaciones, un ACE puede controlar si un usuario o grupo de usuarios, puede alterar la ACL de un objeto.

En algunos casos el ACE puede tener expresamente la denegación de un determinado permiso.

Llamaremos grupos de protección a los grupos de usuarios que se establecen para asignación de derechos de acceso. Un usuario individual, puede pertenecer a varios grupos de seguridad.

Cuando un usuario pide acceso a un objeto, se determina a qué grupos de protección pertenece y se recorre de forma ordenada la lista ACL del objeto para ver si se permite la operación solicitada.

- Por cada ACE se comprueba si dicho usuario o alguno de sus grupos de protección coincide con el del ACL:
  - En caso positivo:
    - Se comparan los permisos solicitados con los permisos denegados en el ACL:
      - Si alguno está denegado, se termina la comprobación con **resultado negativo**.
      - Si ninguno está denegado se procede con el paso siguiente.
    - Se comparan los permisos solicitados con los permisos permitidos en el ACL. Si están todos permitidos, se termina la comprobación con **resultado positivo**.
  - En caso negativo, se pasa al siguiente ACE de la lista ACL.
- Si se llega al final de la lista, se termina la comprobación con **resultado negativo**.

5.10.2. Listas de Control de Acceso en UNIX

En UNIX la implementación de las listas de control de acceso es sencilla, puesto que solamente incluye tres ACE, el relativo al usuario UID, el relativo al grupo del usuario GID y el relativo al mundo. Además, cada ACE solamente permite tres tipos de operaciones: leer (`r`), escribir (`w`) y ejecutar (`x`). De esta forma, la ACL de un objeto requiere sólo 9 bits, información que cabe en el nodo-i del objeto. Esta solución es menos general que un sistema que use ACL de forma completa, pero su implementación es mucho más sencilla.

Este modelo conlleva el que haya que hacer ciertas simplificaciones en cuanto a las operaciones no contempladas. Por ejemplo, eliminar un objeto es posible si se puede escribir en el directorio que contiene ese objeto, atravesar un directorio es posible si se tiene activado el bit `x`, etc.

Como se ha visto en la sección "3.4.8 Privilegios del proceso UNIX" Un proceso UNIX tiene los UID y GID reales y efectivos. Para determinar los privilegios del proceso se utilizan el UID y el GID efectivos, de acuerdo al algoritmo de figura 5.34.

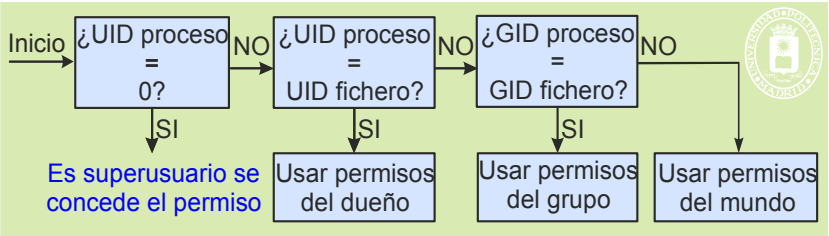


Figura 5.34 *Secuencia seguida para analizar si se puede abrir un fichero. Se utilizan el UID y GID efectivos del proceso.*

Máscara de creación de ficheros y directorios

Todo proceso UNIX contiene una máscara de permisos que se utiliza en la creación de ficheros y directorios, de forma que los bits activos en la máscara son desactivados en la palabra de protección del fichero o directorio. Se aplica,

por tanto, la función lógica: `permisos = mode & ~umask`. Por ejemplo, si máscara = 022, y se crea un fichero solicitando permisos 0777, los permisos con los que se crea realmente el fichero son 0755, puesto que la escritura en el grupo y mundo están enmascarados.

Bits SETUID y SETGID

Los ejecutables UNIX pueden tener activos los dos bits especiales SETUID y SETGID. Cuando un proceso ejecuta un programa (mediante un servicio `exec`) que tiene estos bits activos, se cambia la identidad efectiva del proceso por la del dueño del fichero. En concreto:

- Si está activo SETUID en el ejecutable, el UID efectivo del proceso pasa a ser el UID del dueño del ejecutable.
- Si está activo SETGID en el ejecutable, el GID efectivo del proceso pasa a ser el GID del ejecutable.

Los bits SETUID y SETGID de un ejecutable se pueden activar mediante el servicio `chmod`.

Ejemplos del mandato ls

Ejemplos de la salida del mandato ls:

```
drwxr-x---  2 pepito prof   48 Dec 26 2001  News
drwxr-xr-x  2 pepito prof   80 Sep 29 2004  bin
lrwxrwxrwx  1 root root      3 Jan 23 18:34 lvremove -> lvm
lrwxrwxrwx  1 root root      3 Jan 23 18:34 lvrename -> lvm
drwxrwxrwt 16 root root   1928 Apr 9 20:26  tmp
-rwxr-xr-x  1 root root   2436 Dec 26 2001  termwrap
-rwsr-xr-x  1 root root  22628 Jan 5 10:15  mount.cifs
```

El carácter inicial especifica el tipo de fichero, de acuerdo a lo siguiente:

- fichero regular
d directorio
l enlace simbólico
b dispositivo de bloques
c dispositivo de caracteres
p FIFO o *pipe*
s *socket* UNIX

Por otro lado, el bit **x** puede aparecer como una **s** o como una **t**, de acuerdo a lo siguiente:

- Si aparece una **s** en la posición **x** de usuario, significa que está activo el SETUID.
- Si aparece una **s** en la posición **x** de grupo, significa que está activo el SETGID.
- La **s** no puede aparecer en el mundo.
- Si aparece una **t** en la posición **x** del mundo en un directorio, significa que se usa como directorio de ficheros temporales. Se permite al mundo crear y borrar entradas con su UID efectivo, pero no se podrán borrar las entradas de otros usuarios. Estos directorios se pueden crear con el servicio `mktemp`.

5.10.3. Listas de Control de Acceso en Windows

Windows tiene un sistema de control de acceso a objetos uniforme que se aplica a objetos tales como procesos, *threads*, ficheros, semáforos, ventanas, etc. Este sistema de control se basa en las dos entidades siguientes, que se pueden observar en la figura 5.35: La **ficha de acceso**, asociada a cada proceso, y el **descriptor de seguridad**, asociado a cada objeto que puede ser accedido.

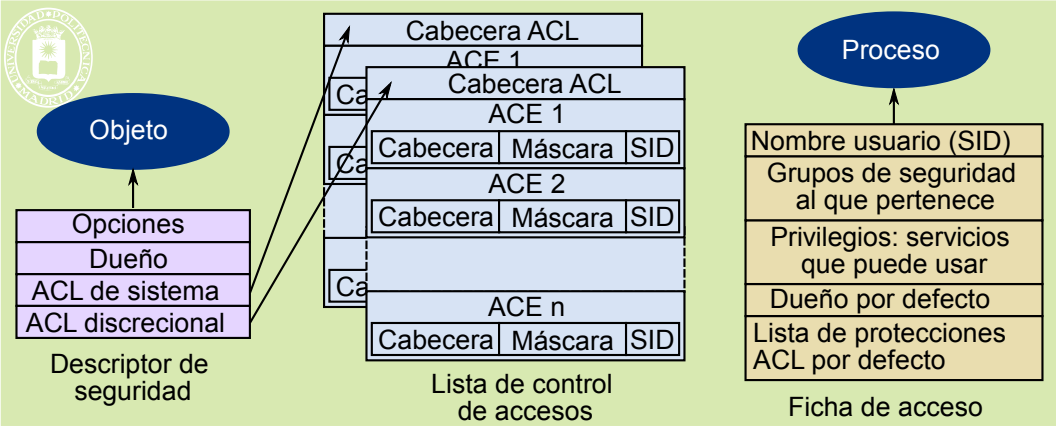


Figura 5.35 *En Windows la seguridad está basada en las fichas de acceso asociada a cada usuario, más los descriptores de seguridad asociado a cada objeto. El descriptor de seguridad apunta a dos listas de control de accesos: una de sistema y otra discrecional.*

Cuando un usuario introduce su identidad en la pantalla de bienvenida y es autenticado, se le crea un proceso *shell* al que se asigna una **ficha de acceso**, que incluye los siguientes campos:

- La **identidad** del usuario o **SID**.
- Los **grupos de seguridad** a los que pertenece dicho usuario. Cada grupo de seguridad también se identifica con un SID.
- Los **privilegios**, es decir, los servicios del sistema sensibles a la seguridad que el usuario puede ejecutar (por ejemplo, crear una ficha de acceso o hacer *backups*).
- El **dueño** por defecto que tendrán los objetos creados por el proceso. Generalmente, es el mismo que el SID, pero se puede cambiar para que sea un grupo al que pertenece el SID.
- Lista **ACL por defecto**. Es la lista inicial de protecciones que se asigna a los objetos que crea el proceso.

El **descriptor de seguridad** tiene los siguientes campos:

- El campo de **opciones** permite establecer, entre otras cosas, si se dispone de ACL de sistema y/o de ACL discrecional.
- El **dueño** del objeto que puede ser un SID individual o un SID de grupo. El dueño puede cambiar el contenido de la ACL discrecional.
- La **ACL de sistema** especifica los tipos de operaciones sobre el objeto que han de generar un registro de auditoría.
- La **ACL discrecional** especifica qué usuarios y grupos pueden hacer qué operaciones.

La secuencia que se sigue para determinar si el proceso tiene los permisos que solicita para acceder a un objeto es el planteado anteriormente en la sección “5.10.1 Listas de control de accesos ACL (Access Control List)”.

5.10.4. Servicios de seguridad

Los servidores de ficheros permiten modificar la información de seguridad de los objetos tales como ficheros y directorios.

Permisos para creación de objetos

Como hemos visto, el sistema de seguridad del servidor de ficheros permite asociar a los procesos una información que sirve para establecer los permisos de los objetos creados por el proceso. En UNIX se trata de la máscara de creación de ficheros y directorios, y en Windows se trata de la ficha de acceso. El servidor de ficheros ofrece servicios que permiten modificar esta información.

Ejemplos de este servicio son `umask` en UNIX e `InitializeSecurityDescriptor` y `SetSecurityDescriptorDacl` en Windows.

Cambio de los permisos de un objeto

El servidor de ficheros permite cambiar los permisos de un objeto, si se tiene permiso para ello. Ejemplos de este servicio son `chmod` en UNIX y `SetFileSecurity` y `SetPrivateObjectSecurity` en Windows.

Cambio del dueño de un objeto

El servidor de ficheros permite cambiar el dueño de un objeto, si se tiene permiso para ello. Ejemplos de este servicio son `chown` en UNIX y `SetSecurityDescriptorGroup` en Windows.

5.10.5. Clasificaciones de seguridad

La clasificación los sistemas de computación según sus requisitos de seguridad ha sido un tema ampliamente discutido desde los años 70. Han existido múltiples clasificaciones entre las que hay que destacar TCSEC e ITSEC. En la actualidad se ha establecido a nivel internacional la clasificación Criterio Común (CC, Common Criteria), que describimos seguidamente.

Criterio común

Esta clasificación, definida conjuntamente en Estados Unidos y Canadá a partir de 1994, se denomina **Criterio Común de Seguridad** (CC, *Common Criteria*) y se ha convertido en un estándar internacional (ISO-IEC 15408). Su objetivo es asegurar que los productos de TI cumplen con estrictos requisitos de seguridad.

Los niveles de evaluación se definen dentro del contexto de los criterios de corrección. La evaluación de la corrección investiga si las funciones y mecanismos dedicados a la seguridad están implementados correctamente. La corrección se aborda desde el punto de vista de la construcción del objeto de evaluación (TOE, *Target Of Evaluation*). Un TOE puede construirse a partir de varios componentes. Algunos no contribuirán a satisfacer los objetivos de seguridad del TOE; otros sí. Estos últimos se denominan ejecutores de la seguridad (*security enforcing*). También puede haber entre los primeros algunos componentes que, sin ser ejecutores de la seguridad, deben operar correctamente para que el TOE ejecute la seguridad; éstos reciben el nombre de relevantes para la seguridad (*security relevant*). La combinación de los componentes ejecutores de la seguridad y relevantes para la seguridad se denomina a menudo la Base Informática Segura (TCB, *Trusted Computing Base*).

En su parte 3, *Security Assurance Requirements*" presenta los siete niveles de Evaluación del Nivel de Confianza (EAL 1 al EAL 7, *Evaluation Assurance Level*) que se usan para clasificar los productos. Estos niveles definen paquetes predefinidos de requisitos de seguridad aceptados internacionalmente para productos y sistemas. A continuación se describen brevemente sus propiedades.

- **EAL-1**. Incluye pruebas funcionales sobre los TOE. Desarrolladas por evaluadores externos. Existe un análisis de las funciones de seguridad usando una especificación funcional y de la interfaz y una documentación guía que define el comportamiento de seguridad.
- **EAL-2**. EAL-1 mejorado con diseño de alto nivel del TOE. Con ello se realizan pruebas funcionales y estructurales. Deben existir pruebas externas y también internas, así como poder demostrar a los probadores evidencias de los resultados de pruebas, fortaleza de los análisis de funciones y de que se han probado las vulnerabilidades obvias (por ejemplo, las de dominio público). También exige una guía de configuración y un procedimiento de distribución seguros.
- **EAL-3.** EAL-2 mejorado con entornos de desarrollo controlados y pruebas de cobertura de funciones más extensas. Exige una metodología de pruebas y comprobaciones.
- **EAL-4.** EAL-3 mejorado con especificación completa de interfaz, diseño de bajo nivel, un subconjunto de la implementación. Modelo no formal de la política de seguridad del TOE. Exige un análisis independiente de vulnerabilidad que demuestre la resistencia a ataques de penetración con bajo potencial de amenaza. Exige una metodología de diseño, pruebas y comprobaciones.
- **EAL-5.** EAL-4 mejorado con descripción formal del diseño del TOE, descripciones semiformales de funcionalidad e interfaz, implementación completa y una arquitectura estructurada. También se exige un análisis de canales encubiertos.
- **EAL-6**. EAL-5 + un análisis más riguroso, representación estructurada de la implementación y la arquitectura, análisis de vulnerabilidad más estricto, identificación sistemática de canales encubiertos, métodos mejorados de gestión de configuración y entorno de desarrollo más controlado.
- **EAL-7**. EAL-6 mejorado con métodos formales de diseño y verificación.

Por ejemplo, los sistemas operativos "Trusted Solaris 8", "Red Hat Enterprise Linux Version 6.2 on 32 bit x86 Architecture" y Microsoft "Windows Server™ 2008" tienen una certificación EAL-4.

La comprobación de cada EAL lleva aparejada una serie de clases y componentes de aseguramiento que es obligatorio cumplir para satisfacer ese nivel, lo que está fuera del ámbito de este libro.

5.11. MONTADO DE SISTEMAS DE FICHEROS

La operación de montaje proyecta la estructura jerárquica de un sistema de ficheros sobre un directorio (punto de montaje) del árbol de directorios del sistema. La figura 5.36 muestra el resultado de montar el sistema de fichero SF2 sobre el nodo /usr. Esta operación está **reservada al administrador** o superusuario.

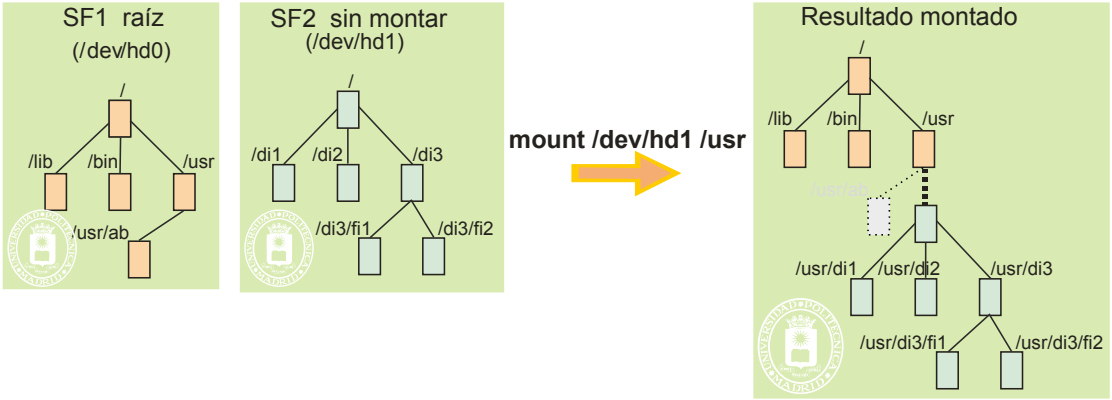


Figura 5.36 *Montaje del sistema de fichero SF2 del dispositivo hd1 sobre el nodo /usr del sistema de ficheros SF1 que se supone es el raíz del árbol de directorios del sistema.*

Pueden observarse los siguientes aspectos:

- El nombre del raíz de SF2 pasa a ser /usr, por lo que todos los nombres quedan afectados de ese prefijo.
- El objeto /usr/ab deja de ser accesible (queda oculto), pero no se borra. Si se desmonta SF2 vuelve a ser accesible.
- Al analizar una ruta que pase por el nodo de montaje, se comprueban los permisos del raíz montado, pero no los del nodo de montaje. En la figura no se analizan los permisos del usr/ original, sino los permisos del raíz del SF2. El efecto del montaje es que el raíz del sistema de ficheros montado sustituye al punto de montaje a todos los efectos. La figura 5.37 muestra otro ejemplo, con montaje y enlace simbólico.
- En la operación de montado se pueden restringir los permisos de los ficheros montados, por ejemplo, conceder solamente permiso de lectura, o que inhibir los permisos de ejecución.

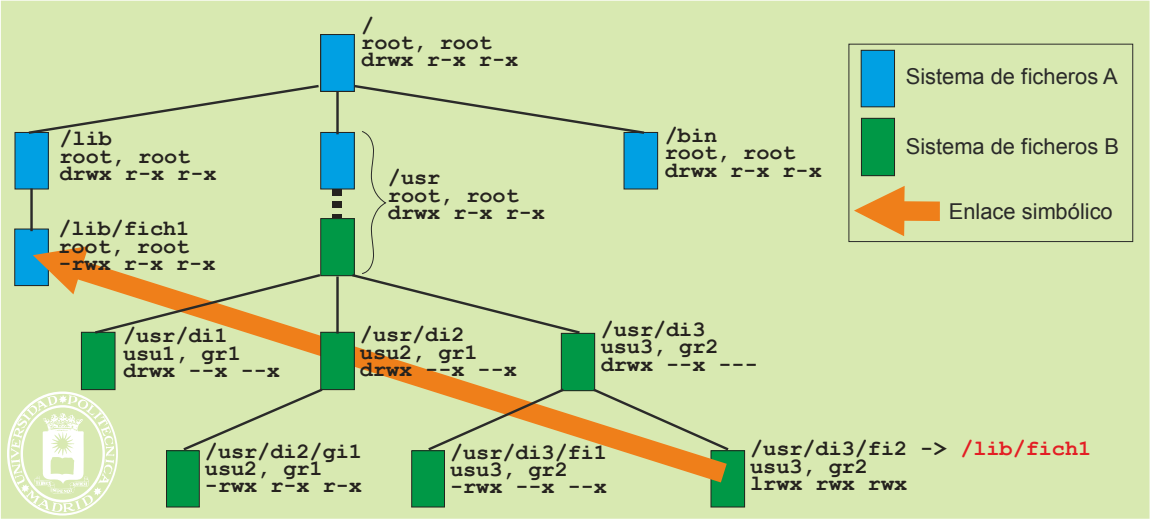


Figura 5.37 *Ejemplo de análisis de permisos. Para abrir /usr/di3/fi2 se comprueban los permisos de: "/" (raíz SF A), "usr/" (raíz SF B), "di3/", "/" (raíz SF A), "lib/", "fich1". No se comprueban los permisos almacenados en el nodo-i de "fi2" ni los del nodo-i original de "usr/".*

Un efecto importante del montaje es que los IDFF ya no son únicos, puesto que cada sistema de ficheros usa sus identificadores. Por ejemplo, los números de nodo-i del raíz de SF1 y del raíz de SF2 serán ambos igual a 2. Al quedar estos números repetidos en los distintos sistemas de ficheros, para identificar un fichero es necesario especificar tanto el IDFF como el sistema de ficheros, como muestra la figura 5.38 para el caso de UNIX.

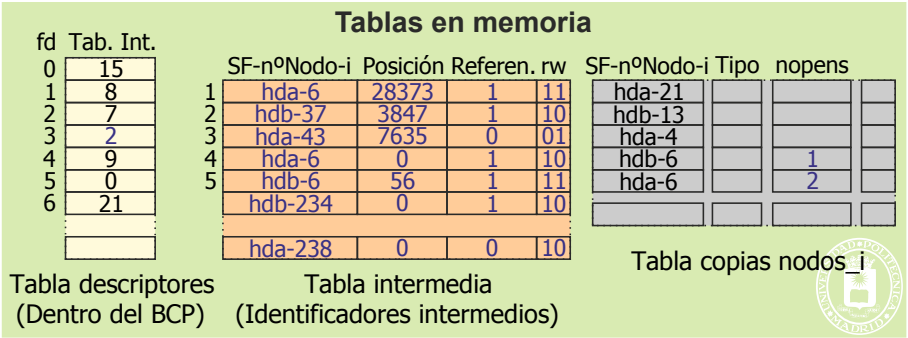


Figura 5.38 *El uso de sistemas de ficheros montados obliga a incluir el número de nodo-i más la identificación del sistema de ficheros para identificar un fichero.*

Ejemplos de este servicio son `mount` en UNIX y `Mountvol` en Windows.

Desmontado de sistema de ficheros

La operación de desmontado de ficheros desproyecta la estructura jerárquica de un sistema de ficheros montada previamente. Por tanto, los ficheros y directorios dejan de ser accesibles. Las características principales son las siguientes:

- Este servicio está restringido al administrador o superusuario.
- El servicio da error si existe algún fichero abierto en el sistema de ficheros que se desmonta.

Ejemplos de este servicio son `umount` en UNIX y `Mountvol` en Windows.

5.12. CONSIDERACIONES DE DISEÑO DEL SERVIDOR DE FICHEROS

5.12.1. Consistencia del sistema de ficheros y journaling

En un sistema de ficheros consistente los mapas de bits representan agrupaciones y DDF realmente libres, los directorios contienen IDFF válidos, cada DFF usado tiene por lo menos una referencia, las direcciones a agrupaciones contenidas en los DFF son válidos, etc. Sin embargo, un sistema de ficheros puede quedar inconsistente después de un apagado brusco, puesto que pueden ocurrir situaciones como las siguientes:

- El DFF no se actualiza con la nueva agrupación asignada al fichero, por lo que los nuevos datos no formarán parte del mismo.
- Si el tamaño del fichero no se actualiza en el DFF, los nuevos datos no formarán parte del fichero.
- Si una agrupación o DFF no se marca en el mapa de bits puede volver a asignarse, con el consiguiente problema.
- Si los datos no se escriben en la agrupación, pero se actualiza el tamaño del fichero en el DFF, se accedería a basura.

Por ello, se han desarrollado programas que comprueban si un sistema de ficheros es consistente, tratando, además, de hacer algunas reparaciones para garantizar la consistencia. El análisis de consistencia incluye los dos aspectos siguientes:

- Comprobar que el disco funciona correctamente. Para ello, se realizan operaciones de escritura y lectura consecutivas sobre el mismo bloque del disco, comparando lo escrito con lo leído. En caso de detectar un error permanente, se marca como erróneo el bloque y se utiliza uno de repuesto.
- Verificar que la estructura lógica del sistema de archivos es correcta. Esta comprobación se desglosa en los siguientes puntos:
  - Se comprueba que el contenido del superbloque responde a las características del sistema de ficheros.
  - Se comprueba que los mapas de bits de DFF se corresponden con los DFF ocupados en el sistema de ficheros.
  - Se comprueba que los mapas de bits de agrupaciones se corresponden con las agrupaciones asignadas a ficheros.
  - Se comprueba que ninguna agrupación esté asignada a más de un fichero.
  - Si no están permitidos los enlaces físicos de directorios, se comprueba que un mismo nodo-i no está asignado a más de un directorio.
- Consistencia sobre ficheros:
  - Se recorre todo el árbol de directorios y se anota el número de veces que se repite cada número de DFF. Esto es lo que denominaremos el contador real. Una vez finalizada esta cuenta, para cada DFF se compara el contador real con el número de enlaces. En caso de no ser idéntico se cambia el número de enlaces y se genera un aviso.
  - También se anota el número de veces NU que se utiliza cada agrupación y se compara con el mapa de bits. Suponiendo que un 0 en mapa de bits indica agrupación ocupada, se pueden dar los casos siguientes:

Mapa bits	NU	
0	1	Situación **correcta**
1	0	Situación **correcta**
0	0	**Error leve**. Como nadie usa la agrupación se marca como libre
1	1	**Error leve**. La agrupación se marca como ocupada.
0 o 1	n > 1	**Error grave**. Se desdobla la agrupación copiándola en agrupaciones libres, que se asignan a cada fichero afectado.

- Se comprueba que cada número de DFF del directorio es válido, comprendido entre los valores mínimo y máximo.
- Se genera un aviso de los ficheros con bits de protección 0007.
- Se genera un aviso de los ficheros con privilegios de root en directorios de usuario.

El mandato fsck de UNIX o chkdsk en Windows comprueban la consistencia del sistema de ficheros.

Un problema grave de estos programas es que deben hacer un análisis exhaustivo del sistema de ficheros, por lo que, dado los tamaños de los discos actuales, tardan mucho tiempo. Para evitar este problema los servidores de ficheros actuales recurren a la técnica del *journaling*.

5.12.2. Journaling

El *journaling* también conocido como *write-ahead logging* tiene su origen en los servidores de bases de datos. Esta técnica consiste en mantener un diario (*journal*) transaccional de cambios. El concepto de transacción se explica en la sección "6.8 Transacciones" y está basado en primitivas de Transaction-begin, Commit, Transaction-abort y Transaction-end, y en un almacenamiento permanente. En este caso, se utilizan las primitivas de Transaction-begin y Transaction-end, y se dedica una zona del disco, que llamamos diario, como almacenamiento permanente.

Cuando se produce un cambio en un fichero, se almacena en el diario el comienzo de la transacción, los cambios producidos tanto en la metainformación como en los datos y el final de la transacción. Por ejemplo:

1. Transaction-begin.
2. Nuevo nodo-i 779 [contenido del nodo-i].
3. Modificado bloque de mapa de bits de nodos-i.
4. Nuevas agrupaciones de datos [contenido de las agrupaciones de datos].
5. Modificado bloque de mapa de bits de agrupaciones.
6. Transaction-end.

Una vez que la transacción se escribe en el disco, se mandan las mismas órdenes al servidor de ficheros que realiza los cambios en el sistema de ficheros. Si todo va bien (e.g. el sistema no es apagado de forma anormal ni se extrae el dispositivo) no se necesita la transacción grabada en el disco, por lo se marca como nula. Antes de montar de nuevo el sistema de ficheros, se analiza el diario para ver si existe alguna transacción no nula. En caso positivo, se mandan las órdenes para completar el cambio en el sistema de ficheros, dado que el diario contiene toda la información necesaria para reconstruir la información. Además, el diario también indica las agrupaciones afectadas, por lo que no hay que recorrer todo el sistema de ficheros para buscar inconsistencias. Si el problema se produce antes de completar la escritura de la transacción en el diario, dicha transacción es parcial al no terminar con un Transaction-end, por lo que se ignora en la fase de reconstrucción. En este caso, se han perdido las modificaciones de dicha transacción parcial, pero el sistema de ficheros seguirá consistente.

Es importante, por consiguiente, garantizar que el fin de transacción no se escribe en el diario antes de haber escrito toda la información de la transacción, lo que podría ocurrir si el algoritmo de planificación del disco cambia el orden de las solicitudes de escritura.

El *journaling* tiene un **alto coste**, puesto que se escriben dos veces en el disco tanto los datos como la metainformación. Una optimización consiste en escribir los datos directamente en el disco, antes de grabar la transacción, eliminando los datos del diario. Se elimina el punto 4 del ejemplo anterior, de forma que los datos se escriben una sola vez. En caso de que el sistema se cierre sin completar la transacción en el diario, no se pierde la consistencia del sistema de ficheros, puesto que las agrupaciones que se han modificado no se han marcado como libres, ni el tamaño del fichero se ha cambiado. Simplemente, se perdería una parte de la modificación del fichero.

El diario suele ser una zona del disco de tamaño fijo, empleándose como un *buffer* circular.

5.12.3. Memoria cache de E/S

La memoria cache de E/S es una memoria intermedia que almacena bloques de información, con la esperanza de poder utilizarlos sin tener que acceder al correspondiente periférico. La memoria cache se puede nutrir de las dos fuentes siguientes:

- Bloques **recientemente leídos o escritos**. En muchas aplicaciones la información de un fichero se está reutilizando repetidamente, esto ocurre, por ejemplo, al editar un fichero. Si los accesos a los ficheros tienen proximidad referencial, es decir, reutilizan la información accedida, se pueden ahorrar accesos al disco, mejorando las prestaciones del sistema. Si los accesos no tiene proximidad referencial, lo que puede ser el caso de una gran base de datos con accesos dispersos, no hay reutilización de bloques, por lo que las prestaciones no mejoran.
- **Lecturas adelantadas**. Muchas veces el acceso a los ficheros es de tipo secuencial, por ejemplo, se lee completamente el fichero. El servidor de ficheros puede solicitar del disco más bloques contiguos de los que pide la aplicación, con la esperanza de que se utilicen en un futuro próximo. El ahorro, en este caso, consiste en que en los discos magnéticos hay un gran coste de tiempo en posicionarse en la ubicación de la información, pero, una vez en posición, se transfiere la información a gran velocidad.

Como muestra la figura 5.39, la memoria cache puede estar en el propio dispositivo, siendo gestionada directamente por el controlador del dispositivo, o puede consistir en una parte de la memoria principal del computador,

siendo gestionada por el sistema operativo. En los sistemas con memoria virtual la gestión de la cache de E/S está asociada a la memoria virtual, de forma que el tamaño del bloque es igual al tamaño de la página.

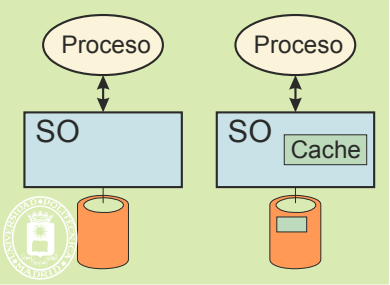


Figura 5.39 *La memoria cache de E/S puede estar en el propio dispositivo o pude consistir en una parte de la memoria principal del computador.*

El problema de utilizar una cache en memoria volátil es la pérdida de información en caso de corte de alimentación de la memoria. Por ello, se plantean distintas políticas de escritura, que analizamos a continuación:

- Política de escritura **diferida** (***write-back***). En esta política, las escrituras al almacenamiento permanente de la información modificada solamente se hacen cuando se elimina de la cache. Esta política optimiza el rendimiento, pero es arriesgada, puesto que la información modificada puede permanecer mucho tiempo en la cache, sin ser enviada al almacenamiento permanente, con el consiguiente problema de pérdida de información si se produce un corte de alimentación o la caída del sistema.
- Política de escritura **retardada** (***delayed-write***). En este caso, la información modificada también se deja en la memoria cache. Sin embargo, se garantiza que el tiempo de permanencia sin copiarse a la memoria permanente esté acotado. En este sentido, los sistemas actuales realizan, cada cierto tiempo (e.g. cada 30 segundos en UNIX), una operación de limpieza de la cache (*sync*), escribiendo todos los bloques modificados al almacenamiento permanente. De esta forma, se garantiza que solamente se puedan perder las modificaciones más recientes.
  En este caso, al igual que en el anterior, es muy importante no extraer un disco del computador sin antes volcar los datos de la cache.
- Política de escritura **inmediata** (***write-through***). En el *write-through* la información modificada se escribe inmediatamente, tanto en la cache como en el almacenamiento permanente. Es la técnica más segura con respecto a la conservación de la información, pero la menos eficiente.
- Política de escritura **al cierre** (***write-on-close***). Cuando se cierra un archivo, se vuelcan al disco los bloques del mismo que tienen datos actualizados.

Es frecuente que los servidores de ficheros utilicen una política híbrida, empleando *delayed-write* para los datos de los ficheros y *write-through* para los metadatos, ya que una pérdida de metadatos puede comprometer todo un fichero.

Flujo de datos

El flujo de datos requiere varias proyecciones para completar la correspondencia entre el *buffer* del proceso peticionario y el dispositivo. Existen dos alternativas de diseño, reflejadas en las figuras 5.40 y 5.41.

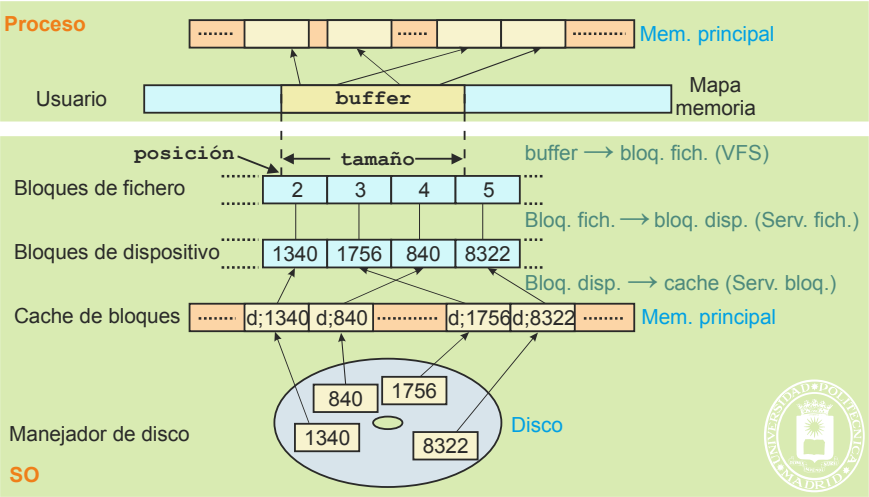


Figura 5.40 *Flujo de datos para el caso en el que el servidor de bloques realiza la proyección de bloques de dispositivo a cache.*

Analizaremos primeramente la solución de la figura 5.40, que comprende las siguientes proyecciones:

- **Buffer de usuario a memoria principal**. Esta proyección la realiza el gestor de memoria virtual, que asigna marcos de página a los espacios virtuales del proceso.
- **Buffer de usuario a bloques lógicos del fichero**. Esta proyección consiste, simplemente, en dividir el valor del puntero de posición del fichero por el tamaño del bloque, para determinar el primer bloque lógico

afectado. El tamaño de la operación de E/S permite determinar el resto de los bloques afectados (bloques 2 a 5 en la figura 5.40).

- **Bloque de fichero a bloque de dispositivo**. Esta proyección la realiza el servidor de ficheros utilizando la DFF del fichero correspondiente.
- **Bloque de dispositivo a cache**. El servidor de bloques se encarga de hacer esta proyección, por lo que tiene que tener una tabla que relaciones los bloques de dispositivos con la cache. Esto se refleja en la figura indicando cada bloque de cache tiene una etiqueta con el bloque de dispositivo.

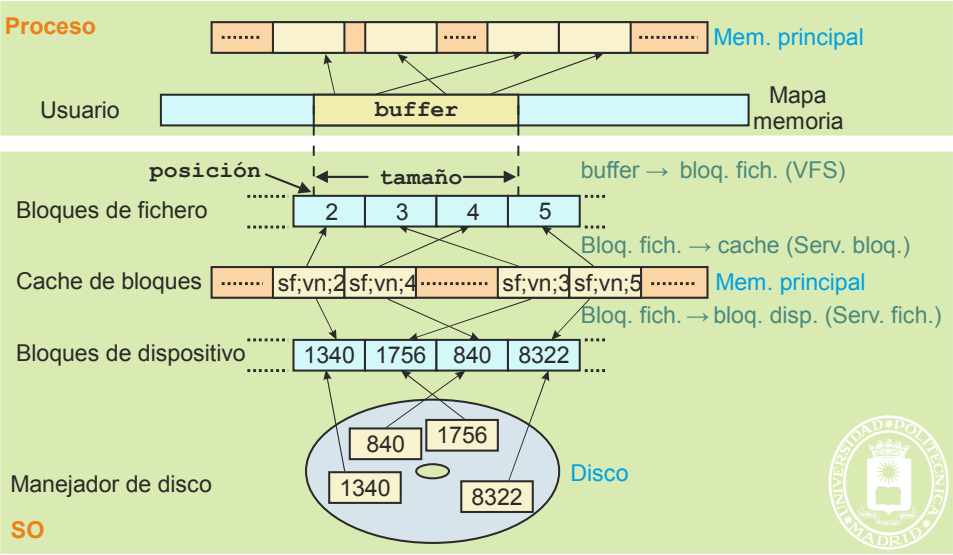


Figura 5.41 *Flujo de datos para el caso en el que el servidor de bloques realiza la proyección de bloques de fichero a cache.*

Para el caso de la figura5.41, las dos primeras proyecciones son las mismas que en el caso anterior, por lo que solamente consideraremos las dos últimas:

- **Proyección de bloque lógico de fichero a cache**. Esta proyección la realiza el servidor de bloques, para lo que debe tener una tabla que relaciones los bloques de lógicos de fichero con la cache. Esto se refleja en la figura indicando que cada bloque de cache tiene una etiqueta con el bloque lógico del fichero, el sistema de ficheros y el número de fichero.
- **Bloque de fichero a bloque de dispositivo**. Esta proyección la realiza el servidor de ficheros utilizando la DFF del fichero correspondiente.

5.12.4. Servidor de ficheros virtual

Un servidor de ficheros virtual es una capa de *software* que se pone por encima de los servidores de ficheros específicos, ofreciendo al usuario unas primitivas de acceso comunes y uniformes a todos ellos. Los sistemas operativos tipo UNIX suelen incluir un servidor de ficheros virtual.

La figura 5.42 muestra cómo el servidor de ficheros virtual encapsula servidores de ficheros específicos tales como ext2, ext4, vfat, NFS, proc, etc., ofreciendo operaciones comunes a todos ellos, tanto sobre ficheros y directorios como sobre sistemas de ficheros completos.

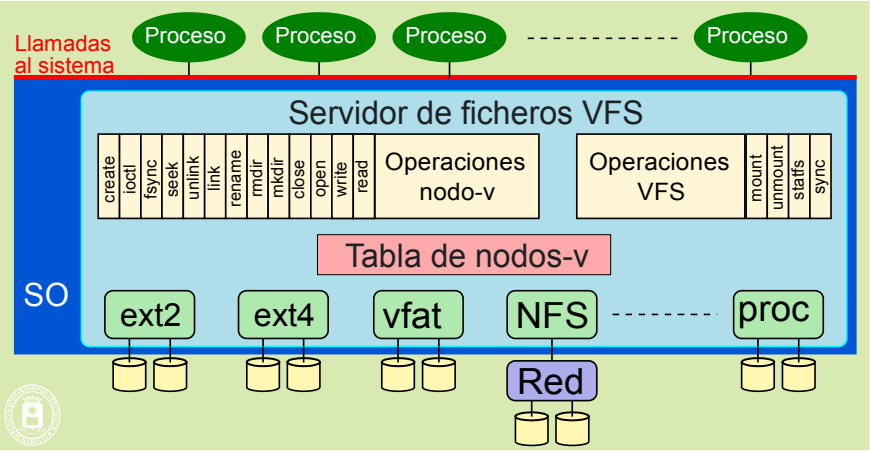


Figura 5.42 *El servidor de ficheros virtual ofrece al usuario una visión uniforme de todos los sistemas de ficheros instalados en el computador.*

El VFS define un modelo de fichero capaz de representar las características y comportamiento de los ficheros de cualquier sistema de ficheros. Considera que los ficheros son objetos ubicados en un dispositivo de almacenamiento secundario que comparten una serie de propiedades con independencia del sistema de ficheros concreto con-

siderado así como del *hardware* disponible. Los ficheros tienen nombres simbólicos que permite identificarlos sin ambigüedades. Un fichero tiene un dueño u protección frente a usos no autorizados, pudiendo ser creado, leído, escrito y eliminado. Para cada tipo de sistema de ficheros es necesario crear una proyección que transforme las características concretas de sus ficheros con las características que espera el servidor de ficheros virtual.

La figura 5.43 muestra el funcionamiento del VFS. Cuando un proceso solicita un servicio relativo a un fichero, el sistema operativo ejecuta la función del VFS relativa a ese servicio. Dicha función realiza las manipulaciones independientes de sistema de ficheros y llama a una función del servidor de ficheros objetivo X. Esta llamada pasa a través de la función de proyección que convierte la función del VFS en la correspondiente del servidor de ficheros objetivo X. El servidor de ficheros X convierte la llamada en peticiones para el dispositivo afectado, peticiones que se envían al correspondiente manejador.

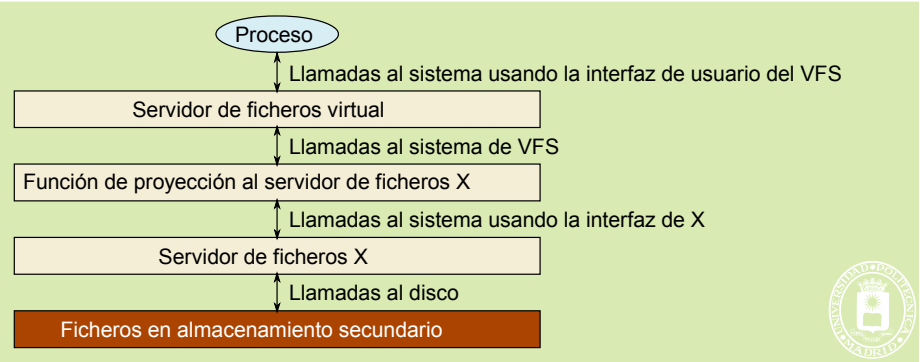


Figura 5.43 *Concepto del servidor de ficheros virtual VFS.*

Nodo-v

VFS mantiene en memoria una estructura de información llamada nodo-v por cada fichero abierto. Dicha estructura es común a todos los sistemas de ficheros subyacentes y enlaza con un descriptor de archivo de cada tipo particular, por ejemplo, con un nodo-i para el caso de un sistema de ficheros ext2. Cuando se abre un fichero se mira si ya existe un nodo-v de ese fichero, en cuyo caso se incrementa su contador de uso. En caso contrario, se crea un nuevo nodo-v.

La figura 5.44 muestra al nodo-v que se organiza en los siguientes campos:

- El nodo-v mantiene **información** de **gestión** relativa al fichero, tal como: Tipo de fichero (regular, directorio, dispositivo de bloques, dispositivo de caracteres o enlace), semáforos, contadores y colas. También incluye un contador de uso.
- Direcciones de las **operaciones virtuales**, tanto de fichero como de directorio.
- Dirección del **nodo-i** específico del fichero.
- Dirección de las **operaciones específicas** del servidor de ficheros correspondiente.

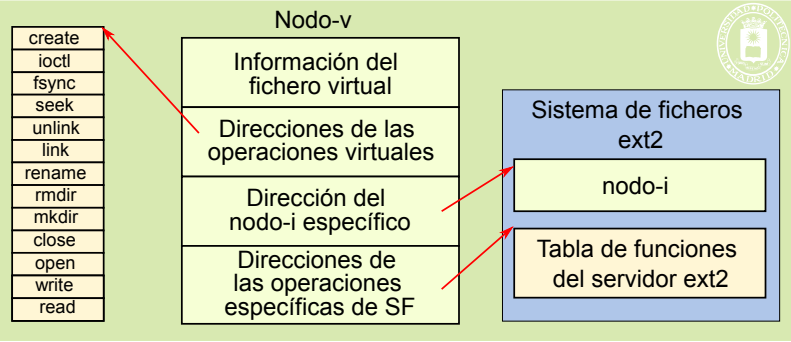


Figura 5.44 *El nodo-v mantiene una referencia a las operaciones virtuales, al nodo-i del fichero y a las funciones del servidor de ficheros afectado.*

Registro de un nuevo servidor de archivos

Antes de poder utilizar un sistema de ficheros de un cierto tipo, hay que dar de alta el correspondiente servidor de ficheros, por ejemplo, utilizando la función:

```
register_filesystem(struct file_system_type*);
```

Con ello, el servidor de ficheros queda añadido a la lista encadenada de servidores disponibles. La figura 5.45 muestra esta lista, así como algunos de los campos almacenados para cada servidor, tales como:

- Función utilizada para obtener el superbloque del sistema de ficheros.
- Nombre del tipo de servidor de ficheros.
- Opción de si requiere dispositivo. Por ejemplo, “proc” no requiere dispositivo.
- Puntero al siguiente elemento de la lista.

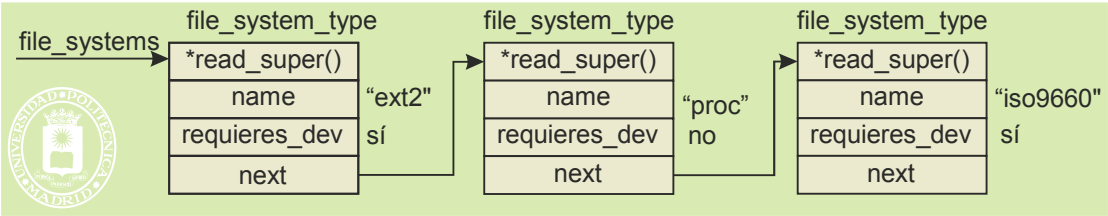


Figura 5.45 *Lista encadenada de los servidores de ficheros dados de alta en un sistema.*

Montado de un sistema de ficheros

La operación de montado incluye los tres argumentos siguientes:

- El tipo de sistema de ficheros, por ejemplo: `ext2`. Para que se pueda hacer el montaje, es necesario que el servidor de ficheros de ese tipo esté registrado, tal y como se vio en la sección anterior.
- El sistema de ficheros a montar, por ejemplo: `/dev/sda`.
- El punto de montaje, por ejemplo: `/usr`.

La figura 5.46 muestra las estructuras de información requeridas en el montaje.

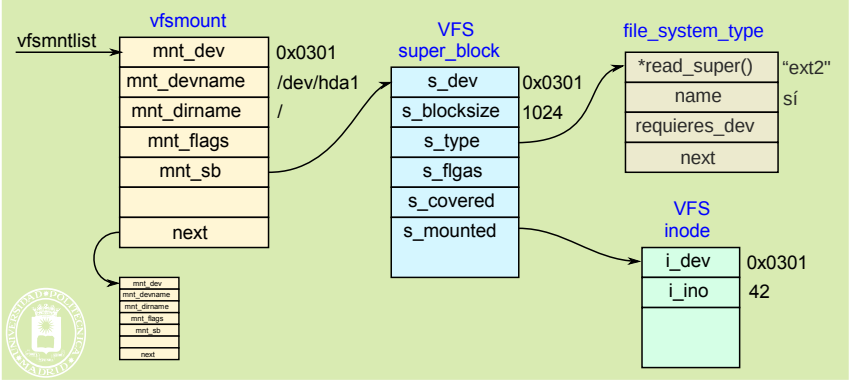


Figura 5.46 *Estructuras de información involucradas en el montaje de un sistema de ficheros.*

Lo primero es incluir la estructura `vfsmount` en la lista encadenada de sistemas de ficheros montados. Esta estructura permite identificar el superbloque VFS del sistema de ficheros. Dicho superbloque, a su vez, apunta a un `file_system_type`, que ha de haber sido creado al registrar el correspondiente servidor de ficheros. Además, apunta a un nodo-v que corresponde con el raíz del sistema de ficheros montado y que se conserva permanentemente en memoria.

5.12.5. Ficheros contiguos ISO-9660

El sistema de ficheros ISO-9660 está diseñado para dispositivos de almacenamiento de una sola escritura tales como los CD o los DVD. Por ello, se efectúa almacenamiento contiguo de ficheros hasta que se termina la sesión de escritura. En ese momento, se pueden hacer dos cosas: cerrar el volumen y cerrar la sesión. En el primer caso, se pone una marca al final de los datos y ya no se puede escribir más en el dispositivo. En el segundo caso, se pone una marca de fin de sesión y posteriormente se puede grabar otras sesiones hasta que se cierre el volumen.

El disco se divide en sectores 2 KiB, que se organizan en *extents* formados por un número entero de sectores consecutivos. Las direcciones de los sectores son de 32 bits, por lo que el tamaño máximo del sistema de ficheros es de $2^{32} \cdot 2$ KiB= 8 TiB. Curiosamente, los valores multi-byte se almacenan generalmente dos veces: una en formato *little-endian*[1] y otra en *big-endian*.

La estructura del sistema de ficheros se muestra en la figura 5.47. Sus campos son los siguientes:

- Los primeros 16 sectores del volumen se dejan libres, pero se usan en las versiones Rock-Ridge y Joliet
- Zona de descriptores, donde cada descriptor ocupa un sector. Los descriptores pueden ser los siguientes:
  - Primario: Este descriptor incluye información de identificación del volumen y sus contenidos, la estructura del volumen y el sector del directorio raíz.
  - Secundario. Opcional.
  - Particiones. Opcional.
  - Boot. Opcional.
  - Terminación. Simplemente indica que termina la zona de descriptores.

1 En el formato *little-endian* el byte menos significativo del dato, por ejemplo de un entero, ocupa la dirección de memoria de menor valor. Por el contrario, en el formato *big-endian* el byte menos significativo ocupa la dirección de memoria de mayor valor.

- Tabla de caminos. Esta tabla representa el árbol de directorios, pero sin incluir los nombres de los ficheros regulares. Es redundante, puesto que reproduce parte de la información de los directorios. Está almacenada de forma contigua en un *extent*, por lo que la búsqueda de un fichero se hace mucho más rápidamente que si se hay que leer todos los directorios involucrados, que están dispersos por el volumen.
- Tanto los directorios como los ficheros se almacenan como *extents*. Cada directorio incluye el «.» y «..».

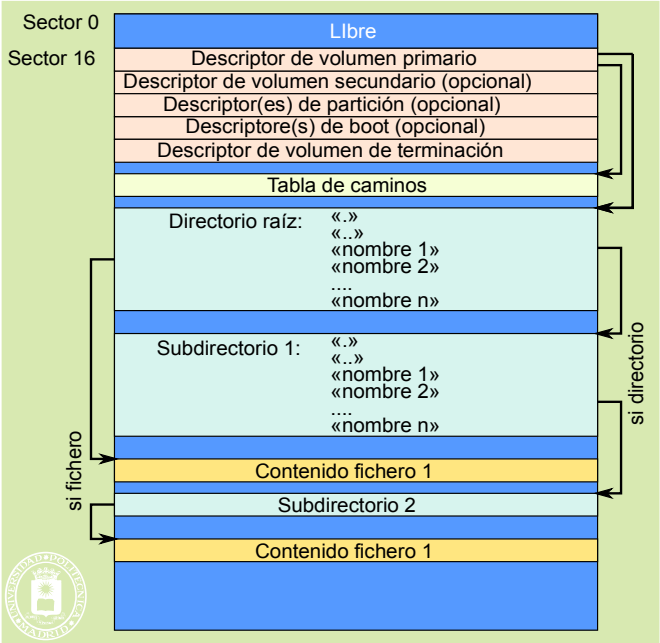


Figura 5.47 *La estructura de un sistema de ficheros ISO-9660 deja libres los 16 primeros sectores del disco. Seguidamente se encuentra el campo de descriptores, cada uno de los cuales ocupa un sector. Algunos descriptores son opcionales.*

La figura 5.48 muestra una entrada de directorio. Se puede observar que el tamaño total es de 256 B, siendo el campo de nombre de tamaño variable. Otros campos importantes son los siguientes:

- Ubicación del fichero. Indica el sector en el que empieza el fichero. Los demás sectores son consecutivos.
- Tamaño del fichero. Se dispone de un campo de 8 B, que almacena el tamaño en ambos formatos *little-endian* y *big-endian*, por lo que se tienen 32 bits útiles. El tamaño máximo es, por tanto, menor que $2^{32} = 4$ GiB, sin embargo se puede usar la funcionalidad de fragmentación que permite crear ficheros multi-*extent* con un tamaño de hasta 8 TiB.

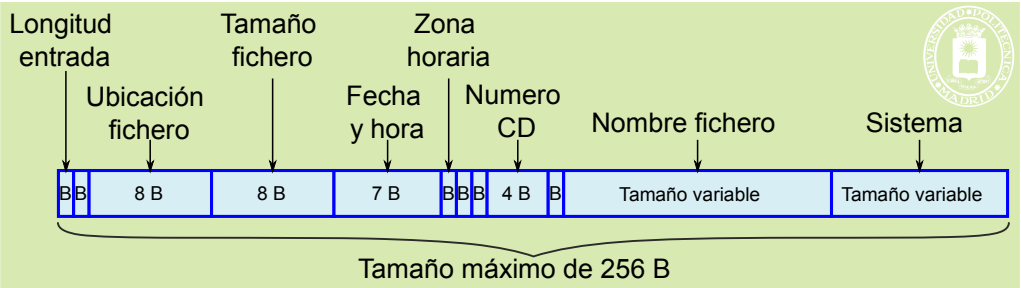


Figura 5.48 *Entrada de directorio del sistema de ficheros ISO-9660.*

5.12.6. Ficheros enlazados FAT

Como ejemplo de sistemas de ficheros, en los que la estructura física de un fichero se representa mediante una lista enlazada de direcciones de agrupaciones, consideraremos el sistema FAT. Este sistema ha sufrido a lo largo de los años grandes modificaciones, pasando de la FAT12 (año 1977) a la FAT16 (año 1984) y al a FAT32 (año 1996). Esta evolución es debida al aumento de los discos y a la falta de previsión de los primeros diseños, ya que las FAT12 y FAT16 solamente soportan respectivamente 4 y 64 Ki_agrupaciones, mientras que FAT32 soporta $2^{28} = 256$ Mi_agrupaciones.

Para la FAT32, el tamaño máximo del volumen viene determinado por el tamaño de la partición (tabla de particiones en el boot), que se almacena en una palabra de 4B, por lo que es: 2^{32}x512 B = 2 TiB.

La figura 5.49 muestra la estructura de los sistemas de ficheros FAT16 y FAT32. Es de destacar que la tabla FAT está duplicada por seguridad, puesto que si se deteriora se pierde toda la información del volumen. En la FART16 la posición del directorio raíz es fija y su tamaño, también fijo, es de 224 entradas.

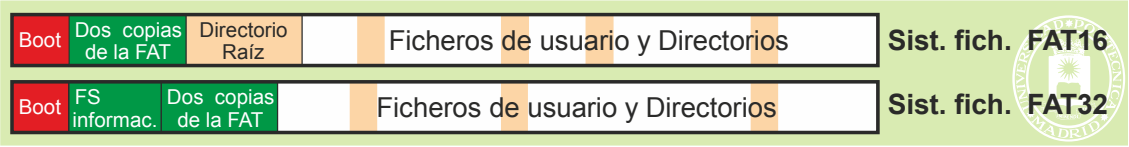


Figura 5.49 *Estructura de los sistemas de ficheros FAT16 y FAT32.*

La metainformación de este sistema se organiza en las tablas directorio y la tabla FAT (*File Allocation Table*), que se muestran en la figura 5.50. La estructura de los directorios se conserva desde el diseño de la FAT 12, con algunas modificaciones, estando basada en entradas de tamaño fijo de 32 B. El nombre viene limitado a 8 B más tres de extensión, mientras que los atributos son solamente los seis bits siguientes: ***R**ead-only*, ***H**idden*, ***S**ystem*, ***V**olume label*, ***D**irectory*, y ***A**rchive next backed*.

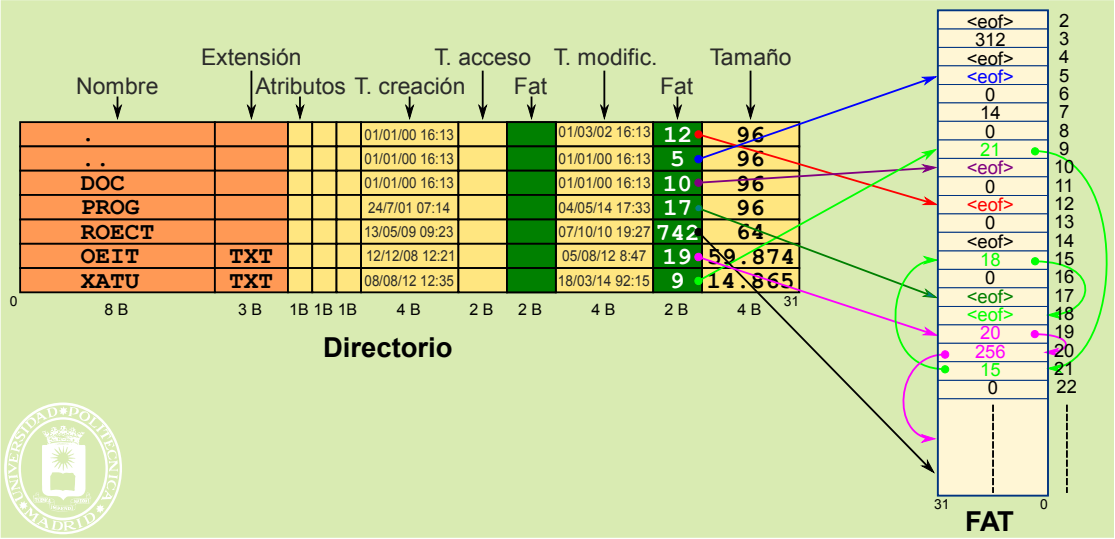


Figura 5.50 *La metainformación de los sistemas de ficheros FAT se organiza en las tablas directorio y la tabla FAT. En la figura se representa el caso de la FAT32. Se puede observar que, por compatibilidad con versiones anteriores, en el directorio la dirección de la primera agrupación, está dividida en dos campos de 2 B.*

El directorio incluye la dirección de la primera agrupación del fichero o directorio. El resto de las agrupaciones se encuentran en la tabla FAT, en forma de lista enlazada. Cada lista se cierra con un eof (final de fichero) cuyo valor está reflejado en la tabla 5.2. En el diseño original de las entradas de directorio se reservaron 2 B para este campo, por lo que solamente se podía empezar un fichero con las agrupaciones de la 2 a la $2^{16} - 1 = 65.535$. Posteriormente, para la FAT32, como se aprecia en la figura 5.50, se añadieron otros dos bytes para llegar hasta un tamaño de fichero de $2^{32} - 1$ B.

Tabla 5.2 *Valores de las direcciones de agrupaciones*

FAT12	FAT16	FAT32	Descripción
0x000	0x0000	0x00000000	Agrupación libre
0x001	0x0001	0x00000001	Reservado, no se usa
0x002–0xFEF	0x0002–0xFFEF	0x00000002–0x0FFFFFEF	Agrupación utilizada; el valor apunta a la siguiente agrupación
0xFF0–0xFF6	0xFFF0–0xFFF6	0x0FFFFFF0–0x0FFFFFF6	Valor reservado, no se usa
0xFF7	0xFFF7	0x0FFFFFF7	Agrupación con sector defectuoso o agrupación reservada
0xFF8–0xFFF	0xFFF8–0xFFFF	0x0FFFFFF8–0x0FFFFFFF	Última agrupación del fichero <eof>

El primer carácter del nombre tiene los siguientes significados:

- 0x00 entrada de directorio libre.
- 0x2E Entrada punto, se usa para los directorios «.» y «..».
- 0xE5 entrada eliminada, pero no borrada. Se puede recuperar el fichero, si sus agrupaciones no han sido asignadas.
- 0x05 primer carácter es E5. Cuando el primer carácter debe ser E5, pero no para indicar que es una entrada eliminada, se utiliza 05.

Nombres largos VFAT

El diseño original del sistema de ficheros FAT utiliza nombres de tamaño fijo, con ocho bytes para el nombre y tres para la extensión, es lo que se denomina nombre "8.3". Esto se ha considerado muy restrictivo, por lo que se introdujo la modificación VFAT, que permite utilizar nombres con hasta 255 caracteres Unicode de 16 bits.

Par ello, además de la entrada de directorio clásica vista en la figura 5.50, que denominaremos ED, se añade otra entrada en el directorio llamada LNF (*Long File Name*), como se puede apreciar en la figura 5.51. Dicha figura presenta un directorio con las seis entradas siguientes:

«.»
«..»
CORTO.PNP
Un fichero nombre largo
CORTO2.DAT
Un fichero con un nombre muy largo.txt

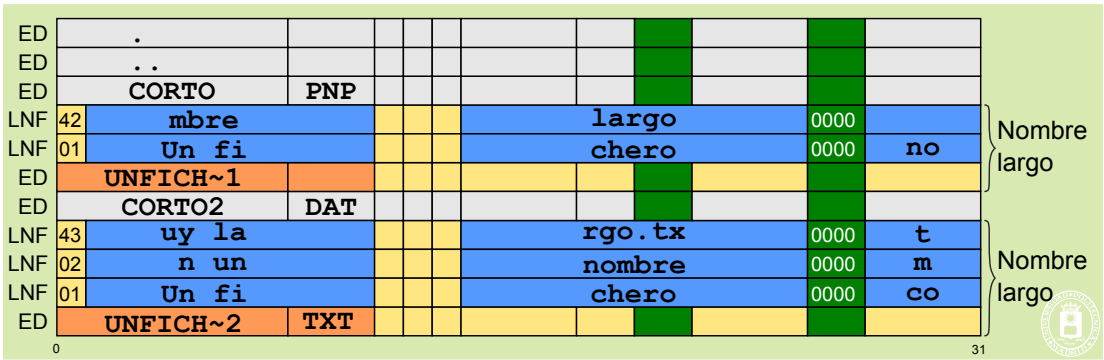


Figura 5.51 *Ejemplo de directorio VFAT con 6 entradas, 4 cortas «.», «..», 'CORTO.PNP' y 'CORTO2.DAT', y dos largas.*

Un nombre largo incluye una entrada ED de nombre 8.3, en el que el nombre se obtiene con los 6 primeros caracteres del nombre largo (eliminando caracteres no permitidos en el nombre corto como los espacios), seguido del carácter «~» y de un dígito numérico. Dicho dígito sirve para diferenciar varios nombres largos que empiecen con los mismos caracteres. Como se puede apreciar en la figura 5.51, las líneas LNF de un nombre preceden a la entrada ED.

Las entradas LNF ocupan 32 B, como las ED, y permiten almacenar hasta 13 caracteres Unicode, por lo que un nombre de 255 caracteres necesita 20 entradas LNF. La estructura de LNF se encuentra en la tabla 5.3.

Tabla 5.3: *Campos de la entrada LNF.*

Bytes	Descripción
0	Primer carácter que sirve para indicar la secuencia de registros LNF que componen un nombre largo. En el último LNF este valor se calcula partiendo del número de secuencia y haciendo una operación OR 0x40.
1 a 10	5 caracteres Unicode del nombre
11	Atributos del fichero
12	Reservado
13	Checksum
14 a 25	6 caracteres Unicode del nombre
26 a 27	Reservado, puesto al valor 0x0000
28 a 31	2 caracteres Unicode del nombre

Al final de los dos últimos caracteres del nombre se añade un 0x0000. Los demás caracteres del nombre no utilizados se rellenan con 0xFFFF.

5.12.7. Sistemas de ficheros UNIX

UNIX System V

En el año 1983 se comercializó el UNIX System V. Su sistema de ficheros tenía las siguientes características:

- Entradas de **directorio de tamaño fijo**, como se puede apreciar en la figura 5.52, con nombres de 14 caracteres.
- Agrupaciones de 512 B o 1 KiB.
- Nodo-i de 64 B con 10 números de agrupaciones más un indirecto simple, un indirecto doble y un indirecto triple. El esquema de utilización de los indirectos es la mostrada en la figura 5.20, página 225.
- Direcciones de agrupación de 24 B.

- La distribución del sistema de ficheros sigue el modelo "a) Lineal" de la figura 5.53. En este modelo primero está el boot, seguido del superbloque, los mapas de bits y los nodos-i. Finalmente, se encuentran todas las agrupaciones.
- La **asignación** de nodo-i y de agrupación libre es **totalmente aleatoria**. Esto conlleva una gran dispersión de los componentes de un fichero lo que produce grandes movimientos del brazo del disco, repercutiendo negativamente en las prestaciones del sistema (solamente se aprovecha un 2-5% del ancho de banda del disco).

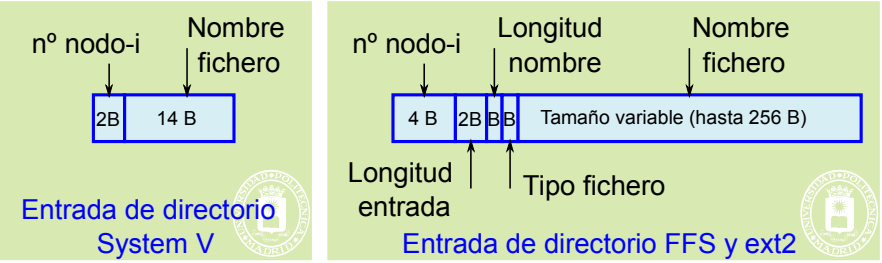


Figura 5.52 *Ejemplos de entradas de directorio para el sistema System V y para el FFS y el ext2.*

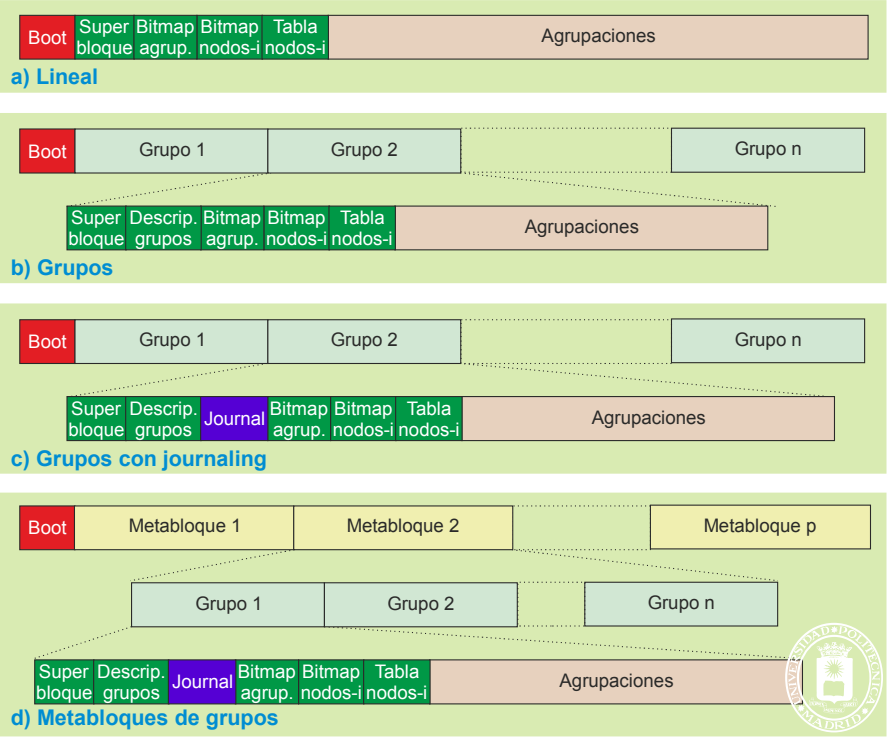


Figura 5.53 *Diversas estructuras de los sistemas de ficheros UNIX.*

Berkeley FFS

El sistema de ficheros Berkeley FFS (*Fast File System*) introduce las siguientes mejoras sobre el System V:

- Permite **nombres de 256 caracteres**, con el formato de la figura 5.52. El campo de tipo de fichero es redundante con el nodo-i. pero se añadió para reducir los accesos al disco al listar directorios.
- Soporta **agrupaciones de 4 KiB** u **8 KiB**. Como tiene un tamaño de agrupación grande, incluye un **mecanismo de fragmentos** para ficheros pequeños y la última agrupación de los grandes. De esta forma, varios ficheros pueden compartir una agrupación, reduciendo así las pérdidas de disco por fragmentación interna.
- La distribución del sistema de ficheros sigue el modelo "b) Grupos" de la figura 5.53. En este caso, se divide en volumen en grupos de cilindros contiguos y se incluyen los mapas de bits y nodos-i en cada grupo. El superbloque se duplica en cada grupo, lo que **mejora la tolerancia a fallos**.
- **Asignación de recursos con proximidad**. La asignación de nodos-i y agrupaciones libres se hace de forma que queden próximos. Así, se intenta que un directorio quede totalmente en un grupo. Cuando se crea un nuevo directorio se lleva a otro grupo, de forma que se repartan los directorios por los grupos uniformemente. De esta forma, se aumentan sustancialmente las prestaciones del sistema (se aprovecha un 14-47% del ancho de banda del disco).
- El nodo-i incluye 13 números de agrupaciones más un indirecto simple, un indirecto doble y un indirecto triple.
- Introduce el concepto de **enlace simbólico**.
- Cache. Utiliza la técnica *delayed-write* para los datos y de *write-through* para los metadatos.

Linux ext2

El sistema de ficheros ext2 (año 1992) es muy similar al FFS, con las siguientes diferencias.

- Permite un tamaño de agrupación de 1 KiB hasta 8 KiB.
- No soporta fragmentos, pero permite agrupaciones más pequeñas que el FFS.
- Su nodo-i (véase figura 5.20, página 225) tiene 12 directos en vez de 13 del FFS.
- Utiliza una política de cache muy agresiva, puesto que emplea *delayed-write* para los datos y metadatos.

Aunque han aparecido versiones posteriores como el ext3 y ext4, sigue siendo una buena alternativa para memorias flash y USB porque no incurre en la sobrecarga que supone el *journaling*.

Linux ext3

El sistema de ficheros ext3 (año 2001) es muy similar al ext2, pero añadiendo *journaling* para aumentar la fiabilidad y reducir el tiempo empleado en reparar la consistencia del sistema de ficheros. La distribución del sistema de ficheros sigue el modelo "c) Grupos con journaling" de la figura 5.53, en la que se observa el diario (*journal*) empleado como un *buffer* circular. Presenta los tres modos de funcionamiento del diario:

- **Journal**: en este modo se copian en el diario los datos y metadatos antes de llevarlos al sistema de ficheros. Es el más fiable, pero el más lento.
- **Ordered**: en este modo se escriben primero los datos en el sistema de ficheros y después se graba en el diario la transacción de los metadatos. Es más rápido y garantiza la consistencia del sistema de ficheros, pero en algunos casos genera ficheros parcialmente modificados.
- **Writeback**: igual al anterior, pero sin garantizar que el diario con los metadatos se escriba después de haber completado la escritura de los datos. Es la opción más rápida, pero la más débil, puesto que algún fichero puede quedar corrupto en caso de fallo. Sin embargo, no produce más corrupción que si no se utiliza *journaling* y permite una recuperación muy rápida de la consistencia.

Linux ext4

El sistema de ficheros ext4 (año 2006) incluye una serie de mejoras sobre el ext3, entre las que se pueden destacar las siguientes:

- Direcciones físicas de agrupaciones de 48 bits y direcciones lógicas de 32 bits.
- Soporta volúmenes de hasta 1 EiB y ficheros de hasta 16 TiB.
- Se utilizan *extents* en vez de lista de agrupaciones para describir la estructura física del fichero.
- Incrementa la resolución de las marcas de tiempo hasta los nanosegundos.
- Diarios con *checksums*.
- Soporte para ficheros ralos (poco poblados).

extent

Un *extent* es un conjunto contiguo de bloques, lo que es interesante para grandes ficheros con agrupaciones contiguas. La figura 5.54 muestra la implementación de los *extents*.

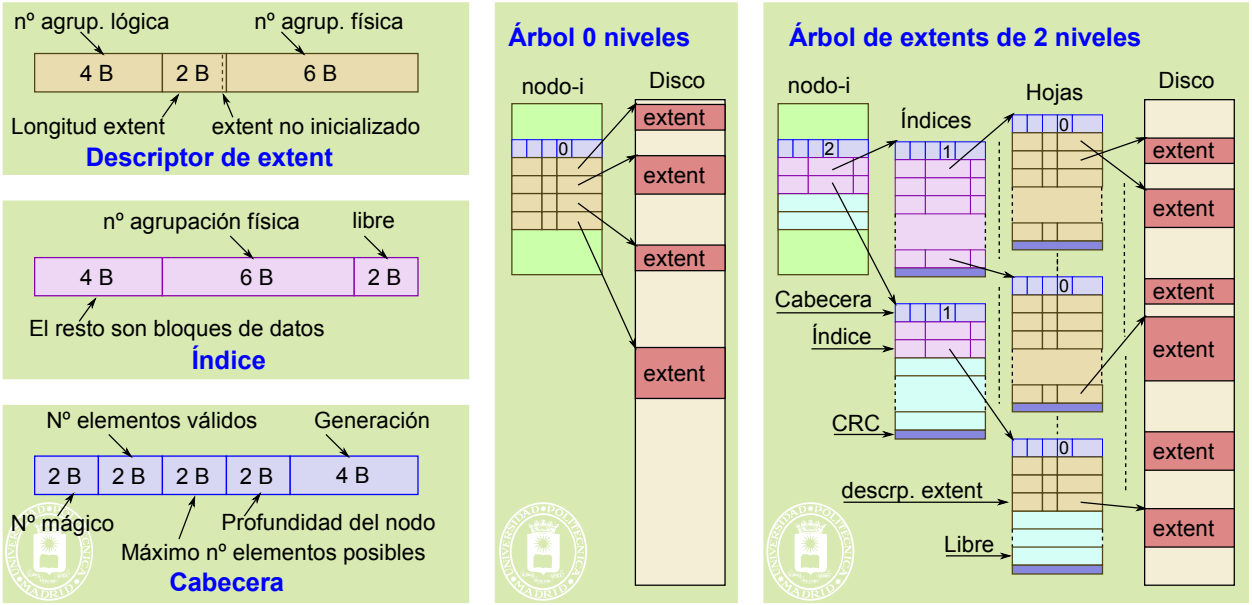


Figura 5.54 *Metainformación asociada a los extents.*

El nodo-i es de tamaño variable teniendo un valor por defecto de 256 B en vez de los 128 del ext2/3. Adicionalmente, en vez de contener las 12 directos y los 3 de indirectos, utiliza este bloque de 60 B, que se denomina i_block, para almacenar 5 elementos de *extent* de 12 B. Además de este i_block, se pueden asignar agrupaciones para contener más elementos de *extent*.

Existen tres tipos de elementos de *extent*: cabecera, índice y descriptor de *extent*. Tanto el i_block como las agrupaciones empiezan por una cabecera y pueden contener índices o descriptores de *extent*, pero no una mezcla. Por seguridad, al final de las agrupaciones se incluye un CRC.

- La **cabecera** indica el número de elementos válidos presentes en el i_block o en la agrupación. También almacena la profundidad del nodo. Esta profundidad puede llegar hasta 5, y su valor 0 indica descriptores de *extent*. La figura 5.54 muestra un ejemplo de árbol de profundidad 0 y otro de profundidad 2.Por su lado, el campo generación contiene la versión del sistema.
- El **índice** permite identificar una agrupación, que puede contener índices o descriptores de *extents*.
- El **descriptor de** *extent* contiene los siguientes cuatro campos:
  - Número de agrupación lógica. Campo de 4 B que expresa la dirección de agrupación lógica con la que comienza el *extent*. Se entiende que el fichero está lógicamente formado por n agrupaciones, de direcciones 0 hasta n-1. El fichero puede, por tanto, tener un máximo de 2^{32} agrupaciones.
  - Número de agrupación física. Campo de 6 B que expresa la dirección de agrupación física con la que comienza el *extent*. El volumen puede, por tanto, tener un máximo de 2^{48} agrupaciones.
  - Longitud. Este campo contiene el número de agrupaciones contiguas que forman el *extent*. Tiene un tamaño de 15 bits, por lo que un *extent* puede tener hasta 2^{15} agrupaciones.
  - Bit de activo. Expresa si la entrada está en uso o no.

5.12.8. NTFS

El sistema de ficheros NTFS remonta su origen al Windows NT 3.1 introducido en el año 1993. Desde entonces, ha sufrido importantes mejoras, hasta la versión v3.1, lanzado junto con Windows XP en el año 2001. Sus principales características son las siguientes:

- Teóricamente, permite volúmenes de hasta 2^{64} -1 agrupaciones.
- Agrupaciones de hasta 64 KiB, siendo 4 KiB el valor más corriente.
- Utiliza una estructuración homogénea tanto para la metainformación como para los ficheros de usuario. Todo son ficheros.
- Incluye mecanismos de compresión y de cifrado de ficheros.
- Utiliza *journaling*, que se puede desactivar en dispositivos que no sean de sistema.
- Permite modificar el tamaño de las particiones de un disco sin perder la información.
- Soporta ficheros ralos (poco poblados).
- El almacenamiento en el disco es de tipo *little-endian*.
- Los ficheros se organizan en *extents*, que en NTFS se llaman *runs*.

El sistema de ficheros NTFS emplea un esquema sencillo pero potente para organizar la información del volumen. La figura 5.55 muestra el formato del volumen NTFS. Dicho formato comprende, además de boot, la tabla MFT (*Master file table*), los ficheros de sistema y la zona para ficheros de usuario y directorios.

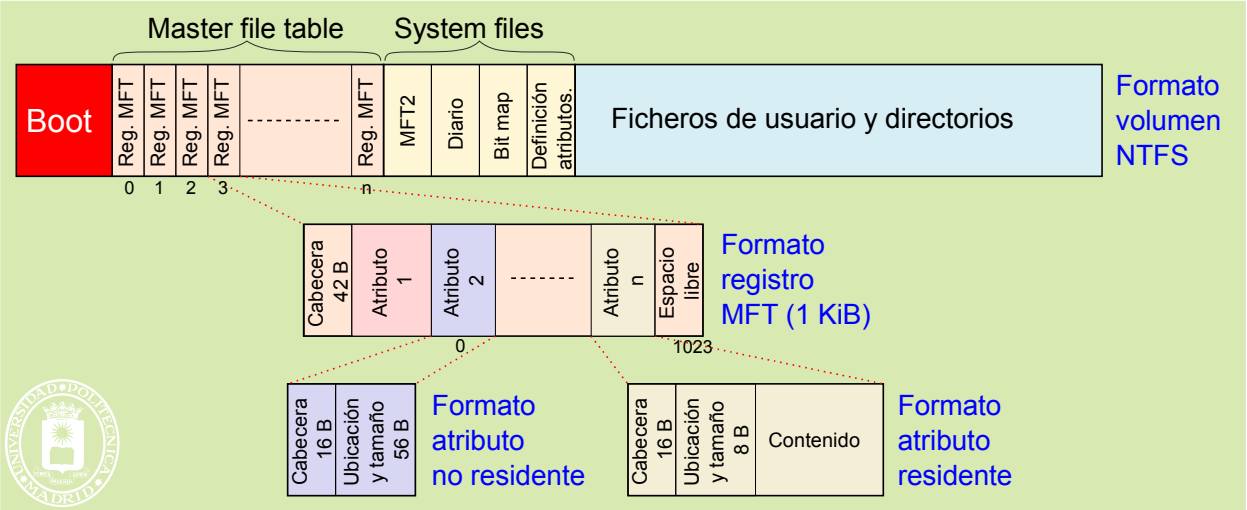


Figura 5.55 *Formatos del volumen NTFS, del registro MFT y de sus atributos.*

El **boot** puede ocupar hasta 16 sectores e incluye, entre otras cosas, el tamaño de la agrupación y las direcciones de comienzo de la tabla MFT y de la tabla MFT2.

La **tabla MFT** está compuesta por registros MFT con las siguientes características:

- El tamaño de cada registro MFT es de 1 KiB.
- Los registros se numeran empezando por el 0.
- Los registros del 0 al 15 están destinados a ficheros de sistema, por ejemplo:

- El registro MFT 5 especifica el directorio raíz.
- El registro MFT 8 contiene las agrupaciones defectuosas.

- Cada registro MFT tiene una cabecera seguida de un número variable de atributos, pudiendo quedar espacio libre al final del registro.
- Típicamente, se asigna un 12.5% del espacio del volumen para la tabla MFT.
- Si un registro tiene muchos atributos se le puede asignar más de una entrada MFT.
- La cabecera incluye un **número de secuencia** que permite saber cuántas veces se ha reutilizado el registro MFT.
- También incluye un indicador de registro usado/libre y de registro de directorio.

Los **atributos** pueden ser **residentes** o **no residentes**. Los atributos residentes incluyen todo su contenido. Por ejemplo, el atributo de nombre de fichero es un atributo residente. Por su lado, los atributos no residentes incluyen el identificador del *extent* en el que está almacenado su contenido.

Como se puede apreciar en la figura 5.57, un fichero de pequeño tamaño (hasta ~900B) cabe totalmente en el registro MFT, por lo que se obtiene con un solo acceso a disco, frente a los dos que se requieren en un sistema de ficheros tipo UNIX. Esto es válido para todo tipo de ficheros, tanto regulares como de directorio y sistema.

Los **ficheros de sistema** incluyen metainformación adicional, que se almacena en forma de ficheros, de acuerdo al mismo esquema que el resto de los ficheros. Vemos cada uno de ellos:

- La tabla **MFT2** duplica las cuatro primeras entradas de la tabla MFT, para garantizar el acceso a la MFT en el caso de fallo simple de un sector del disco.
- El **diario**, necesario para almacenar las transacciones de la función de *journaling*.
- El **mapa de bits** que especifica las agrupaciones libres.
- La tabla de definición de los tipos de atributos soportados por el sistema de ficheros y sus características.

Un fichero se identifica por el número de registro MFT (numerados desde el 0 hasta el n), para lo que se dispone de una palabra de 6 B, más el número de secuencia de 2 B incluido en la cabecera de dicho registro. De esta forma, la identificación es un número de 8 B obtenido concatenando el número de secuencia seguido del número de registro MFT. La inclusión del número de secuencia permite una mejor recuperación frente a errores.

extent

Los *extents* o *runs* están formados por un número entero de agrupaciones. Para definir el espacio ocupado en el disco por un fichero se utiliza una cadena de *extents*, como se puede apreciar en la figura 5.56. Dichas cadenas han de terminar con un descriptor nulo de *extent*.

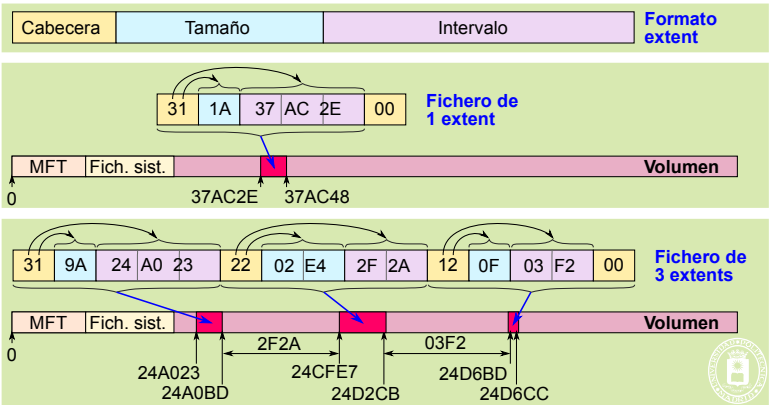


Figura 5.56 *Formato del descriptor de extent, y cadenas de extents de 1 y 3 elementos. Los valores están expresados en hexadecimal.*

El descriptor de un *extent* es de tamaño variable y se compone de los siguientes campos:

- **Cabecera**. Tiene tamaño fijo de 1 B. Los primeros 4 bits indican el tamaño del campo intervalo y los siguientes el tamaño del campo tamaño. La cabecera 0x00 indica descriptor nulo y se utiliza para cerrar una cadena de *extents*.
- **Tamaño**. Indica el número de agrupaciones que tiene el *extent*.
- **Intervalo**. El intervalo del primer elemento de la cadena indica la primera agrupación asignada al fichero. Para los demás, indica la distancia al *extent* anterior en la cadena de *extents*. Puede ser negativo, por lo que los *extents* de una cadena no necesitan estar ordenados (véase figura 5.57).

El ejemplo de fichero de 1 *extent* contiene un *extent* con cabecera 31. El 3 indica que el intervalo ocupa tres bytes, con el valor 0x37AC2E, y el 1 indica que el tamaño ocupa un byte, con el valor 0x1A. La cadena se cierra con una cabecera de valor 0x00.

En el ejemplo de fichero de 3 *extents*, se puede observar que el número de la primera agrupación asignada es 0x24A023. Para obtener el comienzo en el volumen del segundo *extent* hay que sumar el intervalo de 0x2F2A al valor 0x24A0BD, obteniendo el número de agrupación 0x24CFE7.

Un fichero grande y fragmentado requiere una cadena de *extents* muy larga, que puede no caber en el campo de datos del MFT. En este caso, se asigna otro registro MFT para disponer de más espacio. Es de destacar que un fichero NTFS viene definido por una lista de *extents*, frente al árbol de *extents* que utiliza el ext4, lo que implica que las búsquedas en ficheros grandes y fragmentados sea menos eficiente que en ext4.

Ficheros regulares

La figura 5.57 muestra dos ejemplos de ficheros regulares. Los atributos del registro MFT son los tres siguientes:

- Información estándar.
- **Nombre**. El atributo del nombre es de tipo residente y de tamaño variable. Contiene directamente el nombre del fichero tanto en formato 8.3 (MS-DOS) como en formato largo de hasta 255 caracteres Unicode.
- **Contenido**. Se puede observar que el fichero pequeño está totalmente incluido en el atributo contenido del registro MFT, sobrando espacio libre. Para ficheros más grandes, lo que se almacena dicho atributo es una cadena de *extents* que define los espacios de disco asignados al fichero.

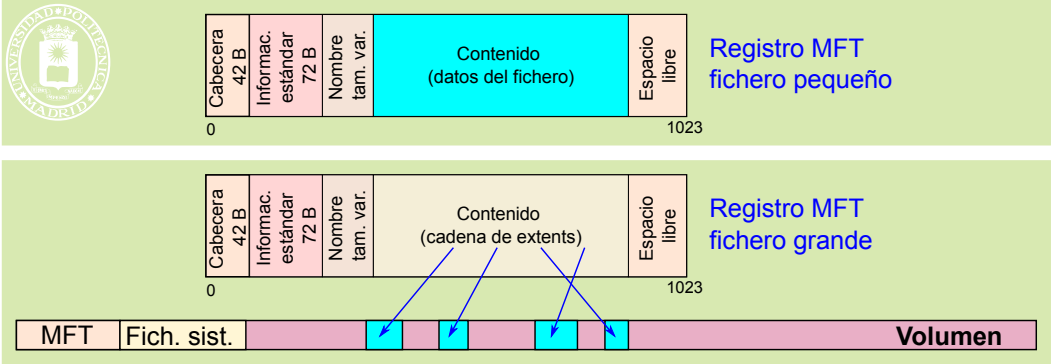


Figura 5.57 *Fichero regular*

Para incluir huecos sin soporte físico basta con incluir en la cadena de *extents* un descriptor con intervalo nulo, como muestra la figura 5.58, en la que se ha introducido un hueco de 0xA6 = 166 agrupaciones sin soporte.

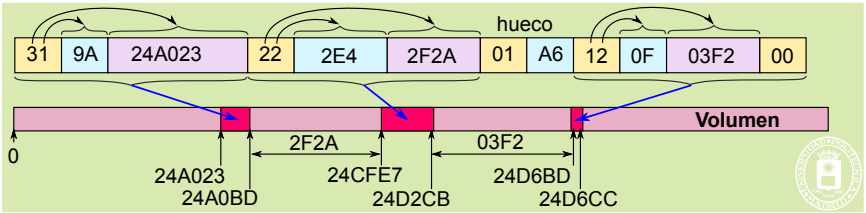


Figura 5.58 *Inclusión de un hueco sin soporte de disco en un fichero.*

Directorios

Los directorios se almacenan con una estructura de árbol B+. Si el directorio tiene pocas entradas, puede caber en el propio registro MFT, sin embargo, si es más grande requiere añadir *extents* para completar el árbol B+, como se muestra en la figura 5.59.

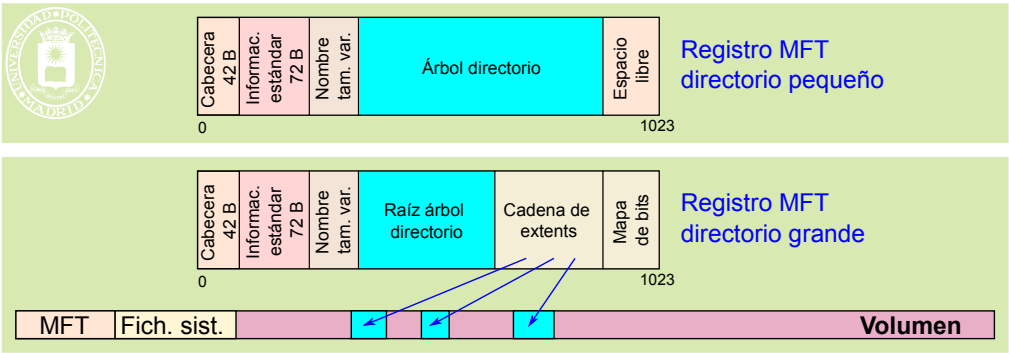


Figura 5.59 *Directorio.*

Los atributos del registro MFT de un directorio son los siguientes:

- Información estándar.
- **Nombre**. El atributo del nombre es de tipo residente y de tamaño variable. Contiene directamente el nombre del fichero tanto en formato 8.3 (MS-DOS) como en formato largo de hasta 255 caracteres Unicode.
- Árbol de **directorio**. El directorio se estructura como un árbol B+. Para un directorio grande se incluye solamente la raíz del árbol B+, estando el resto ubicado en *extents*.
- **Ubicación de índices**. Este atributo no existe en directorios pequeños y almacena la cadena de *extents* utilizados por el resto del directorio.
- **Mapa de bits**. Este atributo no existe en directorios pequeños. Indica las agrupaciones que realmente se están utilizando, de las especificadas en el atributo anterior.

5.12.9. Copias de respaldo

En las copias de respaldo hay que considerar dos operaciones. La creación de la **copia** de respaldo, partiendo de la información a respaldar, y la **recuperación**, consistente en regenerar la información perdida a partir de las copias de respaldo.

Objetivo de las copias de respaldo

El objetivo de las copias de respaldo es contar con una copia de la información que nos permita recuperar la información en caso de que se produzca su pérdida.

Las copias de respaldo se hacen cada cierto tiempo, por ejemplo, cada día, por lo que no se puede garantizar una recuperación perfecta, puesto que lo que modifique desde la última copia de respaldo no está respaldada y se habrá perdido. Cuando no se pude permitir ninguna pérdida de información se pueden utilizar las técnicas de **espejo** o **raid**.

Hay dos formas de plantearse el objetivo del respaldo.

- Recuperación de información después de su **pérdida**, ya sea por borrado o por corrupción.
- Recuperar la información existente en un **instante anterior**, por ejemplo, una versión anterior de un fichero. Exige mantener un histórico con todas las copias de seguridad realizadas a lo largo del periodo de recuperación máximo establecido como objetivo.

Tipos de copias de respaldo

Por la información que se salva, hablamos de copias globales o parciales.

- Copias **globales**. Se crea una imagen de la unidad de almacenado de la que se está haciendo el respaldo.
- Copias **parciales**. Se seleccionan los ficheros que se desean respaldar.

Por el modo en el que se hace la copia, diferenciamos entre copias totales, diferenciales e incrementales.

- Copias **totales**. Se hace un duplicado de toda la información a respaldar.
- Copias **diferenciales**. Se salva solamente las diferencias con respecto a la última copia total.
- Copias **incrementales**. Parecido al diferencial, pero solamente se guardan las diferencias con respecta a la última copia incremental.

Las copias de respaldo totales requieren muchos recursos (espacio y tiempo de copia), por lo que no se hacen con excesiva frecuencia. Las copias incrementales son las que requieren menos recursos para hacerse, pero son más complejas a la hora de hacer la recuperación, puesto que requieren utilizar la última copia total más todas las incrementales desde esa copia. Por el contrario, las diferenciales ocupan más espacio, pero solamente es necesario utilizar la última total y la última diferencial.

Sincronización

La sincronización consiste en hacer que dos directorios A y B de dispositivos distintos tengan el mismo contenido. La sincronización puede ser unidireccional, bidireccional, con borrado y sin borrado.

- En la sincronización **unidireccional** solamente se copian o actualizan los ficheros de B con los de A.
- En la sincronización **bidireccional** se copian o actualizan los ficheros de A a B y de B a A. Tendremos, tanto en A como en B, las versiones más actuales de todos los ficheros.
- En la sincronización **con borrado** de A a B, se borran en B los ficheros que no existen en A.

Ejemplos de políticas de respaldo

Ejemplo 1. Una política mínima de respaldo, que solamente protege frente a pérdida de información, consiste en utilizar una herramienta de sincronización con las opciones unidireccional y de no borrado activadas, que nos haga una copia de la información a respaldar en otro dispositivo. Esta política puede ser adecuada para un computador personal, en el que se utiliza un disco externo para hacer los respaldos. Habrá que ejecutar la herramienta de sincronización de forma periódica, por ejemplo, todas las noches.

Ejemplo 2. Se realiza un respaldo total cada mes y un respaldo diferencial cada día. Además, se almacenan todos los respaldos durante una ventana de tiempo de 5 años.

Consejos sobre copias de respaldo

- Mantener las copias de respaldo en una **ubicación distinta** a los originales. De esta forma se protege frente a robos y catástrofes.
- **Comprobar la recuperación** de forma regular, para garantizar que las copias se están haciendo bien y que se pueden leer. Esto debe hacer, por ejemplo, cada mes.
- Cuando la información sea sensible, **cifrar** las copias de respaldo.

Herramientas

Buena prueba de la importancia de las copias de respaldo es que existe una gran variedad de herramientas para realizar esta función, tanto de herramientas libres como de pago. De acuerdo al ámbito de aplicación, las herramientas se suelen clasificar en las siguientes categorías:

- Para grandes redes de sistemas.
- Para pequeñas redes de sistemas.
- Para un sistema.

Por otra parte, algunas aplicaciones, como los gestores de bases de datos, incorporan su propia herramienta de respaldo que permite hacer copias totales, diferenciales e incrementales de las bases de datos.

5.13. SISTEMAS DE FICHEROS DISTRIBUIDOS

El principal objetivo de un **sistema de ficheros distribuido** es la integración transparente de los ficheros almacenados en computadores conectados mediante una red, permitiendo compartir datos a los usuarios del conjunto. En un sistema de ficheros distribuido cada fichero se almacena en un único servidor, pero se puede acceder desde otros computadores.

Un sistema de ficheros distribuido se construye normalmente siguiendo una arquitectura cliente-servidor (véase la figura 5.60), con los módulos clientes ofreciendo la interfaz de acceso a los datos y los servidores encargándose del nombrado y acceso a los ficheros. El modelo anterior consta, normalmente, de dos componentes claramente diferenciados:

- El servicio de directorio, que se encarga de la gestión de los nombres de los ficheros. El objetivo es ofrecer un espacio de nombres único para todo el sistema, con total transparencia de acceso a los ficheros. Los nombres de los ficheros no deberían hacer alusión al servidor en el que se encuentran almacenados.
- El servicio de ficheros, que proporciona acceso a los datos de los ficheros.

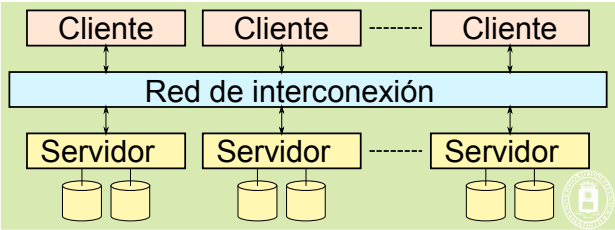


Figura 5.60 *Esquema cliente-servidor de un sistema de ficheros distribuido.*

Los aspectos más importantes relacionados con la implementación de un sistema de ficheros distribuido son el nombrado, el método de acceso a los datos y la posibilidad de utilizar cache en el sistema.

5.13.1. Nombrado

El servicio de directorios de un sistema de ficheros distribuido debe encargarse de ofrecer una visión única del sistema de ficheros. Esto implica que todos los clientes deben «ver» un mismo árbol de directorios. El sistema debe ofrecer un **espacio de nombres global y transparente**. Se pueden distinguir dos tipos de transparencia:

- **Transparencia de la posición**. El nombre del fichero no permite obtener directamente el lugar donde está almacenado.
- **Independencia de la posición**. El nombre no necesita ser cambiado cuando el fichero cambia de lugar.

Para que un cliente pueda acceder a un fichero, el servicio de directorios debe resolver el nombre, obteniendo el identificador interno del fichero y el servidor donde se encuentra almacenado. Para llevar a cabo esta resolución se puede emplear un servidor centralizado o un esquema distribuido.

El **servidor centralizado** se encarga de almacenar información sobre todos los ficheros del sistema. Esta solución, al igual que ocurre con cualquier esquema centralizado, tiene dos graves problemas: el servidor se puede convertir en un cuello de botella y el sistema presenta un único punto de fallo.

En un **esquema distribuido** cada servidor se encarga del nombrado de los ficheros que almacena. La dificultad en este caso estriba en conocer el conjunto de ficheros que maneja cada servidor. Este problema puede resolverse combinando, mediante operaciones de montaje, los diversos árboles de cada servidor para construir un único árbol de directorios. El resultado de estas operaciones de montaje es una tabla de ficheros montados que se distribuye por el sistema y que permite a los clientes conocer el servidor donde se encuentra un determinado sistema de ficheros montado.

5.13.2. Métodos de acceso

El servicio de ficheros se encarga de proporcionar a los clientes acceso a los datos de los ficheros. Existen tres modelos de acceso en un sistema de ficheros distribuido.

Modelo de carga/descarga. En este modelo, cada vez que un cliente abre un fichero, éste se transfiere en su totalidad del servidor al cliente. Una vez en el cliente, los procesos de usuario acceden al fichero como si se almacenará de forma local. Este modelo ofrece un gran rendimiento en el acceso a los datos, ya que a éstos se accede de forma local. Sin embargo, puede llevar a un modelo en el que un mismo fichero resida en múltiples clientes a la vez, lo que presenta problemas de coherencia. Además, es un modelo ineficiente cuando un cliente abre un fichero grande, pero sólo lee o escribe una cantidad de datos muy pequeña.

Modelo de servicios remotos. En este caso, el servidor ofrece todos los servicios relacionados con el acceso a los ficheros. Todas las operaciones de acceso a los ficheros se resuelven mediante peticiones a los servidores, siguiendo un modelo cliente-servidor. Normalmente, el acceso en este tipo de modelos se realiza en bloques. El gran problema de este esquema es el rendimiento, ya que todos los accesos a los datos deben realizarse a través de la red.

Empleo de cache. Este modelo combina los dos anteriores, los clientes del sistema de ficheros disponen de una cache, que utilizan para almacenar los bloques a los que se ha accedido más recientemente. Cada vez que un proceso accede a un bloque, el cliente busca en la cache local. En caso de que se encuentre, el acceso se realiza sin necesidad de contactar con el servidor.

La semántica de coutilización UNIX es muy restrictiva para un sistema de ficheros distribuido, por lo que suelen emplearse semánticas más ligeras como las de sesión, versiones o ficheros inmutables.

Lectura adelantada

La idea central de la lectura adelantada (*prefetching*) es solapar el tiempo de E/S con el tiempo de cómputo de las aplicaciones. Esto se lleva a cabo mediante la lectura por anticipado de bloques antes de que éstos sean solicitados por las aplicaciones. Normalmente, los sistemas de ficheros distribuidos hacen lectura adelantada de los bloques consecutivos a aquellos solicitados por el usuario, lo que favorece los patrones de acceso secuencial.

5.13.3. NFS

El sistema de ficheros en red NFS es una implementación y especificación de un *software* de sistema para acceso a ficheros remotos. Este sistema está diseñado para trabajar en entornos heterogéneos con diferentes arquitecturas y sistemas operativos, de hecho, existen implementaciones para todas las versiones de UNIX y Linux. También existen implementaciones para plataformas Windows.

La figura 5.61 muestra la arquitectura típica de NFS, en la que tanto el cliente como el servidor ejecutan dentro del núcleo del sistema operativo. En la parte cliente, cuando un proceso realiza una llamada al sistema de ficheros, la capa del sistema de ficheros virtual determina si el fichero es remoto, en cuyo caso le pasa la solicitud al cliente NFS.

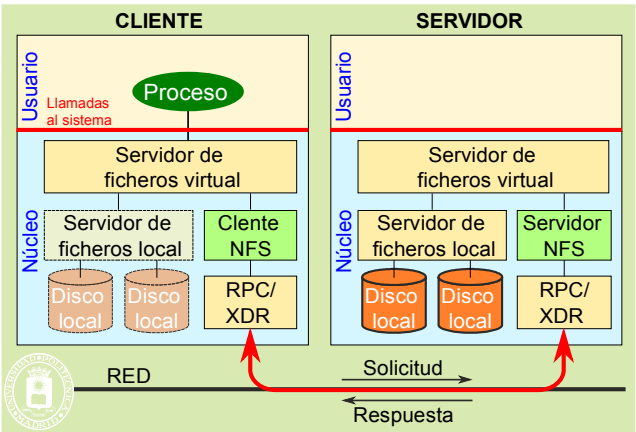


Figura 5.61 *Arquitectura de NFS.*

El NFS se encarga, utilizando llamadas a procedimiento remoto, de invocar la función adecuada en el servidor. Cuando el servidor de NFS, que ejecuta en la máquina servidora, recibe una solicitud remota, pasa la operación a la capa del sistema de ficheros virtual que es la que se encarga del acceso físico al fichero en el lado servidor.

Para que un cliente pueda acceder a un servidor de ficheros utilizando NFS deben darse dos condiciones:

- El servidor debe **exportar** los directorios, indicando clientes autorizados y permisos de acceso de los mismos. El fichero `/etc/exports` contiene los directorios exportados. Ejemplos:
  - `/home/nfs/ 10.1.1.55(rw,sync)`. Exporta el directorio `/home/nfs` a la máquina con dirección IP = 10.1.1.55, con permisos de lectura y escritura, y en modo sincronizado.
  - `/home/nfs/ 10.1.1.0/24(ro,sync)`. Exporta el directorio `/home/nfs` a la red 10.1.1.0 con máscara 255.255.255.0, con permisos de lectura y escritura, y en modo sincronizado.

- `/home/nfs/ *(ro,sync)`. Exporta el directorio `/home/nfs` a todo el mundo con permiso de solo lectura y en modo sincronizado.

- El cliente debe **montar** de forma explícita en su árbol de directorio, el directorio remoto al que se quiere acceder. Por ejemplo, en la figura 5.62, la máquina A, que actúa de servidor, exporta el directorio `/usr` y el directorio `/bin`. Si la máquina B desea acceder a los ficheros situados en estos directorios, debe montar ambos directorios en su jerarquía de directorios local, tarea que realiza el administrador. En una máquina UNIX estos montajes se realizan utilizando el mandato `mount`:

```
mount maquinaA:/usr    /usr /*Monta el directorio /usr de la máquina A en el /usr2 de la máquina B*/
mount maquinaA:/bin    /bin /*Monta el directorio /bin de la máquina A en el /bin de la máquina B*/
```

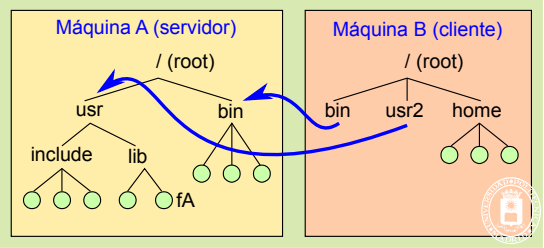


Figura 5.62 *Montaje en NFS. Los usuarios de la máquina cliente ven los subárboles de la máquina servidora que estén montados.*

Cuando se desea que un conjunto de máquinas compartan un mismo espacio de direcciones, es importante que el administrador monte el directorio remoto del servidor en todas las máquinas cliente en el mismo directorio. Si no, podría ocurrir lo que se muestra en la figura 5.63, donde existen dos clientes que montan el mismo directorio remoto, `/usr`, pero cada uno en un directorio distinto. En el cliente A el acceso al fichero `x` debe realizarse utilizando el nombre `/import/usr/x`, mientras que en el cliente B debe utilizarse el nombre /usr/x. Como puede observarse, esto hace que un mismo usuario que estuviera en la máquina A y más tarde en la máquina B no vería el mismo árbol de directorios global, algo que es fundamental cuando se quiere configurar un conjunto de máquinas como un clúster.

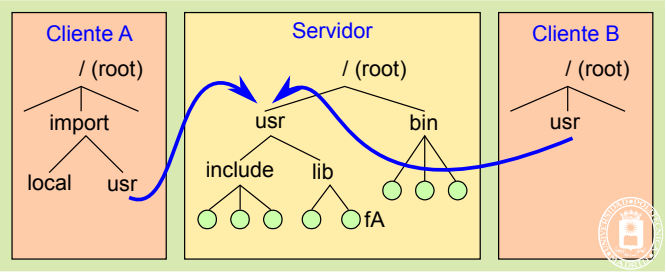


Figura 5.63 *Montado de un directorio remoto en dos clientes.*

El montaje de directorios se realiza en NFS mediante un servicio de montaje que está soportado por un proceso de montado separado del proceso que sirve las operaciones sobre ficheros. Cuando un cliente desea montar un árbol de directorio remoto contacta con este servicio en la máquina remota para realizar el montaje.

Para mejorar las prestaciones en el acceso a los ficheros, tanto el cliente como el servidor mantienen una cache con los bloques de los ficheros y sus atributos. La **cache** en el lado cliente se utiliza de igual forma que con los ficheros locales. La cache en el servidor permite mantener los bloques recientemente utilizados en memoria. Además, el servidor NFS hace lectura adelantada de un bloque cada vez que accede a un fichero. Con ello se pretende mejorar las operaciones de lectura. En cuanto a las escrituras, los clientes pueden, a partir de la versión 3 de NFS, especificar dos modos de funcionamiento:

- **Escritura inmediata**. En este caso todos los datos que se envían al servidor se almacenan en la cache de éste y se escriben de forma inmediata en el disco.
- **Escritura retrasada**. En este caso, los datos sólo se almacenan en la cache del servidor. Estos se vuelcan a disco cuando los bloques se requieren para otros usos o cuando el cliente invoca una operación de *commit* sobre el fichero. Cuando el servidor recibe esta petición escribe a disco todos los bloques del fichero. Una implementación de un cliente NFS podría enviar esta petición cuando se cierra el fichero.

El empleo de una cache en el cliente introduce, sin embargo, un posible problema de coherencia cuando dos clientes acceden a un mismo fichero. NFS intenta resolver los posibles problemas de coherencia de la siguiente forma: son los clientes los encargados de comprobar si los datos almacenados en su cache se corresponden con una copia actualizada o no. Cada vez que un cliente accede a datos o metadatos de un fichero, comprueba cuánto tiempo lleva esa información en la cache del cliente sin ser validada. Si lleva más tiempo de un cierto umbral, el cliente valida la información con el servidor para verificar si sigue siendo correcta. Si la información es correcta se utiliza, en caso contrario se descarta. La selección de este umbral es un compromiso entre consistencia y eficiencia. Un intervalo de refresco muy pequeño mejorará la consistencia pero introducirá mucha carga en el servidor. Las implementaciones típicas que se hacen en sistemas UNIX emplean para este umbral un valor comprendido entre 3 y 30 segundos.

5.13.4. CIFS

El protocolo CIFS (*Common Internet File System*) es un protocolo para acceso a ficheros remotos que se basa en el protocolo SMiB (*Server Message Block*). Este es el protocolo utilizado en sistemas Windows para compartir ficheros. También existen implementaciones en sistemas UNIX y Linux como Samba, que permite exportar directorios de este tipo de sistemas a máquinas Windows. El uso compartido de ficheros en sistemas Windows se basa en un *redirector,* que ejecuta en la máquina cliente. Este *redirector* es un controlador para sistemas de ficheros, que intercepta las llamadas al sistema de ficheros que involucran ficheros remotos y se encarga de transmitir los mensajes CIFS al servidor de ficheros. El servidor de ficheros recibe mensajes CIFS y solicita las operaciones de acceso a los ficheros a su *redirector* local, que se encarga del acceso local a los ficheros. La Figura 5.64 muestra el modelo de comunicaciones que se utiliza en CIFS para compartir ficheros.

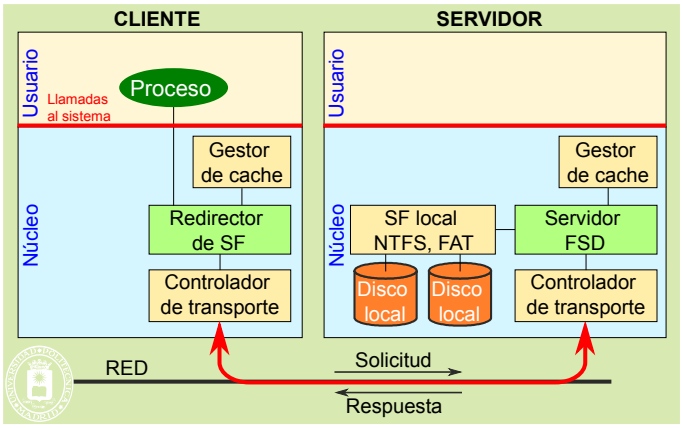


Figura 5.64 *Arquitectura de CIFS en sistemas Windows.*

Todas las comunicaciones entre el cliente y el servidor se realizan utilizando paquetes de petición o respuesta CIFS. Un paquete CIFS incluye una cabecera seguida por una lista de parámetros que dependen del tipo de operación (apertura de un fichero, lectura, etc.).

El protocolo CIFS presenta las siguientes características:

- Operaciones de acceso a ficheros como `open`, `close`, `read`, `write` y `seek`.
- Cerrojos sobre ficheros. Permite bloquear un fichero o parte de un fichero, de forma que una vez bloqueado el fichero o parte del mismo se impide el acceso a otras aplicaciones.
- Se incluye una cache con lectura adelantada y escritura diferida. Incluye un protocolo de coherencia, que permite que varios clientes accedan al fichero simultáneamente para leerlo o escribirlo.
- Notificación de los cambio en un fichero. Las aplicaciones pueden registrarse en un servidor para que éste les notifique cuándo se ha modificado un fichero o directorio.
- Negociación de la versión del protocolo. Los clientes y servidores pueden negociar la versión del protocolo a utilizar en el acceso a los ficheros.
- Atributos extendidos. Se pueden añadir a los atributos típicos de un fichero, otros como, por ejemplo el nombre del autor.
- Volúmenes virtuales replicados y distribuidos. El protocolo permite que diferentes árboles de directorios de varios volúmenes aparezcan al usuario como si fuera un único volumen. Si los ficheros y directorios de un subárbol de directorio se mueven o se replican físicamente, el protocolo redirige de forma transparente a los clientes al servidor apropiado.
- Independencia en la resolución de nombres. Los clientes pueden resolver los nombres utilizando cualquier mecanismo de resolución.
- Peticiones agrupadas. Varias operaciones sobre ficheros se pueden agrupar en un único mensaje para optimizar la transferencia de datos.
- Permiten utilizar nombres de fichero con Unicode.

5.13.5. Empleo de paralelismo en el sistema de ficheros

Los sistemas de ficheros distribuidos tradicionales, como NFS o CIFS, aunque ofrecen un espacio de nombres global, que permite a múltiples clientes compartir ficheros, presentan un importante problema desde el punto de vista de prestaciones y escalabilidad. El hecho de que múltiples clientes utilicen un mismo servidor de ficheros, puede convertir a éste en un cuello de botella en el sistema, que puede afectar gravemente a la escalabilidad del mismo. Este problema se puede resolver mediante el empleo de paralelismo en el sistema de ficheros.

El paralelismo en el sistema de ficheros se obtiene utilizando varios servidores o nodos de entrada/salida, cada uno con sus diferentes dispositivos de almacenamiento, y distribuyendo los datos de los ficheros entre los diferentes servidores y dispositivos de almacenamiento. Este enfoque permite el acceso paralelo a los ficheros. Esta idea es similar a la utilizada en los discos RAID. El empleo de paralelismo en el sistema de ficheros permite mejorar las prestaciones de dos formas:

- Permite el acceso paralelo a diferentes ficheros, puesto que se utilizan varios servidores y dispositivos de almacenamiento para almacenarlos.
- Permite el acceso paralelo a un mismo fichero, ya que los datos de un fichero también se distribuyen entre varios servidores y dispositivos de almacenamiento.

El empleo de paralelismo en el sistema de ficheros es diferente, sin embargo, del empleo de sistemas de ficheros replicados. En efecto, en un sistema de ficheros replicado cada servidor almacena una copia completa de un fichero. Utilizando paralelismo en el sistema de ficheros, cada dispositivo de almacenamiento, en cada servidor, almacena sólo una parte del fichero.

Existen dos arquitecturas básicas de entrada/salida paralela:

- **Bibliotecas de E/S paralelas**, que constan de un conjunto de funciones altamente especializadas para el acceso a los datos de los ficheros. Un ejemplo representativo de estas bibliotecas es MPI-IO, una extensión de la interfaz de paso de mensajes MPI.
- **Sistemas de ficheros paralelos**, que operan de forma independiente de los clientes ofreciendo más flexibilidad y generalidad. Ejemplos representativos de sistemas de ficheros paralelos son PVFS, GPFS y Lustre.

5.14. FICHEROS DE INICIO SESIÓN EN LINUX

Los ficheros de *script* que se ejecutan cuando se inicia una sesión pueden depender de la distribución Linux que se esté utilizando y del tipo de *shell*. Para el caso del *shell* Bash (*Bourne-Again shell*), que es el más utilizado en Linux, los ficheros básicos son los siguientes:

- Ficheros comunes a todos los usuarios:
  - `/etc/profile`. Este fichero se ejecuta cuando cualquier usuario inicia la sesión. Ejemplo de este fichero:

```
#!/etc/profile
# No core files by default
ulimit -S -c 0 > /dev/null 2>&1
# HOSTNAME is the result of running the hostname command
declare –x HOSTNAME=´/bin/hostname´
# No more than 1000 lines of Bash command history
declare –x HISTSIZE=1000
# If PostgreSQL is installed, add the Postgres commands
# to the user's PATH
If test –r /usr/bin/pgsql/bin ; then
  declare –x PATH="$PATH"":/usr/bin/pgsql/bin"
fi
# end of general profile
```

  - `/etc/bashrc`. Se ejecuta cada vez que un usuario, que ya ha iniciado una sesión, lance de forma interactiva el programa bash.
- Ficheros privados de cada usuario.
  - Existen los siguientes ficheros que se pueden ejecutar cuando el usuario inicia la sesión: `~/.bash_profile`, `~/.bash_login` y `~/.profile`. El fichero que realmente se ejecuta viene determinado por el siguiente seudocódigo:

```
IF ~/.bash  profile exists THEN
   execute ~/.bash_profile
ELSE
   IF ~/.bash  login exist THEN
      execute ~/.bash_login
   ELSE
      IF ~/.profile exist THEN
         execute ~/.profile
      END IF
   END IF
END IF
```

  - Cuando el usuario cierra la sesión se ejecuta su fichero: `~/.bash_logout`

5.15. SERVICIOS DE E/S

Se expondrán en primer lugar los servicios relacionados con el reloj y, a continuación, se presentarán los servicios correspondientes a los dispositivos de entrada/salida convencionales.

Servicios relacionados con el reloj

Se pueden clasificar en las siguientes categorías:

- **Servicios de hora y fecha**. Debe existir un servicio para obtener la fecha y la hora, así como para modificarla. Dada la importancia de este parámetro en la seguridad del sistema, el servicio de modificación sólo lo podrán usar procesos privilegiados.
- **Temporizadores**. El sistema operativo suele ofrecer servicios que permiten establecer plazos de espera síncronos, o sea, que bloqueen al proceso el tiempo especificado, y asíncronos, esto es, que no bloqueen al proceso, pero le manden algún tipo de evento cuando se cumpla el plazo.
- **Servicios para obtener la contabilidad y las estadísticas recogidas por el sistema operativo**. Por ejemplo, en casi todos los sistemas operativos existe algún servicio para obtener información sobre el tiempo de procesador que ha consumido un proceso.

Servicios de entrada/salida

La mayoría de los sistemas operativos modernos proporcionan los mismos servicios para trabajar con dispositivos de entrada/salida que los que usan con los ficheros. Esta equivalencia es muy beneficiosa, ya que proporciona a los programas independencia del medio con el que trabajan. Así, para acceder a un dispositivo se usan los servicios habituales para abrir, leer, escribir y cerrar ficheros.

Sin embargo, en ocasiones es necesario poder realizar desde un programa alguna operación dependiente del dispositivo. Por ejemplo, suponga un programa que desea solicitar al usuario una contraseña a través del terminal. Este programa necesita poder desactivar el eco en la pantalla para que no pueda verse la contraseña mientras se teclea. Se requiere, por tanto, algún mecanismo para poder realizar este tipo de operaciones que dependen de cada tipo de dispositivo. Existen al menos dos posibilidades no excluyentes:

- Permitir que este tipo de opciones se puedan especificar como indicadores en el propio servicio para abrir el dispositivo.
- Proporcionar un servicio que permita invocar estas opciones dependientes del dispositivo.

5.15.1. Servicios de entrada/salida en UNIX

Debido al tratamiento diferenciado que se da al reloj, se presentan separadamente los servicios relacionados con el mismo de los correspondientes a los dispositivos de entrada/salida convencionales.

En cuanto a los servicios del reloj, se van a especificar de acuerdo a las tres categorías antes planteadas: fecha y hora, temporizadores y contabilidad. Con respecto a los servicios destinados a dispositivos convencionales, se presentarán de forma genérica, haciendo especial hincapié en los relacionados con el terminal.

Servicios de fecha y hora

◙ time_t **time** (time_t *t);

La llamada `time` obtiene la fecha y hora. Esta función devuelve el número de segundos transcurridos desde el 1 de enero de 1970 en UTC. Si el argumento no es nulo, también lo almacena en el espacio apuntado por el mismo.

Algunos sistemas UNIX proporcionan también el servicio `gettimeofday` que proporciona una precisión de microsegundos.

La biblioteca estándar de C contiene funciones que transforman el valor devuelto por `time` a un formato más manejable (año, mes, día, horas, minutos y segundos), tanto en UTC, la función `gmtime`, como en horario local, la función `localtime`.

◙ int **stime** (time_t *t);

Esta función fija la hora del sistema de acuerdo al parámetro recibido, que se interpreta como el número de segundos desde el 1 de enero de 1970 en UTC. Se trata de un servicio sólo disponible para el super-usuario. En algunos sistemas UNIX existe la función `settimeofday`, que permite realizar la misma función, pero con una precisión de microsegundos.

Todas estas funciones requieren el fichero de cabecera `time.h`. Como ejemplo del uso de estas funciones, se presenta el programa 5.1, que imprime la fecha y la hora en horario local.

Programa 5.1 Programa que imprime la fecha y hora actual.

```
#include <stdio.h>
#include <time.h>
```

```
int main(int argc, char *argv[]) {
    time_t tiempo;
    struct tm *fecha;

    tiempo = time(NULL);

    fecha = localtime(&tiempo);

    /* hay que ajustar el año ya que lo devuelve respecto a 1900 */
    printf ("%02d/%02d/%04d %02d:%02d:%02d\n",
        fecha->tm_mday, fecha->tm_mon,
        fecha->tm_year+1900, fecha->tm_hour,
        fecha->tm_min, fecha->tm_sec);

    return 0;
}
```

Servicios de temporización

- unsigned int **alarm** (unsigned int segundos);

El servicio `alarm` permite establecer un temporizador en UNIX. El plazo debe especificarse en segundos. Cuando se cumple dicho plazo, se le envía al proceso la señal SIGALRM. Sólo se permite un temporizador activo por cada proceso. Debido a ello, cuando se establece un temporizador, se desactiva automáticamente el anterior. El argumento especifica la duración del plazo en segundos. La función devuelve el número de segundos que le quedaban pendientes a la alarma anterior, si había alguna pendiente. Una alarma con un valor de cero desactiva un temporizador activo.

En UNIX existe otro tipo de temporizador que permite establecer plazos con una mayor resolución y con modos de operación más avanzados. Este tipo de temporizador se activa con la función `setitimer`.

Servicios de contabilidad

UNIX define diversas funciones que se pueden englobar en esta categoría. Este apartado va a presentar una de las más usadas.

- clock_t **times** (struct tms *info);

El servicio `times` devuelve información sobre el tiempo de ejecución de un proceso y de sus procesos hijos. Esta función rellena la zona apuntada por el puntero recibido como argumento con información sobre el uso del procesador, en modo usuario y en modo sistema, tanto del propio proceso como de sus procesos hijos. Además, devuelve un valor relacionado con el tiempo real en el sistema (habitualmente, el número de interrupciones de reloj que se han producido desde el arranque del sistema). Este valor no se usa de manera absoluta. Normalmente, se compara el valor devuelto por dos llamadas a `times` realizadas en distintos instantes para determinar el tiempo real transcurrido entre esos dos momentos.

El programa 5.2 muestra el uso de esta llamada para determinar cuánto tarda en ejecutarse un programa que se recibe como argumento. La salida del programa muestra el tiempo real, el tiempo de procesador en modo usuario y el tiempo de procesador en modo sistema consumido por el programa especificado.

Programa 5.2 Programa que muestra el tiempo real, el tiempo en modo usuario y en modo sistema que se consume durante la ejecución de un programa.

```
#include <stdio.h>
#include <unistd.h>
#include <time.h>
#include <sys/times.h>

int main(int argc, char *argv[]) {
    struct tms InfoInicio, InfoFin;
    clock_t t_inicio, t_fin;
    long tickporseg;

    if (argc<2) {
        fprintf(stderr, "Uso: %s programa [args]\n", argv[0]);
        return 1;
    }
    /* obtiene el número de int. de reloj por segundo */
    tickporseg = sysconf(_SC_CLK_TCK);
```

```
    t_inicio = times(&InfoInicio);

    if (fork()==0) {
        execvp(argv[1], &argv[1]);
        perror("error ejecutando el programa");
        return 1;
    }
    wait(NULL);
    t_fin = times(&InfoFin);
    printf("Tiempo real: %8.2f\n",
        (float)(t_fin - t_inicio)/tickporseg);

    printf("Tiempo de usuario: %8.2f\n",
        (float)(InfoFin.tms_cutime –
        InfoInicio.tms_cutime)/tickporseg);

    printf("Tiempo de sistema: %8.2f\n",
        (float)(InfoFin.tms_cstime –
        InfoInicio.tms_cstime)/tickporseg);

    return 0;
}
```

Servicios de entrada/salida sobre dispositivos

Como ocurre en la mayoría de los sistemas operativos, se utilizan los mismos servicios que para acceder a los ficheros. En el caso de UNIX, por tanto, se trata de los servicios `open`, `read`, `write` y `close`.

- int **ioctl** (int descriptor, int petición, ...);

La llamada `ioctl` permite configurar las opciones específicas de cada dispositivo o realizar operaciones que no se pueden expresar usando un `read` o un `write` (como, por ejemplo, rebobinar una cinta). El primer argumento corresponde con un descriptor de fichero asociado al dispositivo correspondiente. El segundo argumento identifica la clase de operación que se pretende realizar sobre el dispositivo, que dependerá del tipo de mismo. Los argumentos restantes, normalmente sólo uno, especifican los parámetros de la operación que se pretende realizar. Aunque esta función se incluya en prácticamente todas las versiones de UNIX, el estándar no la define.

El uso más frecuente de `ioctl` es para establecer parámetros de funcionamiento de un terminal. Sin embargo, el estándar define un conjunto de funciones específicas para este tipo de dispositivo. Las dos más frecuentemente usadas son `tcgetattr` y `tcsetattr`, destinadas a obtener los atributos de un terminal y a modificarlos, respectivamente. Sus prototipos son los siguientes:

- int **tcgetattr** (int descriptor, struct termios *atrib);
- int **tcsetattr** (int descriptor, int opción, struct termios *atrib);

La función `tcgetattr` obtiene los atributos del terminal asociado al descriptor especificado. La función `tcgetattr` modifica los atributos del terminal que corresponde al descriptor pasado como parámetro. Su segundo argumento establece cuándo se producirá el cambio: inmediatamente o después de que la salida pendiente se haya transmitido.

La estructura `termios` contiene los atributos del terminal, que incluye aspectos tales como el tipo de procesado que se realiza sobre la entrada y sobre la salida, los parámetros de transmisión, en el caso de un terminal serie, o la definición de qué caracteres tienen un significado especial. Dada la gran cantidad de atributos existentes (más de 50), en esta exposición no se entrará en detalles sobre cada uno de ellos, mostrándose simplemente un ejemplo de cómo se usan estas funciones. Como ejemplo, se presenta el programa 5.3, que lee una contraseña del terminal desactivando el eco en el mismo para evitar problemas de seguridad.

Programa 5.3 Programa que lee del terminal desactivando el eco.

```
#include <stdio.h>
#include <termios.h>
#include <unistd.h>

#define TAM_MAX 32
int main(int argc, char *argv[]) {
    struct termios term_atrib;
    char linea[TAM_MAX];

    /* Obtiene los atributos del terminal */
    if (tcgetattr(0, &term_atrib)<0)
```

```
    {
        perror("Error obteniendo atributos del terminal");
        return 1;
    }
    /* Desactiva el eco. Con TCSANOW el cambio sucede inmediatamente */
    term_atrib.c_lflag &= ~ECHO;
    tcsetattr(0, TCSANOW, &term_atrib);
    printf ("Introduzca la contraseña: );

    /* Lee la línea y a continuación la muestra en pantalla */
    if (fgets(linea, TAM_MAX, stdin))
    {
        linea[strlen(linea)-1]='\0'; /* Eliminando fin de línea */
        printf("\nHas escrito %s\n", linea);
    }

    /* Reactiva el eco */
    term_atrib.c_lflag |= ECHO;
    tcsetattr(0, TCSANOW, &term_atrib);
    return 0;
}
```

5.15.2. Servicios de entrada/salida en Windows

Se van a presentar clasificados en las mismas categorías que se han usado para exponer los servicios UNIX.

Servicios de fecha y hora

- BOOL **GetSystemTime** (LPSYSTEMTIME tiempo);

Es la llamada para obtener la fecha y hora. Esta función devuelve en el espacio asociado a su argumento la fecha y hora actual en una estructura organizada en campos que contienen el año, mes, día, hora, minutos, segundos y milisegundos. Los datos devueltos corresponden con el horario estándar UTC. La función `GetLocalTime` permite obtener información según el horario local.

- BOOL **SetSystemTime** (LPSYSTEMTIME tiempo);

Llamada que permite modificar la fecha y hora en el sistema. Esta función establece la hora del sistema de acuerdo al parámetro recibido, que tiene la misma estructura que el usado en la función anterior (desde el año a los milisegundos actuales).

Como ejemplo, se plantea el programa 5.4, que imprime la fecha y hora actual en horario local.

Programa 5.4 Programa que imprime la fecha y hora actual.

```
#include <windows.h>
#include <stdio.h>

int main (int argc, char *argv [])
{
    SYSTEMTIME Tiempo;

    GetLocalTime(&Tiempo);
    printf("%02d/%02d/%04d %02d:%02d:%02d\n",
            Tiempo.wDay, Tiempo.wMonth,
            Tiempo.wYear, Tiempo.wHour,
            Tiempo.wMinute, Tiempo.wSecond);
    return 0;
}
```

Servicios de temporización

- UINT **SetTimer** (HWND ventana, UINT evento, UINT plazo, TIMERPROC función);

La función `SetTimer` permite establecer un temporizador con una resolución de milisegundos. Un proceso puede tener múltiples temporizadores activos simultáneamente. Esta función devuelve un identificador del temporizador, y recibe como parámetros el identificador de la ventana asociada con el temporizador (si tiene un valor nulo, no se le

asocia a ninguna), el identificador del evento (este parámetro se ignora en el caso de que el temporizador no esté asociado a ninguna ventana), un plazo en milisegundos y la función que se invocará cuando se cumpla el plazo.

Servicios de contabilidad

En Windows existen diversas funciones que se pueden encuadrar en esta categoría. Como en el caso de UNIX, se muestra como ejemplo el servicio que permite obtener información sobre el tiempo de ejecución de un proceso. Esta función es `GetProcessTimes` y su prototipo es el siguiente:

- BOOL **GetProcessTimes** (HANDLE proceso, LPFILETIME t_creación, LPFILETIME t_fin, LPFILETIME t_sistema, LPFILETIME t_usuario);

Esta función obtiene, para el proceso especificado, su tiempo de creación y finalización, así como cuánto tiempo de procesador ha gastado en modo sistema y cuánto en modo usuario. Todos estos tiempos están expresados como el número de centenas de nanosegundos transcurrido desde el 1 de enero de 1601, almacenándose en un espacio de 64 bits. La función `FileTimeToSystemTime` permite transformar este tipo de valores en el formato ya comentado de año, mes, día, hora, minutos, segundos y milisegundos.

Como ejemplo del uso de esta función, se presenta el programa 5.5, que imprime el tiempo que tarda en ejecutarse el programa que se recibe como argumento, mostrando el tiempo real de ejecución, así como el tiempo de procesador en modo usuario y en modo sistema consumido por dicho programa.

Programa 5.5 Programa que muestra el tiempo real, el tiempo en modo usuario y en modo sistema que se consume durante la ejecución de un programa.

```
#include <windows.h>
#include <stdio.h>
#define MAX 255

int main (int argc, char *argv [])
{
    int i;
    BOOL result;
    STARTUPINFO InfoInicial;
    PROCESS_INFORMATION InfoProc;
    union {
        LONGLONG numero;
        FILETIME tiempo;
    } TiempoCreacion, TiempoFin, TiempoTranscurrido;

    FILETIME TiempoSistema, TiempoUsuario;
    SYSTEMTIME TiempoReal, TiempoSistema, TiempoUsuario;

    CHAR mandato[MAX];

    /* Obtiene la línea de mandato saltándose el primer argumento */
    sprintf(mandato,"%s", argv[1]);
    for (i=2; i<argc; i++)
        sprintf(mandato,"%s %s", mandato, argv[i]);

    GetStartupInfo(&InfoInicial);

    result= CreateProcess(NULL, mandato, NULL, NULL, TRUE,
            NORMAL  PRIORITY_CLASS, NULL, NULL, &InfoInicial,
            &InfoProc);
    if (! result) {
        fprintf(stderr, ”error creando el proceso\n”);
        return 1;
    }

    WaitForSingleObject(InfoProc.hProcess, INFINITE);

    GetProcessTimes(InfoProc.hProcess, &TiempoCreacion.tiempo,
        &TiempoFin.tiempo, &TiempoSistema, &TiempoUsuario);

    TiempoTranscurrido.numero= TiempoFin.numero –
     TiempoCreacion.numero;

    FileTimeToSystemTime(&TiempoTranscurrido.tiempo, &TiempoReal);
```

```
    FileTimeToSystemTime(&TiempoSistema, &TiempoSistema);
    FileTimeToSystemTime(&TiempoUsuario, &TiempoUsuario);
    printf ("Tiempo real: %02d:%02d:%02d:%03d\n",
        TiempoReal.wHour, TiempoReal.wMinute, TiempoReal.wSecond,
        TiempoReal.wMilliseconds);

    printf ("Tiempo de usuario: %02d:%02d:%02d:%03d\n",
        TiempoUsuario.wHour, TiempoUsuario.wMinute,
        TiempoUsuario.wSecond, TiempoUsuario.wMilliseconds);
    printf ("Tiempo de sistema: %02d:%02d:%02d:%03d\n",
        TiempoSistema.wHour, TiempoSistema.wMinute,
        TiempoSistema.wSecond, TiempoSistema.wMilliseconds);
    return 0;
}
```

Servicios de entrada/salida sobre dispositivos

Al igual que en UNIX, para acceder a los dispositivos se usan los mismos servicios que se utilizan para los ficheros. En este caso, `CreateFile`, `CloseHandle`, `ReadFile` y `WriteFile`.

Sin embargo, como se ha analizado previamente, estos servicios no son suficientes para cubrir todos los aspectos específicos de cada tipo de dispositivo. Centrándose en el caso del terminal, Windows ofrece un servicio que permite configurar el modo de funcionamiento del mismo. Se denomina `SetConsoleMode` y su prototipo es el siguiente:

- BOOL **SetConsoleMode** (HANDLE consola, DWORD modo);

Esta función permite configurar el modo de operación de una determinada consola. Se pueden especificar, entre otras, las siguientes posibilidades: entrada orientada a líneas (`ENABLE_LINE_INPUT`), eco activo (`ENABLE_ECHO_INPUT`), procesamiento de los caracteres especiales en la salida (`ENABLE_PROCESSED_OUTPUT`) y en la entrada (`ENABLE_PROCESSED_INPUT`).

Es importante destacar que para que se tengan en cuenta todas estas opciones de configuración, hay que usar para leer y escribir del dispositivo las funciones `ReadConsole` y `WriteConsole`, respectivamente, en vez de utilizar `ReadFile` y `WriteFile`. Los prototipos de estas dos nuevas funciones son casi iguales que las destinadas a ficheros:

- BOOL **ReadConsole** (HANDLE consola, LPVOID mem, DWORD a_leer, LPDWORD leidos, LPVOID reservado);
- BOOL **WriteConsole** (HANDLE consola, LPVOID mem, DWORD a_escribir, LPVOID escritos, LPVOID reservado);

El significado de los argumentos de ambas funciones es el mismo que los de `ReadFile` y `WriteFile`, respectivamente, excepto el último parámetro que no se utiliza.

Como muestra del uso de estas funciones, se presenta el programa 5.6 que lee la información del terminal desactivando el eco del mismo.

Programa 5.6 Programa que lee del terminal desactivando el eco.

```
#include <windows.h>
#include <stdio.h>
#define TAM_MAX 32

int main (int argc, char *argv[])
{
    HANDLE entrada, salida;
    DWORD leidos, escritos;
    BOOL result;
    BYTE clave[TAM_MAX];

    entrada= CreateFile("CONIN$", GENERIC_READ | GENERIC_WRITE, 0,
            NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

    if (entrada == INVALID_HANDLE_VALUE) {
        fprintf(stderr, "No puede abrirse la consola para lectura\n");
        return 1;
    }

    salida= CreateFile("CONOUT$", GENERIC_WRITE, 0,
            NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
```

```
    if (salida == INVALID_HANDLE_VALUE) {
        fprintf(stderr, "No puede abrirse la consola para escritura\n");
        return 1;
    }

    result = SetConsoleMode(entrada, ENABLE_LINE_INPUT |
            ENABLE_PROCESSED_INPUT)
        && SetConsoleMode(salida, ENABLE_WRAP_AT_EOL_OUTPUT |
            ENABLE_PROCESSED_OUTPUT)
        && WriteConsole(salida, "Introduzca la contraseña: ", 26,
                &escritos, NULL)
        && ReadConsole(entrada, clave, TAM_MAX, &leidos, NULL);

    if (! result) {
        fprintf(stderr, "error accediendo a la consola\n");
        return 1;
    }

    CloseHandle(entrada);
    CloseHandle(salida);
    return result;
}
```

5.16. SERVICIOS DE FICHEROS Y DIRECTORIOS

Un fichero es un tipo abstracto de datos. Por tanto, para que esté completamente definido, es necesario definir las operaciones que pueden ejecutarse sobre un objeto fichero. En general, los sistemas operativos proporcionan operaciones para crear un fichero, almacenar información en él y recuperar dicha información más tarde, algunas de las cuales ya se han nombrado en este capítulo.

En la mayoría de los sistemas operativos modernos los directorios se implementan como ficheros que almacenan una estructura de datos definida: entradas de directorios. Por ello, los servicios de ficheros pueden usarse directamente sobre directorios. Sin embargo, la funcionalidad necesaria para los directorios es mucho más restringida que la de los ficheros, por lo que los sistemas operativos suelen proporcionar servicios específicos para satisfacer dichas funciones de forma eficiente y sencilla.

En esta sección se muestran los servicios más frecuentes para ficheros y directorios usando dos interfaces para servicios de sistemas operativos: el estándar UNIX y en Windows. También se muestran ejemplos de uso de dichos servicios.

5.16.1. Servicios UNIX para ficheros

UNIX proporciona una visión lógica de fichero equivalente a una tira secuencial de bytes. Para acceder al fichero, se mantiene un **puntero de posición**, a partir del cual se ejecutan las operaciones de lectura y escritura sobre el fichero.

A continuación se describen los **principales servicios de ficheros** descritos en el estándar UNIX. Estos servicios están disponibles en prácticamente todos los sistemas operativos modernos.

- int **open** (const char path, int oflag, /* mode_t mode */ ...);

El `open` es una operación que parte del nombre de un fichero regular y devuelve el descriptor de fichero que permite utilizarlo. Sus **argumentos** son los siguientes:

- `path` es el nombre lógico absoluto o relativo de un fichero regular (no directorio).
- `oflag` permite especificar las opciones de apertura. Se puede utilizar una combinación de ellos.
  - O_RDONLY: Sólo lectura
  - O_WRONLY: Sólo escritura
  - O_RDWR: Lectura y escritura
  - O_APPEND: Se accede siempre al final del fichero
  - O_CREAT: Si existe no tiene efecto. Si no existe lo crea
  - O_TRUNC: Trunca a cero si se abre para escritura
- `mode`: Especifica los bits de permiso para el nuevo fichero. Valen sólo cuando realmente se crea el fichero, es decir, se usa la opción O_CREAT y el fichero no existía previamente. Este argumento se expresa en octal, lo que se indica empezando por 0.

El servicio **devuelve** un descriptor de fichero, pero si fracasa devuelve -1 y un código de error en la variable `errno`. El descriptor devuelto es el primero libre de la tabla de descriptores del proceso que solicita el servicio.

El `open` puede fracasar por un error (fichero no existe, no se tienen permisos, no se puede crear, etc.) o porque le llega una señal al proceso, lo que se puede saber si `errno` devuelve el valor EINTR*.

Permisos: Para poder abrir un fichero, se requieren permisos de acceso desde el directorio raíz hasta el directorio donde está ubicado el fichero. Además, se han de tener sobre el fichero los permisos solicitados en `oflag`. Para poder crear el fichero es necesario tener permisos de acceso hasta el directorio donde se añade el fichero, así como permisos de escritura en dicho directorio.

Descripción : Utiliza una entrada libre de la tabla intermedia y se copia el nodo-i del fichero en la tabla de nodos-i en memoria o incrementa su nopens si ya está en memoria.

Ejemplos:

```
fd = open("/home/juan/dat.txt", O_RDONLY);
fd = open("/home/juan/dat.txt", O_WRONLY|O_CREAT|O_TRUNC, 0640);
```

El segundo ejemplo, que es equivalente al servicio `creat`, creará un fichero si no existe. Si existe simplemente se abre el fichero y se vacía, poniendo su tamaño a 0, pero no se cambian sus permisos ni su dueño. En caso de crearse realmente el fichero se le asignan los siguientes atributos:

- UID dueño del fichero se hace igual al UID_efectivo del proceso.
- GID dueño del fichero se hace igual al GID_efectivo del proceso.
- Se le asignan los permisos indicados en `mode` enmascarados por la máscara `umask` de creación de ficheros y directorios del proceso (`mode & ~umask`).

La figura 5.66 muestra el efecto de ejecutar `fd2 = open("datos.txt",O_RDONLY);`. Se utiliza la primera entrada libre de la tabla de descriptores, una entrada de la tabla intermedia y, como se ha supuesto que el fichero ya está abierto, se incrementa nopens.

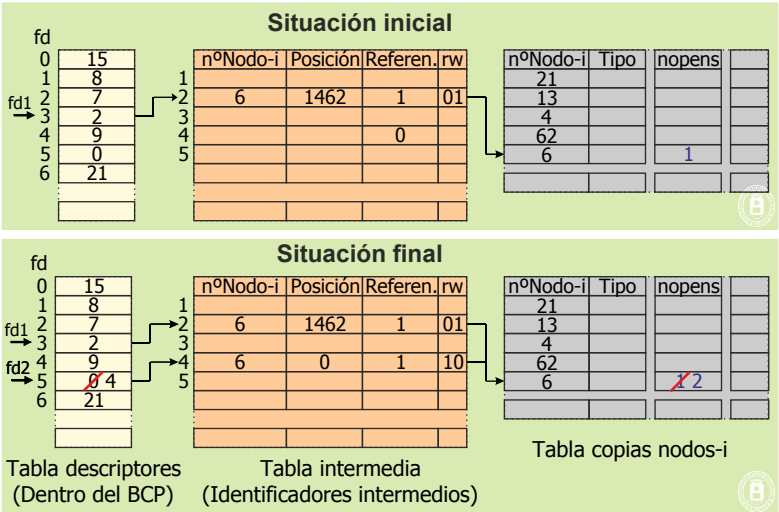


Figura 5.65 Efecto de ejecutar `fd2 = open("datos.txt",O_RDONLY);`

◙ int **creat**(const char *path, mode_t mode);

Este servicio es idéntico al servicio **`open`** con las opciones `O_WRONLY|O_CREAT|O_TRUNC`, analizado anteriormente.

Ejemplo: `fd = creat("datos.txt", 0751);`

Observe que, aunque UNIX no hace uso de las extensiones de los ficheros, no parece razonable darle permisos de ejecución a un fichero que, por su nombre, parece un fichero de texto.

◙ ssize_t **read** (int fildes, void *buf, size_t nbyte);

Este servicio permite al proceso leer datos de un fichero abierto y en una zona de su espacio de memoria.

Los **argumentos** son los siguientes:

- `fildes`: descriptor de fichero.
- `buf`: zona donde almacenar los datos.
- `nbytes`: número de bytes a leer.

Devuelve el número de bytes realmente leídos o -1 si fracasa.

Descripción:

- La lectura se lleva a cabo a partir del valor actual del puntero de posición del fichero.
- Transfiere `nbytes` como máximo, puesto que puede leer menos datos de los solicitados si se llega al fin de fichero o se lee de un fichero especial.
- También se emplea para leer de ficheros especiales como un terminal, de un *pipe* o de un *socket*.
- El servicio puede fracasar por una señal, retornando -1. Lo que no es error: `errno` devuelve la razón del fracaso.

- Para los ficheros regulares, después de la lectura, se incrementa el puntero del fichero con el número de bytes realmente transferidos.
- En ficheros regulares, si el puntero está al final de fichero, el `read` retorna 0, indicando final de fichero.
- En ficheros especiales, si no hay datos, se queda bloqueado hasta que lleguen datos, por ejemplo, se pulse una tecla o llegue información al *socket*.
- El servicio no comprueba el tamaño del *buffer*, por lo que se puede **desbordar** si `nbyte` es mayor que el tamaño del *buffer*.

◙ ssize_t **write** (int fildes, const void *buf, size_t nbyte);

Este servicio permite al proceso escribir en un fichero una porción de datos existente en su espacio de memoria.

Los **argumentos** son los siguientes:

- `fildes`: descriptor de fichero.
- `buf`: zona de datos a escribir.
- `nbytes`: número de bytes a escribir.

Devuelve el número de bytes realmente escritos -1 si fracasa.

Descripción:

- La escritura se lleva a cabo a partir del valor actual del puntero de posición del fichero.
- Transfiere `nbytes` o menos, lo que es poco frecuente debido al punto siguiente.
- Si se rebasa el fin de fichero el fichero aumenta de tamaño automáticamente, siempre que se pueda, es decir, que no se llegue al tamaño máximo del fichero o se rebase algún límite de implementación del sistema operativo.
- El servicio puede fracasar por una señal, retornando -1. Lo que no es error: `errno` devuelve la razón del fracaso.
- Para los ficheros regulares, después de la escritura, se incrementa el puntero del fichero con el número de bytes realmente escritos.
- El servicio no comprueba el tamaño del *buffer*, por lo que se puede escribir basura si `nbyte` es mayor que el *buffer*.
- Si, en la apertura del fichero, se especificó `O_APPEND`, antes de escribir, se pone el apuntador de posición al final del fichero, de forma que la información siempre se añade al final.

Ejemplos de servicios `read` y `write`.

```
int total;
n = read(d, &total, sizeof(int));     /*Se lee un entero */
total = 1246;                         /*Si sizeof(int) = 4 total es 00 00 04 DE */
n = write(d, &total, sizeof(int));
/* Suponiendo que el computador utiliza representación big-endian se
escriben en este orden los cuatro bytes 00, 00, 04 y DE, no los cuatro  */
caracteres ASCII 1, 2, 4 y 6 */
float m[160];
n = read(d, m, sizeof(float)*160);
/*Se leen hasta 160 float, lo que pueden ser 4*160 bytes */
typedef struct registro{
int identificador;
float edad, altura, peso;
} registro;
registro individuo[500];
n = read(d, individuo, 500*sizeof(registro));
estadistica_ciega(individuo, n/sizeof(registro));
```

◙ int **close** (int fildes);

La llamada `close` libera el descriptor de fichero obtenido cuando se abrió el fichero, dejándolo disponible para su uso posterior por el proceso. Se decrementa el número de referencias y, en su caso, de nopens. **Devuelve** 0 o -1 si fracasa, lo que ocurre, por ejemplo, si se intenta cerrar un descriptor ya cerrado.

Descripción:

- Si la llamada termina correctamente, se liberan los posibles cerrojos sobre el fichero fijados por el proceso y se anulan las operaciones pendientes de E/S asíncrona.
- Si nopens llega a 0 se liberan los recursos del sistema operativo ocupados por el fichero, incluyendo posibles proyecciones en memoria del mismo.

◙ off_t **lseek** (int fildes, off_t offset, int whence);

Este servicio permite cambiar el valor del puntero de posición de un fichero regular abierto, de forma que posteriores operaciones de E/S se ejecuten a partir de esa posición.

Los **argumentos** son los siguientes:

- `fildes`: descriptor de fichero.
- `offset`: desplazamiento (positivo o negativo).

- `whence`: base del desplazamiento.

Devuelve la nueva posición del puntero o -1 si fracasa. Solamente tiene sentido para ficheros regulares, por lo tanto si el descriptor se refiere a un terminal, *pipe*, *socket* o FIFO fracasa con errno=EPIPE.

Descripción:

- Coloca el puntero de posición asociado a `fildes` en el punto definido.
- La nueva posición, que **no puede ser negativa**, se calcula según el valor de `whence`, que puede tener los valores siguientes:
  - SEEK_SET: nueva posición = `offset` (`offset` no puede ser negativo en este caso)
  - SEEK_CUR: nueva posición = posición actual + `offset`
  - SEEK_END: nueva posición = final del fichero + `offset`
- Nos podemos **salir del tamaño** del fichero, lo que no implica aumentar el tamaño del fichero. Solamente se hará efectivo el aumento de tamaño si seguidamente se escribe en esa posición del fichero.

int **dup** (int fildes);

El servicio `dup` duplica un descriptor de fichero.

Argumentos:

- `fildes`: descriptor de fichero.

Devuelve un descriptor de fichero que comparte todas las propiedades del `fildes` o -1 si fracasa.

Descripción:

- Crea un nuevo descriptor de fichero que tiene en común con el anterior las siguientes propiedades:
  - Accede al mismo fichero.
  - Comparte el mismo puntero de posición.
  - El modo de acceso (lectura y/o escritura) es idéntico.
- El nuevo descriptor el primero libre de la tabla de descriptores del proceso.
- Se incrementa en 1 el número de referencias en la tabla intermedia.

La figura 5.66 muestra el efecto de ejecutar `fd2= dup(fd1);`

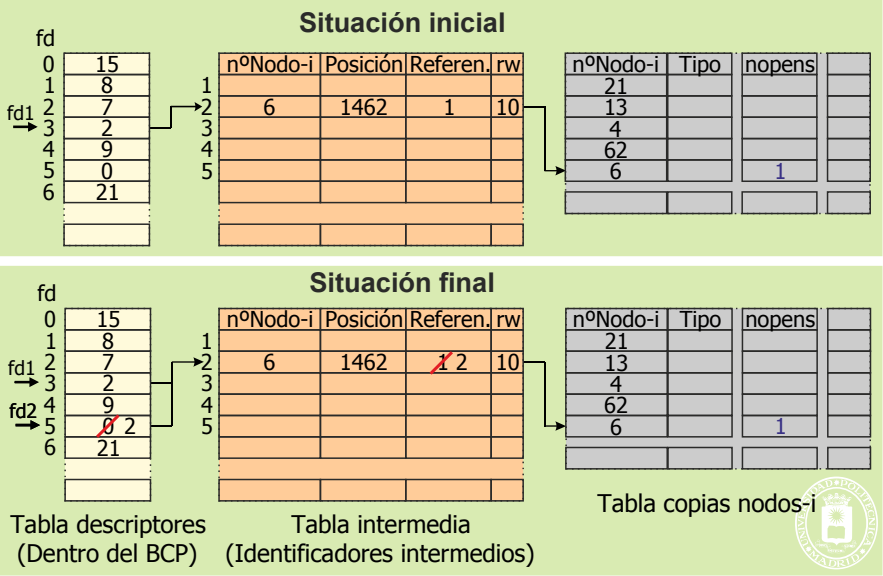


Figura 5.66 *Efecto de ejecutar el servicio: fd2= dup(fd1);*

int dup2(int oldfd, int newfd);

El servicio `dup2` duplica un descriptor de fichero. Se diferencia del `dup` en que se especifica el valor del nuevo descriptor.

Argumentos:

- `oldfd`: descriptor de fichero existente.
- `newfd`: nuevo descriptor de fichero.

Devuelve el nuevo descriptor de fichero `newfd` o -1 si fracasa.

Descripción:

- Crea un nuevo descriptor de fichero, cuyo número es `newfd`, que tiene en común con `oldfd` las siguientes propiedades:
  - Accede al mismo fichero.
  - Comparte el mismo puntero de posición.
  - El modo de acceso (lectura y/o escritura) es idéntico.

- Si `newfd` estaba abierto, lo **cierra** antes de realizar el duplicado.
- Se incrementa en 1 el número de referencias en la tabla intermedia.

- int **stat** (const char *path, struct stat *buf);
- int **fstat** (int fildes, struct stat *buf);
- int **lstat**(const char *path, struct stat *buf);

Los servicios `stat`, `fstat` y `lstat` permiten consultar los atributos de un fichero, si bien en el caso de `fstat` dicho fichero debe estar abierto.

Argumentos:

- `path`: nombre del fichero.
- `fildes`: descriptor de fichero.
- `buf`: puntero a un objeto de tipo `struct stat` donde se almacenará la información del fichero.

`lstat` se diferencia del `stat` en que devuelve el estado del propio enlace simbólico y no el del fichero al que apunta dicho enlace, como hace el `stat`.

Devuelve cero si tiene éxito o -1 si fracasa.

Descripción: obtiene información sobre un fichero y la almacena en una estructura de tipo `struct stat`, que se detalla a continuación:

```
struct stat {
mode_t     st_mode;      /* tipo de fichero + permisos */
ino_t      st_ino;       /* número del fichero */
dev_t      st_dev;       /* dispositivo */
nlink_t    st_nlink;     /* número de enlaces */
uid_t      st_uid;       /* UID del propietario */
gid_t      st_gid;       /* GID del propietario */
off_t      st_size;      /* número de bytes */
blksize_t  st_blksize;   /* tamaño bloque para I/O */
blkcnt_t   st_blocks;    /* nº de agrupaciones asignadas */
time_t     st_atime;     /* último acceso */
time_t     st_mtime;     /* última modificacion de datos */
time_t     st_ctime;     /* última modificacion del nodo-i*/
};
```

Existen unas macros que permiten comprobar del tipo de fichero aplicadas al campo **`st_mode`**:

S_ISDIR(s.st_mode)	Cierto si directorio
S_ISREG(s.st_mode)	Cierto si fichero regular
S_ISLNK(s.st_mode)	Cierto si enlace simbólico
S_ISCHR(s.st_mode)	Cierto si especial de caracteres
S_ISBLK(s.st_mode)	Cierto si especial de bloques
S_ISFIFO(s.st_mode)	Cierto si *pipe* o FIFO
S_ISSOCK(s.st_mode)	Cierto si *socket*

- int **fcntl** (int fildes, int cmd, /* arg */ ...);

UNIX especifica el servicio `fcntl` para controlar los distintos atributos de un fichero abierto. El descriptor de fichero abierto se especifica en `fildes`. La operación a ejecutar se especifica en `cmd` y los argumentos para dicha operación, cuyo número puede variar de una a otra, en `arg`. Los mandatos que se pueden especificar en esta llamada permiten duplicar el descriptor de fichero, leer y modificar los atributos del fichero, leer y modificar el estado del fichero, establecer y liberar cerrojos sobre el fichero y modificar las formas de acceso definidas en la llamada `open`. Los argumentos son específicos para cada mandato, por lo que se remite al lector a la información de ayuda del sistema operativo.

En la sección "6.10.11 Cerrojos en UNIX" se analiza el uso de este servicio para establecer cerrojos.

Otros servicios de ficheros.

El estándar UNIX describe detalladamente muchos más servicios para ficheros que los descritos previamente. Permite crear ficheros especiales para tuberías (`pipe`) y tuberías con nombre (`mkfifo`), truncarlo (`truncate`), realizar operaciones de E/S asíncronas (`aio_read` y `aio_write`) y esperar por ellas (`aio_suspend`) o cancelarlas (`aio_cancel`), etcétera Además define varias llamadas para gestionar dispositivos especiales, como terminales o líneas serie. Se remite al lector a la información de ayuda de su sistema operativo (mandato `man` en UNIX y LINUX) y a los manuales de descripción del estándar UNIX para una descripción detallada de dichos servicios.

5.16.2. Ejemplo de uso de servicios UNIX para ficheros

Para ilustrar el uso de los servicios de ficheros que proporciona UNIX, se presenta en esta sección un pequeño programa en lenguaje C que copia un fichero en otro. Los nombres de los ficheros origen y destino se reciben como parámetros de entrada. El programa ejecuta la siguiente secuencia de acciones:

1.- Abrir el fichero origen.
2.- Crear el fichero destino.
3.- Bloquear el fichero origen con acceso compartido y el destino con exclusivo.
4.- Mientras existan datos en el fichero origen, leer datos y añadirlos al fichero destino.
5.- Liberar los bloqueos.
6.- Cerrar los ficheros origen y destino.

El código fuente en lenguaje C que se corresponde al ejemplo propuesto se muestra en el Programa 5.7. Para ver más ejemplos de programación consulte el libro de Rockind.

Programa 5.7. Copia de un fichero sobre otro protegida con cerrojos.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/file.h>
#include <fcntl.h>
#include <stdio.h>

#define TAMANYO_ALM 1024            /* CONSEJO 9.1 */
main (int argc, char **argv)
{
   int    fd_ent, fd_sal;          /* identificadores de ficheros */
   char   almacen[TAMANYO_ALM];    /* almacén de E/S */
   int    n_read;          /* contador de E/S */
   /* Se comprueba que el número de argumentos sea el adecuado. En caso contrario, termina y
   devuelve un error */
   if (argc != 3) {
      fprintf (stderr, "Uso: copiar origen destino \n");
      exit(-1);
   }

   /* Abrir el fichero de entrada */
   fd_ent = open (argv[1], O_RDONLY);
   if (fd_ent < 0)
   {
      perror ("open");
      exit (-1);
   }
   /* Crear el fichero de salida. Modo de protección: rw-r--r--*/
   fd_sal = creat (argv[2], 0644);
   if (fd_sal < 0)
   {
      perror ("creat");
      exit (-1);
   }

/* Adquirir un cerrojo compartido sobre el fichero origen */
if (flock (fd_ent, LOCK_SH) == -1) {
      perror (“flock origen”); close(fd_ent); close(fd_sal);
      exit (-1);
}
/* Adquirir un cerrojo exclusivo sobre el fichero destino */
if (flock (fd_sal, LOCK_EX) == -1) {
      perror (“flock destino”);
      flock (fd_ent, LOCK_UN);
close(fd_ent); close(fd_sal);
      exit (-1);
}

   /* Bucle de lectura y escritura */
   while ((n_read = read (fd_ent, almacen, TAMANYO_ALM)) >0 )
   {
      /* Escribir el almacen al fichero de salida */
      if (write (fd_sal, almacen, n_read) < n_read)
      {
         perror ("write");
         close (fd_ent);
```

```
            close (fd_sal);
            exit (-1);
        } /* If */
    } /* While */
    if (n_read < 0)
    {
        perror ("read");
        close (fd_ent);
        close (fd_sal);
        exit (-1);
    }
/* Operación terminada. Desbloquear ficheros */
flock (fd_ent, LOCK_UN);
flock (fd_sal, LOCK_UN);
    /* Todo correcto. Cerrar ficheros */
    close (fd_ent);
    close (fd_sal);
    exit (0);
}
```

El programa 5.8 presenta el esquema de lo que hace el *shell* cuando se le pide que ejecute el mandato: `ls > fichero`

Este mandato incluye, como hace el *shell*, la creación de un proceso hijo y la redirección de la salida estándar del hijo al fichero de nombre `fichero`. Seguidamente el hijo pasa a ejecutar el programa ls.

Aclaración 5.4. Los mandatos del intérprete de mandatos de UNIX están escritos para leer y escribir de los descriptores estándar, por lo que se puede hacer que ejecuten sobre cualquier fichero sin más que redireccionar los descriptores de ficheros estándares.

La figura 5.67 muestra la evolución de las tablas de descriptores de ficheros del proceso padre y del hijo.

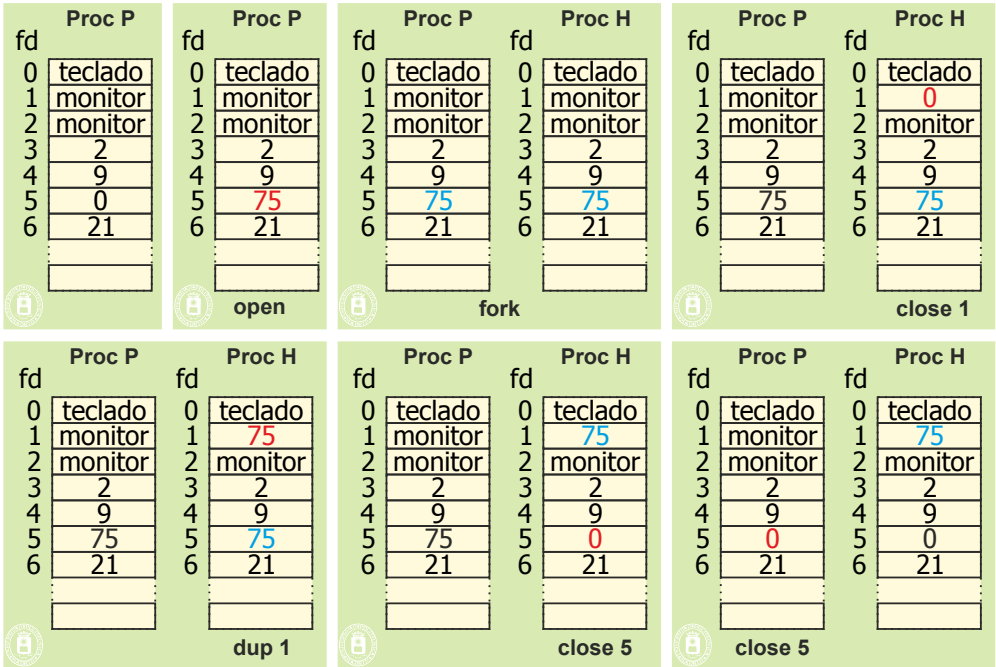


Figura 5.67 *Evolución de las tablas de descriptores de ficheros del programa 5.8.*

Programa 5.8. Redirección al ejecutar `.ls > fichero`

```
int main(void){
    pid_t pid;
    int status, fd;
    fd = open("fichero", O_WRONLY|O_CREAT|O_TRUNC, 0666);
    if (fd < 0) {perror("open"); exit(1);}
    pid = fork();
    switch(pid) {
        case -1: /* error */
            perror("fork"); exit(1);
```

```
        case 0: /* proceso hijo ejecuta "ls" */
            close(1);
            dup(fd); /* el descriptor se copia en el 1 */
            close(fd);
            execlp("ls","ls",NULL);
            perror("execlp"); exit(1);
        default: /* proceso padre */
            close(fd);
            while (pid != wait(&status));
    }
    return 0;
}
```

El programa 5.9 muestra un ejemplo de redirección de la entrada/salida estándar a un fichero y a la impresora. En este caso los nombres de los ficheros origen y destino se han definido como constantes en el código, aunque lo habitual es recibirlos como parámetros. El programa ejecuta la siguiente secuencia de acciones:

1. Cierra la entrada estándar y abre el fichero origen de datos. Esto asigna al fichero el descriptor 0, correspondiente a `stdin`.
2. Cierra la salida estándar y abre la impresora destino de los datos. Esto asigna al fichero el descriptor 1, correspondiente a `stdout`.
3. Lee los datos de `stdin` y los vuelca por `stdout`.
4. Cerrar `stdin` y `stdout` y termina. ¡Cuidado con esta operación! Si se hace antes de que el proceso vaya a terminar, se queda sin entrada y salida estándar.

Consejo 5.2 El tamaño del almacén de entrada/salida se usa en este caso para indicar el número de caracteres a leer o escribir en cada operación. El rendimiento de estas operaciones se ve muy afectado por este parámetro. Un almacén de un byte, daría un rendimiento pésimo porque habría que hacer una llamada al sistema cada vez y, en algunos casos, múltiples accesos a disco. Para poder efectuar operaciones de entrada/salida con accesos de tamaño pequeño y rendimiento aceptable, algunos lenguajes de programación, como C y C++, incluyen una biblioteca de objetos de entrada/salida denominada *stream*. Los métodos de esta biblioteca hacen almacenamiento intermedio de datos sin que el usuario sea consciente de ello. De esta forma, hace operaciones de entrada/salida de tamaño aceptable (4 u 8 KiB). Si se encuentra alguna vez con accesos de E/S pequeños, no dude en usar esta biblioteca. Este consejo es válido para entornos que utilizan el estándar C y C++.

Programa 5.9. Redirección de nombres estándar a un fichero y a la impresora.

```
/* Mandato: cat < hijo > /dev/lp */

#include <fcntl.h>

#define STDIN 0
#define STDOUT 1
#define TAMANYO_ALM 1024

main (int argc, char **argv)
{
    int     fd_ent, fd_sal;          /* identificadores de ficheros */
    char    almacen[TAMANYO_ALM];    /* almacén de E/S */
    int     n_read;          /* contador de E/S */


    /* Cerrar entrada estándar */
    fd_ent = close (STDIN);
    if (fd_ent < 0)
    {
        perror ("close STDIN");          /* Escribe por STDERR */
        exit (-1);
    }
    /* Abrir el fichero de entrada */
    /* Como el primer descriptor libre es 0 (STDIN), se asigna a fd_ent */
    fd_ent = open ("hijo", O_RDONLY);
    if (fd_ent < 0)
    {
        perror ("open hijo");
        exit (-1);
```

```
    }
    /* Cerrar salida estándar */
    fd_sal = close (STDOUT);
    if (fd_sal < 0)
    {
        perror ("close STDIN");
        exit (-1);
    }
    /* Abrir la impresora como dispositivo de salida. */
    /* Como el primer descriptor libre es 1 (STDOUT), se asigna a fd_sal */
    fd_sal = open ("/dev/lp", O_WRONLY);
    if (fd_sal < 0)
    {
        perror ("open lp");
        exit (-1);
    }

    /* Bucle de lectura y escritura */
    while ((n_read = read (STDIN, almacen, TAMANYO_ALM)) >0 )
    {
        /* Escribir el almacen al fichero de salida */
        if (write (STDOUT, almacen, n_read) < n_read)
        {
            perror ("write stdout");
            close (fd_ent);
            close (fd_sal);
            exit (-1);
        } /* If */
    } /* While */
    if (n_read < 0)
    {
        perror ("read hijo");
        close (fd_ent);
        close (fd_sal);
        exit (-1);
    }
    /* Todo correcto. Cerrar ficheros estándar*/
    close (STDIN);
    close (STDOUT);
    exit (0);
}
/* Si no puede abrir /dev/lp por permisos de protección, use cualquier otro fichero creado por usted */
```

El programa 5.10 muestra un ejemplo de consulta de atributos del fichero "`datos`" y de escrituras aleatorias en el mismo. El programa ejecuta la siguiente secuencia de acciones:

1. Antes de abrir el fichero comprueba si tiene permisos de lectura. Si no es así, termina.
2. Abre el fichero para escritura.
3. Salta en el fichero desde el principio del mismo y escribe.
4. Salta más allá del final del fichero y escribe.
5. Cierra el fichero y termina.

Programa 5.10. Consulta de atributos y escrituras aleatorias en un fichero.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#define STDIN 0
#define STDOUT 1

#define TAMANYO_ALM 1024
main (int argc, char **argv)
{
    int     fd_sal;                         /* identificador de fichero */
```

```
    char  almacen[TAMANYO_ALM];      /* almacén de E/S */
    int   n_read;                    /* contador de E/S */
    struct stat atributos;    /* estructura para almacenar atributos */


    /* Control de permiso de escritura para el usuario*/
    fd_sal = stat ("datos", &atributos);
    if ((S_IWUSR & atributos.st_mode) != S_IWUSR)
    {
       perror ("no se puede escribir en datos");
       exit (-1);
    }

    /* Apertura del fichero */
    fd_sal = open ("datos", O_WRONLY);
    if (fd_sal < 0)
    {
       perror ("open datos");
       exit (-1);
    }
    /* Escritura con desplazamiento 32 desde el inicio*/
    if ((n_read = lseek (fd_sal, 32, SEEK_SET)) >0 )
    {
       /* Escribir al fichero de salida */
       if (write (fd_sal, "prueba 1", strlen("prueba 1")) < 0)
       {
          perror ("write SEEK_SET");
          close (fd_sal);
          exit (-1);
       } /* If */
    }
    /* Escritura con desplazamiento 1024 desde posición actual */
    if ((n_read = lseek (fd_sal, 1024, SEEK_CUR)) >0 )
    {
    /* Escribir al fichero de salida */
       if (write (fd_sal, "prueba 2", strlen("prueba 2")) < 0)
       {
          perror ("write SEEK_CUR");
          close (fd_sal);
          exit (-1);
       } /* If */
    }
    /* Escritura con desplazamiento 1024 desde el final */
    if ((n_read = lseek (fd_sal, 1024, SEEK_END)) >0 )
    {
       /* Escribir al fichero de salida */
       if (write (fd_sal, "prueba 3", strlen("prueba 3")) < 0)
       {
          perror ("write SEEK_END");
          close (fd_sal);
          exit (-1);
       } /* If */
    }

    /* Todo correcto. Cerrar fichero */
    close (fd_sal);
    exit (0);
  }
  /* Para ver el efecto del último lseek, haga el mismo ejemplo sin el write de prueba 3 */
```

La figura 5.68 muestra el efecto de las operaciones de salto. Observe que se generan zonas vacías (agujeros) en medio del fichero cuando se hace un `lseek` más allá del fin del fichero y luego una escritura en esa posición. El agujero resultante no tiene inicialmente ningún valor, aunque cuando el usuario intenta acceder a esa posición, se le asignan bloques de disco y se le devuelven datos rellenos de ceros.

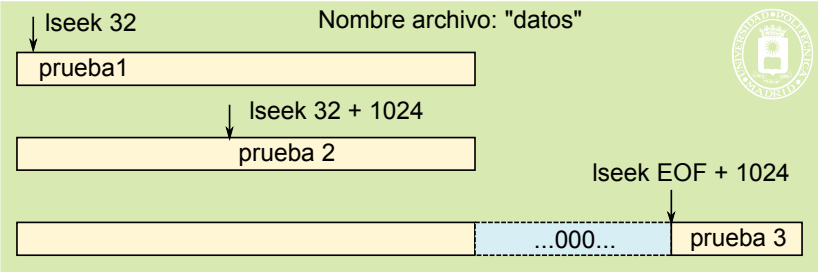


Figura 5.68 *Efecto de las operaciones de salto del programa 5.10.*

5.16.3. Servicios Windows para ficheros

Windows proporciona una visión lógica de fichero equivalente a una tira secuencial de bytes. Para acceder al fichero, se mantiene un apuntador de posición, a partir del cual se ejecutan las operaciones de lectura y escritura sobre el fichero. Para identificar a un fichero, el usuario usa nombres jerárquicos, como por ejemplo `C:\users\miguel\datos`.

Cada fichero se define como un objeto dentro del núcleo de Windows. Por ello, cuando se abre un fichero, se crea en memoria un objeto fichero y se le devuelve al usuario un manejador (`HANDLE`) para ese objeto.

Nuevas operaciones de apertura dan lugar a la creación de nuevos objetos en memoria y de sus correspondientes manejadores. Al igual que en UNIX, los procesos pueden obtener manejadores de objetos estándar para entrada/salida (`GetStdHandle`, `SetStdHandle`). El objetivo de estos descriptores estándares es poder escribir programas que sean independientes de los ficheros sobre los que han de trabajar. Las aplicaciones de consola reciben los tres manejadores para los ficheros estándares de entrada/salida por defecto (`stdin, stdout, stderr`), aunque las gráficas no tienen estos descriptores estándares. Estos manejadores se conceden a nivel de sistema y no están asociados al proceso que abre el fichero como en UNIX, por lo que son únicos a nivel de sistema.

Cada manejador de objeto de tipo fichero incluye parte de la información del registro que describe al fichero y además incluye una relación de las operaciones posibles sobre dicho fichero. Es interesante resaltar que el apuntador de posición del fichero para cada instanciación del objeto se incluye en esta estructura. Cuando se cierra un fichero, se libera la memoria correspondiente a la representación de ese objeto fichero. Una característica de FS es que puede tener varios flujos independientes sobre un fichero de forma concurrente. Si hay conflictos, la concurrencia sobre cada objeto fichero se resuelve mediante semáforos que serializan las operaciones que afectan a dicho objeto.

Usando servicios de Windows, se pueden consultar los atributos de un fichero. Estos atributos son una parte de la información existente en el manejador del objeto. Entre ellos se encuentran el nombre, tipo y tamaño del fichero, el sistema de ficheros al que pertenece, su dispositivo, tiempos de creación y modificación, información de estado, apuntador de posición del fichero, información de sobre cerrojos, protección, etcétera El modo de protección es especialmente importante porque permite controlar el acceso al fichero por parte de los usuarios. Cada objeto fichero tiene asociado un descriptor de seguridad.

A continuación se muestran los servicios más comunes para gestión de ficheros que proporciona Windows. Como se puede ver, son similares a los de UNIX, si bien los prototipos de las funciones que los proporcionan son bastante distintos.

- HANDLE **CreateFile**(LPCSTR lpFileName, DWORD dwDesiredAccess, DWORD dwShareMode, LPVOID lpSecurityAttributes, DWORD CreationDisposition, DWORD dwFlagsAndAttributes, HANDLE hTemplateFile);

Este servicio permite crear, o abrir, un nuevo fichero. Su efecto es la creación, o apertura, de un fichero con nombre `lpFileName`, modo de acceso `dwDesiredAccess` (`GENERIC_READ`, `GENERIC_WRITE`) y modo de compartición `dwShareMode` (`NO_SHARE`, `SHARE_READ`, `SHARE_WRITE`). `lpSecurityAttributes` define los atributos de seguridad para el fichero. La forma de crear o abrir el fichero se especifica en `CreationDisposition`. Modos posibles son: `CREATE_NEW`, `CREATE_ALWAYS`, `OPEN_EXISTING`, `OPEN_ALWAYS`, `TRUNCATE_EXISTING`. Estas primitivas indican que se debe crear un fichero si no existe, sobrescribirlo si ya existe, abrir uno ya existente, crearlo y abrirlo si no existe y truncarlo si ya existe, respectivamente. El parámetro `dwFlagsAndAttributes` especifica acciones de control sobre el fichero. Por su extensión, se refiere al lector a manuales específicos de Windows. `hTemplateFile` proporciona una plantilla con valores para la creación por defecto. Si el fichero ya está abierto, se incrementa el contador de aperturas asociado al manejador. El resultado de la llamada es un manejador al nuevo fichero. En caso de que haya un error devuelve un manejador nulo.

- BOOL **DeleteFile** (LPCTSTR lpszFileName);

Este servicio permite borrar un fichero indicando su nombre. Esta llamada libera los recursos asignados al fichero. El fichero no se podrá acceder nunca más. El nombre del fichero a borrar se indica en `lpszFileName`. Devuelve `TRUE` en caso de éxito o `FALSE` en caso de error.

- BOOL **CloseHandle** (HANDLE hObject);

La llamada `CloseHandle` libera el descriptor de fichero obtenido cuando se abrió el fichero, dejándolo disponible para su uso posterior. Si la llamada termina correctamente, se liberan los posibles cerrojos sobre el fichero fijados por el proceso, se invalidan las operaciones pendientes y se decrementa el contador del manejador del objeto. Si el

contador del manejador es cero, se liberan los recursos ocupados por el fichero. Devuelve `TRUE` en caso de éxito o `FALSE` en caso de error.

- BOOL **ReadFile** (HANDLE hFile, LPVOID lpbuffer, DWORD nNumberOfBytesToRead, LPDWORD lpNumberOfBytesRead, LPVOID lpOverlapped);

Este servicio permite a un proceso leer datos de un fichero abierto y copiarlos a su espacio de memoria. El manejador del fichero se indica en `hFile`, la posición de memoria donde copiar los datos se especifica en el argumento `lpbuffer` y el número de bytes a leer se especifica en `lpNumberOfBytesToRead`. La lectura se lleva a cabo a partir de la posición actual del apuntador de posición del fichero. Si la llamada se ejecuta correctamente, devuelve el número de bytes leídos realmente, que pueden ser menos que los pedidos, en `lpNumberOfBytesRead` y se incrementa el apuntador de posición del fichero en esa cantidad. `lpOverlapped` sirve para indicar el manejador de evento que se debe usar cuando se hace una lectura no bloqueante (asíncrona). Si ese parámetro tiene algún evento activado, cuando termina la llamada se ejecuta el manejador de dicho evento. Devuelve `TRUE` en caso de éxito o `FALSE` en caso de error.

- BOOL **WriteFile** (HANDLE hFile, LPVOID lpbuffer, DWORD nNumberOfBytesToWrite, LPDWORD lpNumberOfBytesWritten, LPVOID lpOverlapped);

Esta llamada permite a un proceso escribir en un fichero una porción de datos existente en el espacio de memoria del proceso. El manejador del fichero se indica en `hFile`, la posición de memoria desde donde copiar los datos se especifica en el argumento `lpbuffer` y el número de bytes a escribir se especifica en `lpNumberOfBytesToWrite`. La escritura se lleva a cabo a partir de la posición actual del apuntador de posición del fichero. Si la llamada se ejecuta correctamente, devuelve el número de bytes escritos realmente, que pueden ser menos que los pedidos, en `lpNumberOfBytesWritten` y se incrementa el apuntador de posición del fichero en esa cantidad. `lpOverlapped` sirve para indicar el manejador de evento que se debe usar cuando se hace una lectura no bloqueante (asíncrona). Si ese parámetro tiene algún evento activado, cuando termina la llamada se ejecuta el manejador de dicho evento. Devuelve `TRUE` en caso de éxito o `FALSE` en caso de error.

- DWORD **SetFilePointer** (HANDLE hFile, LONG lDistanceToMove, LONG FAR *lpDistanceToMoveHigh, DWORD dwMoveMethod);

Esta llamada permite cambiar el valor del apuntador de posición de un fichero abierto previamente, de forma que operaciones posteriores de E/S se ejecuten a partir de esa posición. El manejador de fichero se indica en `hFile`, el desplazamiento (positivo o negativo) se indica en `lDistanceToMove` y el lugar de referencia para el desplazamiento se indica en `dwMoveMethod`. Hay tres posibles formas de indicar la referencia de posición: `FILE_BEGIN`, el valor final del apuntador es `lDistanceToMove`; `FILE_CURRE`, el valor final del apuntador es el valor actual más (o menos) `lDistanceToMove`; `FILE_END`, el valor final es la posición de fin de fichero más (o menos) `lDistanceToMove`. Si la llamada se ejecuta correctamente, devuelve el nuevo valor del apuntador de posición del fichero (Consejo 9.2).

Consejo 5.3 La llamada **`SetFilePointer`** permite averiguar la longitud del fichero especificando `FILE_END` y desplazamiento `0`.

- DWORD **GetFileAttributes**(LPCTSTR lpszFileName);
- BOOL **GetFileTime** (HANDLE hFile, LPFILETIME lpftCreation, LPFILETIME lpftLastAccess, LPFILETIME lpftLastWrite);
- BOOL **GetFileSize** (HANDLE hFile, LPDWORD lpdwFileSize);

Windows especifica varios servicios para consultar los distintos atributos de un fichero, si bien la llamada más genérica de todas es `GetFileAttributes`. La llamada `GetFileAttributes` permite obtener un subconjunto reducido de los atributos de un fichero que indican si es temporal, compartido, etcétera Dichos atributos se indican como una máscara de bits que se devuelve como resultado de la llamada. La llamada `GetFileTime` permite obtener los atributos de tiempo de creación, último acceso y última escritura de un fichero abierto. La llamada `GetFileSize` permite obtener la longitud de un fichero abierto.

- BOOL **LockFile** (HANDLE hFile, DWORD dwFileOffsetLow, DWORD dwFileOffsetHigh, DWORD dwNumberofBytestoLockLow, WORD dwNumberofBytestoLockHigh);
- BOOL **LockFileEx** (HANDLE hFile, DWORD dwFlags, DWORD dwFileOffsetLow, WORD dwFileOffsetHigh, DWORD dwNumberofBytestoLockLow, DWORD dwNumberofBytestoLockHigh);
- BOOL **UnlockFile** (HANDLE hFile, DWORD dwFileOffsetLow, DWORD dwFileOffsetHigh, DWORD dwNumberofBytestoLockLow, DWORD dwNumberofBytestoLockHigh);

La llamada de Windows es equivalente al `flock` de UNIX es `LockFile`. Esta función bloquea el fichero que se especifica para acceso exclusivo del proceso que la llama. Si el bloqueo falla no deja bloqueado el proceso, sino que retorna con un error. Si se quiere un bloqueo con espera, se debe usar `LockFileEx` que puede funcionar de forma síncrona o asíncrona. Para liberar el bloqueo se usa la llamada `UnlockFile`. donde `dwFileOffsetLow` y `dwFileOffsetHigh` indican la posición origen del bloqueo y `dwNumberofBytestoLockLow` y `dwNumberofBytestoLockHigh` indican la longitud del bloqueo. Para bloquear todo el fichero basta con poner 0 y `FileLength`.

Otros servicios de ficheros.

Windows describe detalladamente muchos más servicios para ficheros que los descritos previamente. Permite manejar descriptores de ficheros de entrada/salida estándar (`GetStdHandle`, `SetStdHandle`), trabajar con la consola (`SetConsoleMode`, `ReadConsole`, `FreeConsole`, `AllocConsole`), copiar un fichero (`CopyFile`), renombrarlo (`MoveFile`) y realizar operaciones con los atributos de un fichero (`SetFileTime`, `GetFileTime`, `GetFileAttributes`).

Se remite al lector a la información de ayuda del sistema operativo Windows (botón `help`), y a los manuales de descripción de Windows, para una descripción detallada de dichos servicios.

5.16.4. Ejemplo de uso de servicios Windows para ficheros

Para ilustrar el uso de los servicios de ficheros que proporciona el sistema operativo, se presenta en esta sección el ejemplo de copia de un fichero sobre otro, pero usando llamadas de Windows.

El código fuente en lenguaje C que se corresponde al ejemplo propuesto se muestra en el programa 5.11.

Programa 5.11. Copia de un fichero sobre otro usando llamadas Windows.

```
#include <windows.h>
#include <stdio.h>

#define TAMANYO_ALM 1024

int main (int argc, LPTSTR argv )
{
   HANDLE hEnt, hSal;
   DWORD nEnt, nSal, dwPos;
   CHAR Almacen [TAMANYO_ALM];
   /* Se comprueba que el número de argumentos sea el adecuado. En caso contrario, termina y devuelve error */
   if (argc != 3) {
      fprintf (stderr, "Uso: copiar fichero1 fichero2\n");
      return -1;
   }
   /* Abrir el fichero origen. Opción: OPEN_EXISTING */
   hEnt = CreateFile (argv [1], GENERIC_READ, FILE_SHARE_READ, NULL,
         OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
   if (hEnt == INVALID_HANDLE_VALUE) {
      fprintf (stderr,      "No se puede abrir el fichero de entrada. Error: %x\n",  GetLastError ());
   /* CONSEJO 9.3 */
      return -1;
   }
   /* Crear el fichero destino. Opción: CREATE_ALWAYS */
   hSal = CreateFile (argv [2], GENERIC_WRITE, 0, NULL,
         CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
   if (hSal == INVALID_HANDLE_VALUE) {
      fprintf (stderr,      "No se puede abrir el fichero de salida. Error: %x\n", GetLastError ());
      return -1;
   }
   /* Bucle de lectura del fichero origen y escritura en el destino. Se ejecuta mientras no se llegue al final del
   fichero origen. Se bloquea parte a parte del fichero en bloques de TAMANYO_ALM.
   Solo se prueba el bloqueo en el primer byte. Estrictamente, habría  Que ver longitud y bloquear todo */
   LockFile(hEnt, 1, 0, 1, 0);
    LockFile(hSal, 1, 0, 1, 0);
   while (ReadFile (hEnt, Almacen, TAMANYO_ALM, &nEnt, NULL)
      &&   nEnt > 0) {
      /* Sólo se escriben los datos que se han leído realmente */
      dwPos = SetFilePointer(hSal, 0, NULL, FILE_END);
      WriteFile (hSal, Almacen, nEnt, &nSal, NULL);
      if (nEnt != nSal) {
         fprintf (stderr,"Error fatal de escritura. Error: %x\n", GetLastError ());
         return -1;
      }
   }
   /* Desbloquear ficheros y cerrar */
   UnlockFile(hEnt, 1, 0, 1, 0);
```

```
    UnlockFile(hSal, 1, 0, 1, 0);
    CloseHandle (hEnt);
    CloseHandle (hSal);
    return 0;
}
```

Consejo 5.4 La función `GetLastError` permite imprimir el código de error que el sistema operativo Windows devuelve en caso de error de una llamada al sistema. Esta llamada es muy útil para depurar programas.

Para ilustrar el uso de las funciones de conveniencia de servicios de ficheros que proporciona Windows, se resuelve también en esta sección el ejemplo de copia de un fichero sobre otro con la llamada `CopyFile`. El código fuente en lenguaje C que se corresponde al ejemplo propuesto se muestra en el programa 5.12. Como se puede ver, comparando con el programa 5.11, esta función se encarga de abrir y crear los ficheros, leer y escribir los datos y cerrar los ficheros (Prestaciones 5.1).

Prestaciones 5.1. Normalmente las funciones complejas, como `CopyFile` , que proporciona el sistema operativo son más eficientes que su equivalente programado. Existen varias razones para este comportamiento, pero dos muy importantes son que cada llamada al sistema operativo tiene un coste de ejecución alto y que a nivel interno el sistema operativo usa directamente servicios optimizados e incluso instrucciones máquina complejas que resuelven las tareas muy rápidamente.

Programa 5.12. Copia de un fichero sobre otro usando funciones de conveniencia de Windows.

```
#include <windows.h>
#include <stdio.h>

int main (int argc, LPCTSTR argv )
{
    if (argc != 3) {
        fprintf (stderr, "Uso: copiar fichero1 fichero2 \n"));
        return -1;
    }
    if (!CopyFile (argv[1], argv[2], FALSE)) {
        fprintf (stderr, "Error de copia: %x \n", GetLastError());
    }
    return 0;
}
```

5.16.5. Servicios UNIX de directorios

Un directorio en UNIX tiene la estructura lógica de una tabla, cada una de cuyas entradas contiene una estructura `dirent` que relaciona un nombre de fichero con su nodo-i (como se describe en la sección "5.6.4 Metainformación del fichero", página 219). Para facilitar la gestión de las entradas de directorio, el estándar UNIX define los servicios específicos que debe proporcionar un sistema operativo para gestionar los directorios. En general se ajustan a los servicios genéricos descritos en la sección anterior. A continuación se describen los servicios de directorios más comunes del estándar UNIX.

- DIR ***opendir** (const char *dirname);

La función `opendir` abre el directorio de nombre especificado en la llamada y devuelve un identificador de directorio. El apuntador de posición indica a la primera entrada del directorio abierto. En caso de error devuelve `NULL`.

Requiere **permisos** de lectura de `dirname`.

- struct dirent ***readdir** (DIR *dirp);

Este servicio permite leer de un directorio abierto, obteniendo como resultado la **siguiente** entrada del mismo. El servicio **devuelve** Un puntero a un objeto del tipo `struct dirent`, que contiene un registro de directorio o NULL si hubo error o se ha llegado al final del directorio. Nunca devuelve entradas cuyos nombres están vacíos.

- int **closedir** (DIR *dirp);

Un directorio abierto, e identificado por `dirp`, puede ser cerrado ejecutando este servicio. En caso de éxito devuelve 0 y -1 en caso de error. El servicio cierra la asociación entre `dirp` y la secuencia de entradas de directorio.

- void **rewindir** (DIR *dirp);

Este servicio sitúa el puntero de posición del directorio en el primer registro o entrada.

◙ int **mkdir** (const char *path, mode_t mode);

El servicio `mkdir` permite crear un nuevo directorio en UNIX. Los **argumentos** son los siguientes:

- `path`: nombre del directorio
- `mode`: bits de protección

El servicio **devuelve**: Cero en caso de éxito y -1 si fracasa.

Permisos: El servicio requiere permisos de acceso desde el raíz hasta el directorio donde se añade `path`, así como permisos de escritura en dicho directorio.

Descripción: Crea el directorio especificado en `path` con el modo de protección especificado en `mode`. El dueño del directorio es el dueño del proceso que crea el directorio.

◙ int **rmdir** (const char *path);

El estándar UNIX permite borrar un directorio especificando su nombre.

Esta llamada **devuelve** cero en caso de éxito y -1 en caso de error.

Descripción:

- El directorio se borra únicamente cuando está vacío. Para borrar un directorio no vacío es obligatorio borrar antes todas sus entradas.
- El comportamiento de la llamada no está definido cuando se intenta borrar el directorio raíz o el directorio de trabajo.
- Si el contador de enlaces del directorio es mayor que cero, se decrementa dicho contador y se borra la entrada del directorio, pero no se liberan los recursos del mismo hasta que el contador de enlaces sea cero.

Permisos: El servicio requiere permisos de acceso desde el raíz hasta el directorio donde se elimina `path`, así como permisos de escritura en dicho directorio que contiene `path`.

◙ int **link** (const char *existing, const char *new);

El servicio `link` crea un enlace físico, añadiendo una entrada en el correspondiente directorio.

Argumentos:

- `existing`: nombre del fichero existente, que debe existir.
- `new`: nombre de la nueva entrada que será un enlace al fichero existente. No puede existir previamente

Devuelve: Cero si tiene éxito o -1 si fracasa.

Descripción:

- Crea un nuevo enlace físico para un fichero existente de su mismo sistema de ficheros.
- El sistema no registra cuál es el enlace original.
- `existing` no debe ser el nombre de un directorio salvo que se tenga privilegio suficiente y la implementación soporte el enlace de directorios.
- Las entradas «.» y «..» son enlaces físicos de directorio, pero controlados por el sistema operativo. Por tanto, el número de enlaces de un directorio es de: 1 en el directorio padre, 1 en el propio directorio «..» y 1 en cada directorio hijo «.».

Requiere **permisos** de búsqueda para `existing` y de escritura en el directorio donde esté `new`.

Recordemos que los servicios `open` y `creat` también añaden una entrada de directorio, cuando realmente crean un fichero.

◙ int **symlink** (const char *existing, const char *new);

El servicio `symlink` crea un enlace simbólico, añadiendo una entrada en el correspondiente directorio.

Argumentos:

- `existing`: nombre del fichero existente. No se comprueba que exista.
- `new`: nombre de la nueva entrada que será un enlace al fichero existente. No puede existir previamente.

Devuelve: Cero si tiene éxito o -1 si fracasa.

Descripción:

- Crea un nuevo enlace simbólico para un fichero o directorio que puede existir o no existir y que puede estar en otro sistema de ficheros.
- Requiere permiso de búsqueda y de escritura en el directorio donde esté `new`.
- El fichero o directorio puede desaparecer dejando el enlace "colgado".
- Se pueden crear ciclos en el árbol de directorio, lo que da lugar a un error de acceso.
- Al abrir un fichero (no al ejecutar el `symlink`) se comprueban también los derechos de todo el camino definido en el enlace simbólico.
- Una vez creado el enlace, la gran mayoría de los servicios que utilicen `new` como argumento se realizarán realmente sobre `existing`. Por ejemplo el `chmod` se hará sobre `existing`

◙ int **unlink** (const char *path);

Este servicio permite eliminar el nombre `path` de un fichero.

Devuelve: Cero si se ejecuta con éxito o -1 si fracasa.

Descripción:

- Elimina la entrada del directorio y decrementa el nº de enlaces del fichero correspondiente.
- Cuando el número de enlaces es igual a cero:
  - Si ningún proceso lo mantiene abierto, se marca como libre el espacio y nodo-i ocupado por el fichero en los correspondientes mapas de bits.
  - Si algún proceso lo mantiene abierto, se conserva el fichero hasta que lo cierren todos.
- Si se hace sobre un enlace simbólico se decrementa el nº de enlaces del nodo-i simbólico. Cuando llega a 0 se libera dicho nodo-i. No se hace nada sobre el fichero enlazado.

◙ int **chdir** (const char *path);

Este servicio permite cambiar el directorio de trabajo al valor especificado por `path`.

Devuelve: Cero si tiene éxito o -1 si fracasa.

Descripción: Modifica el directorio actual o de trabajo. Los nombre relativos se expresan a partir de este directorio.

Requiere **permisos** de búsqueda para `path`

◙ int **`rename`**(char *old, char *new);

Este servicio permite cambiar el nombre de un fichero o directorio.

Argumentos:

- `old`: nombre del fichero o directorio existente, que debe existir.
- `new`: nuevo nombre del fichero o directorio. No puede existir previamente.

Devuelve: Cero si tiene éxito o -1 si fracasa.

Descripción:

- Cambia el nombre del fichero `old` al nuevo nombre `new`.
- Si `old` y `new` especifican rutas distintas, el cambio de nombre implica mover el fichero de un punto del directorio a otro.

Requiere permiso de búsqueda y de escritura en el o los dos directorios que contienen `old` y `new`.

◙ char ***`getcwd`**(char *buf, size_t size);

Este servicio obtiene el nombre del directorio actual o de trabajo.

Argumentos:

- `buf`: dirección del espacio donde se va a almacenar el nombre del directorio actual.
- `size`: longitud en bytes de dicho espacio.

Devuelve un puntero a buf o NULL si fracasa (e.g. si el nombre ocupa más de size).

◙ int **`mount`**(const char *source, const char *target, const char *filesystemtype, unsigned long mountflags, const void *data);

Este servicio monta un sistema de ficheros (volumen o dispositivo).

Argumentos:

- `source`: sistema de ficheros que se monta, generalmente un dispositivo como /dev/cdrom, /dev/hdb0, /dev/sda1 o /home/imagen.iso.
- `target`: directorio sobre el que se monta.
- `filesystemtype`: tipo de sistema de ficheros, como "minix", "ext2", "ext3", "ext4", "msdos", "vfat", "proc" o "nfs".
- `ountflags`: Opciones como:
  - MS_NOEXEC enmascara los bits de ejecución del sistema montado.
  - MS_NOSUID enmascara los bits SUID y SGID del sistema montado.
  - MS_RDONLY montado para lectura solamente.
- `data` : opciones que dependen del tipo de sistema de ficheros.

Devuelve: cero si tiene éxito o -1 si fracasa.

Descripción: monta un dispositivo en un directorio del árbol de nombres existente.

Requiere **privilegios** de **superusuario**.

No confundir este servicio con el mandato del *shell* mount. El fichero /etc/fstab contiene los sistemas de ficheros disponibles con las opciones de montado para el **mandato** mount.

◙ int **`umount`**(const char *target);

Este servicio desmonta un sistema de ficheros.

Argumento: `target` : directorio que se desmonta.

Devuelve: cero si tiene éxito o -1 si fracasa.

Descripción:

- Desmonta un sistema de ficheros.
- Si hay ficheros abiertos no desmonta y devuelve error.

Requiere **privilegios** de superusuario.

Otros servicios UNIX

Es interesante resaltar la inexistencia de una llamada para escribir en un directorio, equivalente a `readdir`. Esto es así para que el sistema operativo pueda garantizar la coherencia del árbol de directorios, permitiendo únicamente añadir o borrar entradas, y cambiar nombres, a través de llamadas específicas para ello. (recordatorio 5.1)

Recordatorio 5.1. Como se dijo en secciones anteriores, los directorios se suelen implementar mediante ficheros. Por ello, aunque existen las llamadas específicas descritas para manejar directorios, se pueden obtener los mismos resultados usando llamadas de acceso a ficheros y manipulando las estructuras de datos de los directorios.

5.16.6. Ejemplo de uso de servicios UNIX para directorios

En esta sección se presentan dos pequeños programas en lenguaje C que muestran las entradas de un directorio, cuyo nombre se recibe como argumento.

El programa 5.13 muestra dichas entradas por la salida estándar del sistema. Para ello, el programa ejecuta la siguiente secuencia de acciones:

1.- Muestra el directorio actual.
2.- Imprime el directorio cuyo contenido se va a mostrar.
3.- Abre el directorio solicitado.
4.- Mientras existan entradas en el directorio, lee una entrada e imprime el nombre de la misma.
4.- Cierra el directorio.

Programa 5.13. Programa que muestra las entradas de un directorio usando llamadas UNIX.

```
#include <sys/types.h>
#include <dirent.h>
#include <stdio.h>
#include <error.h>

#define TAMANYO_ALM 1024

void main (int argc, char **argv)
{
    DIRP *dirp;
    struct dirent *dp;
    char almacen[TAMANYO_ALM];
    /* Comprueba que el número de parámetros sea el adecuado. En caso contrario, termina con un error */
    if (argc != 2) {
        fprintf (stderr, "Uso: mi_ls directorio\n");
        exit(1);
    }
    /* Obtener e imprimir el directorio actual */
    getcwd (almacen, TAMANYO_ALM);
    printf ("Directorio actual: %s \n", almacen);
    /* Abrir el directorio recibido como argumento */
    dirp = opendir (argv[1]);
    if (dirp == NULL)
    {
        fprintf (stderr, "No se pudo abrir el directorio: %s \n", argv[1]);
        perror();
        exit (1);
    }
    else
    {
        printf ("Entradas en el directorio: %s \n", argv[1]);
        /* Leer el directorio entrada a entrada */
        while ((dp = readdir(dirp)) != NULL)
        printf ("%s\n", dp->d_name);          /* Imprimir nombre */
        closedir (dirp);
    }
    exit (0);
}
```

Observe que la ejecución de este programa ejemplo no cambia el directorio de trabajo actual. Por ello, si se usan nombres absolutos, y se tienen los permisos adecuados, se pueden ver los contenidos de cualquier directorio sin movernos del actual. Además del nombre, la estructura de directorio incluye otra información que puede verse sin más que añadir las instrucciones para imprimir su contenido. El mandato `ls` de UNIX se implementa de forma similar al ejemplo.

El programa 5.14 copia el contenido del directorio a un fichero, cuyo nombre se recibe como parámetro. Para ello, ejecuta acciones similares a las del Programa 9.6, pero además redirige el descriptor de salida estándar hacia dicho fichero. El mandato `ls > "fichero"` de UNIX se implementa de forma similar al ejemplo.

Programa 5.14. Programa que copia las entradas de un directorio a un fichero.

```
#include <sys/types.h>
#include <dirent.h>
#include <stdio.h>
#include <error.h>

#define TAMANYO_ALM 1024

void main (int argc, char **argv)
{
    DIRP *dirp;
    struct dirent *dp;
    char almacen[TAMANYO_ALM];
    int fd;
    /* Comprueba que el número de parámetros sea el adecuado. En caso contrario, termina con un error */
    if (argc != 4) {
        fprintf (stderr, "Uso: mi_ls directorio > fichero \n");
        exit(1);
    }
    /* Crear el fichero recibido como argumento */
    fd = creat (argv[3], 0600);
    if (fd == -1)
    {
        fprintf (stderr, "Error al crear fichero de salida: %s \n", argv[3]);
        perror();
        exit (1);
    }
    /* Redirigir la salida estándar al fichero */
    close(stdout);     /* Cierra la salida estándar */
    dup(fd);           /* Redirige fd a stdout */
    close(fd);         /* Cierra fd */
    /* Ahora "fichero" está accesible por la salida estándar
    El resto del programa sigue igual que el programa 9.4 */
    /* Obtener e imprimir el directorio actual */
    getcwd (almacen, TAMANYO_ALM);
    printf ("Directorio actual: %s \n", almacen);
    /* Abrir el directorio recibido como argumento */
    dirp = opendir (argv[1]);
    if (dirp == NULL)
    {
        fprintf (stderr, "No se pudo abrir el directorio: %s \n", argv[1]);
        perror();
        exit (1);
    }
    else
    {
        printf ("Entradas en el directorio: %s \n", argv[1]);
        /* Leer el directorio entrada a entrada */
        while ((dp = readdir(dirp)) != NULL)
        printf ("%s\n", dp->d_name);          /* Imprimir nombre */
        closedir (dirp);
    }
    exit (0);
}
```

Observe que el programa es idéntico al 9.4, excepto en el bloque inicial, en el que se redirige la salida estándar al fichero solicitado. El mismo mecanismo se usa en UNIX cuando hay redirección de la salida de un mandato a un fichero.

5.16.7. Servicios Windows para directorios

A continuación se describen los servicios más comunes de Windows para gestión de directorios.

- BOOL **CreateDirectory** (LPCSTR lpPathName, LPVOID lpSecurityAttributes);

El servicio `CreateDirectory` permite crear un nuevo directorio en Windows. Esta llamada al sistema crea el directorio especificado en `lpPathName` con el modo de protección especificado en `lpSecurityAttributes`. El nombre del directorio puede ser absoluto para el dispositivo en que está el proceso o relativo al directorio de trabajo. En caso de éxito devuelve `TRUE`.

- BOOL **RemoveDirectory** (LPCSTR lpszPath);

Windows permite borrar un directorio especificando su nombre. El directorio `lpszPath` se borra únicamente cuando está vacío. Para borrar un directorio no vacío es obligatorio borrar antes sus entradas. Esta llamada devuelve `TRUE` en caso de éxito.

- HANDLE **FindFirstFile**(LPCSTR lpFileName, LPWindows_FIND_DATA lpFindFileData);
- BOOL **FindNextFile**(HANDLE hFindFile, LPWindows_FIND_DATA lpFindFileData);
- BOOL **FindClose**(HANDLE hFindFile);

En Windows hay tres servicios que permiten leer un directorio: `FindFirstFile`, `FindNextFile` y `FindClose`.

`FindFirstFile` es equivalente al `opendir` de UNIX. Permite obtener un manejador para buscar en un directorio. Además busca la primera ocurrencia del nombre de fichero especificado en `lpFileName`. `FindNextFile` es equivalente al `readdir` de UNIX, permite leer la siguiente entrada de fichero en un directorio que coincida con el especificado en `lpFileName`. `FindClose` es equivalente a `closedir` de UNIX y permite cerrar el manejador de búsqueda en el directorio.

- BOOL **SetCurrentDirectory** (LPCTSTR lpszCurDir);

Para poder viajar por la estructura de directorios, Windows define la llamada `SetCurrentDirectory`. El directorio destino se especifica en `lpszCurDir`. Si se realiza con éxito, esta llamada cambia el directorio de trabajo, es decir, el punto de partida para interpretar los nombres relativos. Si falla, no se cambia al directorio especificado. Si se especifica un nombre de dispositivo, como `A:\` o `D:\`, se cambia el directorio de trabajo en ese dispositivo.

- DWORD **GetCurrentDirectory** (DWORD cchCurDir, LPCTSTR lpszCurDir);

Windows permite conocer el directorio de trabajo actual mediante la llamada `GetCurrentDirectory`. El directorio de trabajo se devuelve en `lpszCurDir`. La longitud del `string` se indica en `cchCurDir`. Si el almacén es de longitud insuficiente, la llamada indica cómo debería ser de grande.

5.16.8. Ejemplo de uso de servicios Windows para directorios

Para ilustrar el uso de los servicios de ficheros que proporciona Windows, se resuelve en esta sección un ejemplo que muestra las entradas del directorio de trabajo. El código fuente, en lenguaje C, del ejemplo propuesto se muestra en el programa 5.15.

Programa 5.15. Programa que muestra las entradas del directorio de trabajo usando llamadas Windows.

```
#include <windows.h>
#include <direct.h>
#include <stdio.h>

#define LONGITUD_NOMBRE MAX_PATH + 2

int main (int argc, LPTSTR argv )
{
    BOOL Flags [MAX  OPTIONS];
    LPTSTR pSlash, pNombreFichero;
    int i, IndiceFichero;
    /* Almacen para el nombre del directorio */
    TCHAR Almacenpwd [LONGITUD_NOMBRE];
    DWORD CurDirLon;
    /* Manejador para el directorio */
    HANDLE SearchHandle;
```

```
    Windows_FIND_DATA FindData;

    /* Obtener el nombre del directorio de trabajo */
    CurDirLon = GetCurrentDirectory (LONGITUD_NOMBRE, Almacenpwd);
    if (CurDirLon == 0) {
       printf("Fallo al obtener el directorio. Error: %x\n",
          GetLastError());
       return -1;
    }
    if (CurDirLon > LONGITUD  NOMBRE) {
       printf("Nombre de directorio demasiado largo. Error: %x\n“,
          GetLastError());
       return -1;
    }
    /* Todo está bien. Leer e imprimir entradas */
    /* Encontrar el \ del final del nombre, si lo hay.
    Si no, añadirlo y restaurar el directorio de trabajo */
    pSlash = strrchr (Almacenpwd, '\\');
    if (pSlash != NULL) {
       *pSlash = '\0';
       SetCurrentDirectory (Almacenpwd);
       *pSlash = '\\';
       pNombreFichero = pSlash + 1;
    }
    else
       pNombreFichero = Almacenpwd;
    /* Abrir el manejador de busqueda del directorio y obtener
       el primer fichero. */
    SearchHandle = FindFirstFile (Almacenpwd, &FindData);
    if (SearchHandle == INVALID_HANDLE_VALUE)
    {
       printf ("Error %x abriendo el directorio.\n",
       GetLastError());
       return -1;
    }
    /* Buscar el directorio */
    do
    {
       printf (" %s \n", FindData.cFileName);
       /* Obtener el siguiente nombre de directorio. */
    } while (FindNextFile (SearchHandle, &FindData));
    return 0;
}
```

La estructura `FindData` contiene bastante más información que la mostrada en este programa. Se deja como ejercicio la extensión del Programa 5.13 para que muestre todos los datos de cada entrada de directorio.

5.17. SERVICIOS DE PROTECCIÓN Y SEGURIDAD

Los servicios de protección y seguridad de un sistema varían dependiendo de la complejidad del sistema implementado. En esta sección se muestran algunos servicios de protección en sistemas operativos clásicos, sin entrar en mecanismos de protección de sistemas distribuidos o en los problemas de distribución de claves por redes.

Dichos servicios se concretarán para UNIX y para Windows, incluyendo además ejemplos de uso de las llamadas a ambos sistemas operativos.

En general, todos los sistemas operativos crean la información de protección cuando se crea un objeto, por lo que no es muy frecuente encontrar servicios de creación y destrucción de ACL o capacidades disponibles para los usuarios. Es mucho más frecuente que los servicios de protección incluyan llamadas al sistema para cambiar características de la información de protección o para consultar dichas características.

Generalmente, existen llamadas al sistema para manipular los descriptores de protección: crear, destruir, abrir, obtener información, cambiar información y fijar información por defecto.

A continuación se detallan los servicios de protección existentes en UNIX y Windows.

5.17.1. Servicios UNIX de protección y seguridad

El estándar UNIX define servicios que, en general, se ajustan a los servicios genéricos descritos en la sección anterior. Sin embargo, no existen servicios específicos para crear, destruir o abrir descriptores de protección. Estos se asocian a los objetos y se crean y se destruyen con dichos objetos.

◙ int **access** (const char *path, int amode);

El servicio `access` comprueba si un fichero está accesible con unos ciertos privilegios.

Argumentos:

- `path`: nombre del fichero o directorio. No hace falta tener abierto el fichero.
- `amode`: modo de acceso que se quiere comprobar. amode es el OR inclusivo de R_OK, W_OK, X_OK o F_OK (comprobar existencia).

Devuelve: 0 si el proceso tiene acceso al fichero (para lectura, escritura o ejecución) o -1 en caso contrario.

Descripción: Utiliza el **UID real** y el **GID real** (en lugar de los efectivos) para comprobar los derechos de acceso sobre un fichero, pero no abre el fichero.

Ejemplos:

- `access("fichero",F_OK);` devuelve 0 si el fichero existe o -1 si no existe.
- `access("fichero",R_OK|W_OK);` devuelve 0 si el proceso que ejecuta la llamada tiene permisos de acceso de lectura y escritura sobre el fichero (utilizando el UID real y el GID real)

Este servicio se emplea, por ejemplo, en ejecutables que tengan activo el bit SETUID o SEGID, en los cuales la identidad real y efectiva no coincida. Por ejemplo, el administrador de un sistema quiere preparar un programa para que los usuarios puedan montar un disco USB, pero solamente en algún directorio al que tengan derechos de escritura. Como el servicio mount solamente lo puede ejecutar el superusuario, tiene que preparar un programa que ha de pertenecer al root y que ha de tener activo el bit SETUID, para que cualquier usuario lo pueda ejecutar. El problema es que dicho programa, al ejecutar como root, siempre podrá hacer el montado en cualquier directorio. Para evitar esto, el programa deberá incluir la secuencia siguiente:

```
if (0 == access("archivo", W_OK)) {
    /* El usuario puede escribir en el directorio, se realiza el montado, pero sin derechos de ejecución */
    imont = mount("archivo", "/mnt/sd", "vfat", MS_NOEXEC, NULL);
.....
} else {
    /* El punto de montado no pertenece al usuario, se niega la operación */
    .....
}
```

◙ mode_t **umask** (mode_t cmask);

La llamada `umask` permite definir una máscara de protección que será aplicada por defecto a todos los objetos creados por el proceso a partir de ese instante.

El **argumento** `cmask` es la nueva máscara.

Este servicio siempre tiene éxito y **devuelve** el valor de la máscara anterior a este cambio.

◙ int **chmod** (const char *path, mode_t mode);

El servicio `chmod` cambia los derechos de acceso a un archivo.

Argumentos:

- `name`: nombre del fichero al que se le cambian los permisos.
- `mode`: Nuevos bits de protección. El `mode` se expresa en octal, lo que se indica empezando por un 0.

Devuelve: Cero si tiene éxito o -1 si fracasa.

Descripción:

- Modifica los bits de permiso y los bits SETUID y SETGID del fichero, de acuerdo con `mode`. Estos últimos quedan definidos por:
  - S_ISUID = 04000
  - S_ISGID = 02000
- Sólo el propietario del fichero o el administrador pueden ejecutar este servicio.
- NO se utiliza la máscara, se ponen directamente los permisos indicados en `mode`.
- Si algún proceso tiene el archivo abierto, esta llamada no afectará a sus privilegios de acceso hasta que no lo cierre.

Ejemplo: Si "pepe" tiene los permisos 00750, y el dueño ejecuta `chmod ("pepe", 06711);` y tiene la máscara = 0077, el fichero quedará con los permisos 0711 y con los bits SETUID y SETGID activos.

◙ int **chown** (const char *path, uid_t owner, gid_t group);

`chown` cambia el propietario y el grupo de un archivo.

Argumentos:

- `name`: nombre del fichero al que se le cambia el propietario.
- `owner`: nuevo propietario del fichero.
- `group`: nuevo identificador de grupo del fichero.

Devuelve: Cero si tiene éxito o -1 si fracasa.
Descripción:

- Modifica el identificador de usuario y de grupo del fichero, por lo que un usuario puede 'regalar' un fichero a otro usuario.
- Los bits SETUID y SETGID son borrados. Esto es muy importante, puesto que evita que un usuario prepare un ejecutable maligno, le active el bit de SETUID y, seguidamente, se lo 'regale' al root. Se tendría, en principio, un fichero maligno del root con el bit de SETUID activo. Si ese usuario ejecutase dicho fichero, lo haría con usuario efectivo root. Este truco no funciona, puesto que al 'regalarle' el fichero al root se desactivan los bits SETUID y SETGID.
- Sólo el propietario del fichero o el administrador pueden cambiar estos atributos.
- Si algún proceso tiene el archivo abierto, esta llamada no afectará a sus privilegios de acceso hasta que no lo cierre.

Otros servicios UNIX.

El estándar UNIX define algunas llamadas más relacionadas con la identificación de usuarios y las sesiones de trabajo de los mismos. La llamada `getgroups` permite obtener la lista de grupos de un usuario. Las llamadas `getlogin` y `getlogin_r` devuelven el nombre del usuario asociado a un proceso. La llamada `setsid` crea un identificador de sesión para un usuario. La llamada `uname` permite identificar el sistema en el que se ejecuta el sistema operativo. Por último las llamadas `crypt` y `encrypt` permiten cifrar caracteres y bloques de bytes usando DES.

Para una referencia más completa de estas llamadas se remite al lector al archivo `/user/incluye/unistd.h` y a los manuales de descripción especializados.

5.17.2. Ejemplo de uso de los servicios de protección de UNIX

Para ilustrar el uso de los servicios de protección que proporciona UNIX, se presentan en esta sección dos pequeños programas, con su código fuente en lenguaje C. El primero (véase el programa 5.16) ejecuta la siguiente secuencia de acciones:

1. Comprueba que el archivo origen se puede leer y que la identificación efectiva y real del usuario son las mismas.
2. Crea el archivo destino con la máscara por defecto.
3. Copia el archivo antiguo al nuevo y restaura el modo de protección anterior.
4. Cambia el propietario y el grupo del archivo destino.

Programa 5.16 Copia con cambio del modo de protección de un archivo.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#define FIC_OR     "/tmp/fic_origen"
#define FIC_DES    "/tmp/fic_destino"
#define NEW_OWN  512     /* alumno miguel */
#define NEW_GRP  500     /* alumnos */

main (int argc, char **argv)
{
   int oldmask;

   if (getuid() != geteuid()) {
      printf ("Error: las identidades no coinciden \n");
      exit(0);
   }
   if (access(FIC_OR, R_OK) == -1) {
      printf ("Error: %s no se puede leer \n", FIC_OR);
      exit(0);
   }
   /* Comprobaciones bien. Crear el destino con máscara 007 */
   oldmask = umask (0007);
   if ( creat (FIC_DES, 0755) < 0) {
      printf ("Error: %s no pudo ser creado \n", FIC_DES);
```

```
        exit(0);
    }
    /* Restaurar la máscara de protección anterior */
    umask (oldmask);
    /* Si todo fue bien, el modo de protección efectivo es 750 */
    system (" cp /tmp/fic  origen /tmp/fic  destino");
    /* Cambiar su dueño y su grupo. Solo lo puede hacer el superusuario */
    chown (FIC_DES, NEW_OWN, NEW_GRP);
    exit (0);
}
```

El programa 5.17 permite consultar la identidad del dueño, de su grupo y los derechos de acceso de un archivo, cuyo nombre recibe como parámetro de entrada, usando la llamada al sistema `stat`. Este ejemplo no modifica nada, sólo extrae los parámetros y se los muestra al usuario. Para ello, el programa ejecuta la siguiente secuencia de acciones:

1. Ejecuta la llamada `stat` para el archivo solicitado y comprueba si ha recibido un error.
2. Si no le ha devuelto un error, extrae de la estructura de datos devuelta como parámetro de salida (struct stat) la información pedida. El identificador del dueño está en el campo `st_uid`, el de su grupo en `st_gid` y los permisos de acceso en `st_mode`.
3. Da formato a los datos obtenidos y los muestra por la salida estándar.

Programa 5.17 Extracción de la identidad del dueño, de su grupo y los derechos de acceso de un archivo.

```
#include <sys/types.h>
#include <stat.h>

main (int argc, char **argv)
{
        struct stat InfArchivo;             /* Estructura para datos de stat */
        int fd, EstadoOperacion, i;
        unsigned short Modo;                /* Modo de protección */
        char NombreArchivo[128];            /* Longitud máxima arbitraria: 128 */

        /* Se copia el nombre del archivo a la variable NombreArchivo */
        sprintf (NombreArchivo, "%s", argv[1]);
        /* Se escribe el nombre por salida estándar */
        printf ("\n %s ", NombreArchivo);

        EstadoOperacion = stat (NombreArchivo, &InfArchivo);
        if (EstadoOperacion != 0) {
            printf ("Error: %s nombre de archivo erróneo \n", NombreArchivo);
            exit(-1);
        } else {
            /* Se escribe el identificador de usuario por salida estándar */
            printf ("uid: %d ", InfArchivo.st_uid);
                /* Se escribe el identificador de grupo por salida estándar */
            printf ("gid: %d ", InfArchivo.st_gid);
            /* Se cogen grupos de 3 bits del Modo de protección. Los tres primeros son del dueño, los siguientes
             del grupo y los siguientes del mundo. Para estas operaciones se usan máscaras de bits y desplazamiento
             binario*/
            for (i=6; i>=0; i = i-3) {
                Modo = InfArchivo.st_mode >> i;
                if (Modo & 04) printf ("r"); else printf ("-");
                if (Modo & 02) printf ("w"); else printf ("-");
                if (Modo & 01) printf ("x"); else printf ("-");
            }
        }
        printf ("\n");
    }
    exit (0);
}
```

Nota: se puede obtener información detallada de la llamada al sistema `stat` y de la estructura de datos que maneja mediante el manual de usuario del sistema (`man stat`).

5.17.3. Servicios Windows de protección y seguridad

Windows tiene un nivel de seguridad C2 según la clasificación de seguridad del *Orange Book* del DoD, lo que supone la existencia de control de acceso discrecional, con la posibilidad de permitir o denegar derechos de acceso para cualquier objeto partiendo de la identidad del usuario que intenta acceder al objeto. Para implementar el modelo de seguridad, Windows usa un descriptor de seguridad y listas de control de acceso (ACL), que a su vez incluyen dos tipos de entradas de control de acceso (ACE): las de permisos y las de negaciones de accesos.

Las llamadas al sistema de Win32 permiten manipular la descripción de los usuarios, los descriptores de seguridad y las ACL. A continuación se describen las llamadas de Win32 más frecuentemente usadas.

- BOOL **InitializeSecurityDescriptor** (PSECURITY_DESCRIPTOR psd, DWORD dwRevision);

El servicio `InitializeSecurityDescriptor` inicia el descriptor de seguridad especificado en `psd` con unos valores de protección por defecto. `dwRevision` tiene el valor de la constante `SECURITY_DESCRIPTOR_REVISION`. Esta llamada devuelve `TRUE` si `psd` es un descriptor de seguridad correcto y `FALSE` en otro caso.

- BOOL **GetUserName** (LPTSTR NombreUsuario, LPDWORD LongitudNombre);

La llamada `GetUserName` permite obtener el identificador de un usuario que ha accedido al sistema. En caso de éxito, devuelve el identificador solicitado en `NombreUsuario` y la longitud del nombre en `LongitudNombre`. Esta función siempre debe resolverse con éxito, por lo que no hay previsto caso de error.

- BOOL **GetFileSecurity** (LPCTSTR NombreArchivo, SECURITY_INFORMATION secinfo, PSECURITY_DESCRIPTOR psd, DWORD cbSd, LPDWORD lpcbLongitud);

El servicio `GetFileSecurity` extrae el descriptor de seguridad de un archivo. No es necesario tener el archivo abierto. El nombre del archivo se especifica en `NombreArchivo`. `secinfo` es un tipo enumerado que indica si se desea la información del usuario, de su grupo, la ACL discrecional o la del sistema. `psd` es el descriptor de seguridad en el que se devuelve la información de seguridad. Esta llamada devuelve `TRUE` si todo es correcto y `FALSE` en otro caso.

- BOOL **SetFileSecurity** (LPCTSTR NombreArchivo, SECURITY_INFORMATION secinfo, PSECURITY_DESCRIPTOR psd);

El servicio `SetFileSecurity` fija el descriptor de seguridad de un archivo. No es necesario tener el archivo abierto. El nombre del archivo se especifica en `NombreArchivo`. `secinfo` es un tipo enumerado que indica si se desea la información del usuario, de su grupo, la ACL discrecional o la del sistema. `psd` es el descriptor de seguridad en el que se pasa la información de seguridad. Esta llamada devuelve `TRUE` si todo es correcto y `FALSE` en otro caso.

- BOOL **GetSecurityDescriptorOwner** (PSECURITY_DESCRIPTOR psd, PSID psidOwner);
- BOOL **GetSecurityDescriptorGroup** (PSECURITY_DESCRIPTOR psd, PSID psidGroup);

Las llamadas `GetSecurityDescriptorOwner` y `GetSecurityDescriptorGroup` permiten extraer el identificador del usuario de un descriptor de seguridad y del grupo al que pertenece. Habitualmente, el descriptor de seguridad pertenece a un archivo. Estas llamadas son sólo de consulta y no modifican nada. El descriptor de seguridad se especifica en `psd`. `psidOwner` es un parámetro de salida donde se obtiene el identificador del usuario y `psidGroup` es un parámetro de salida donde se obtiene el grupo del usuario. En caso de éxito devuelve `TRUE` y si hay algún error `FALSE`.

- BOOL **SetSecurityDescriptorOwner** (PSECURITY_DESCRIPTOR psd, PSID psidOwner, BOOL fOwnerDefaulted);
- BOOL **SetSecurityDescriptorGroup** (PSECURITY_DESCRIPTOR psd, PSID psidGroup, BOOL fGroupDefaulted);

Las llamadas `SetSecurityDescriptorOwner` y `SetSecurityDescriptorGroup` permiten modificar la identificación del usuario en un descriptor de seguridad y del grupo al que pertenece. Habitualmente, el descriptor de seguridad pertenece a un archivo. El descriptor de seguridad se especifica en `psd`. `psidOwner` es un parámetro de entrada donde se indica el identificador del usuario y `psidGroup` es un parámetro de entrada donde se indica el grupo del usuario. El último parámetro de cada llamada especifica que se debe usar la información de protección por defecto para rellenar ambos campos. En caso de éxito devuelve `TRUE` y si hay algún error `FALSE`.

Gestión de ACLs y ACEs.

- BOOL **InitializeAcl** (PACL pAcl, DWORD cbAcl, DWORD dwAclRevision);

Las llamadas `InitializeAcl`, `AddAccessAllowedAce` y `AddAccessDeniedAce` permiten iniciar una ACL y añadir entradas de concesión y denegación de accesos. `pAcl` es la dirección de una estructura de usuario de longitud `cbAcl`. El último parámetro debe ser `ACL_REVISION`.

- BOOL **SetSecurityDescriptorDacl** (PSECURITY_DESCRIPTOR psd, BOOL fDaclPresent, PACL pAcl, BOOL

fDaclDefaulted);

La ACL se debe asociar a un descriptor de seguridad, lo que puede hacerse usando la llamada `SetSecurityDescriptorDacl`. `psd` incluye el descriptor de seguridad. `fDaclPresent` a `TRUE` indica que hay una ACL válida en `pAcl`. `fdaclDefaulted` a `TRUE` indica que se debe iniciar la ACL con valores por defecto.

- BOOL **AddAccessAllowedAce** (PACL pAcl, DWORD dwAclRevision, DWORD dwAccessMask, PSID pSid);
- BOOL **AddAccessDeniedAce** (PACL pAcl, DWORD dwAclRevision, DWORD dwAccessMask, PSID pSid);

`AddAccessAllowedAce` y `AddAccessDeniedAce` permiten añadir entradas con concesión y denegación de accesos a una ACL. `pAcl` es la dirección de una estructura de tipo ACL, que debe estar iniciada. `dwAclRevision` debe ser `ACL_REVISION`. `pSid` apunta a un identificador válido de usuario. `dwAccessMask` determina los derechos que se conceden o deniegan al usuario o a su grupo. Los valores por defecto varían según el tipo de objeto.

Otras llamadas de Win32.

Win32 proporciona algunas llamadas más relacionadas con la identificación de usuarios. `LookupAccountName` permite obtener el identificador de un usuario mediante un nombre de cuenta suya. `LookupAccountSid` permite obtener el nombre de la cuenta de un usuario a partir de su identificador. Además incluye llamadas para obtener información de las ACLs (`GetAclInformation`), obtener las entradas de una ACE (`GetAce`) y borrarlas (`DeleteAce`).

Existe un conjunto de llamadas similar al mostrado para proporcionar seguridad en objetos privados de los usuarios, tales como *socket* o bases de datos propietarias. `CreatePrivateObjectSecurity`, `SetPrivateObjectSecurity` y `GetPrivateObjectSecurity` son algunas de estas funciones.

También es posible proteger objetos del núcleo, tales como la memoria. `CreateKernelObjectSecurity`, `SetKernelObjectSecurity` y `GetKernelObjectSecurity` son algunas de estas funciones.

5.17.4. Ejemplo de uso de los servicios de protección de Windows

Para ilustrar el uso de los servicios de protección que proporciona Win32, se presentan en esta sección dos pequeños programas en lenguaje C que consultan y manipulan los descriptores de seguridad de un objeto.

El programa 5.18 lee los atributos de seguridad de un archivo. Para ello extrae primero la longitud del descriptor de seguridad del archivo y luego el propio descriptor de seguridad, que devuelve como salida de la función. Como ejercicio se sugiere al lector que modifique este programa para extraer la ACL del archivo y cada una de sus entradas (ACEs).

Programa 5.18 Lectura de los atributos de seguridad de un archivo.

```
/* atrib.c – Permite obtener los atributos de seguridad de un archivo */
#include <windows.h>

PSECURITY_DESCRIPTOR LeerPermisosDeUnArchivo (LPCTSTR NombreArchivo)
/* Devuelve los permisos de un archivo */
{
   PSECURITY_DESCRIPTOR pSD = NULL;
   DWORD Longitud;
   HANDLE Procheap = GetProcessHeap ();
   /* Obtiene el tamaño del descriptor de seguridad. */
   GetFileSecurity (NombreArchivo, OWNER_SECURITY_INFORMATION |
         GROUP_SECURITY_INFORMATION | DACL_SECURITY_INFORMATION,
         pSD, 0, &Longitud);
   /* Obtiene el descriptor de seguridad del archivo*/
   pSD = HeapAlloc (Procheap, heap_GENERATE_EXCEPTIONS, Longitud);
   if (!GetFileSecurity (NombreArchivo, OWNER_SECURITY_INFORMATION |
         GROUP_SECURITY_INFORMATION | DACL_SECURITY_INFORMATION,
         pSD, Longitud, &Longitud))
      printf (“Error GetFileSecurity\n");
   return pSD;
}
```

El programa 5.19 crea un descriptor de seguridad y le asigna los valores de protección por defecto. Para ello extrae primero la longitud del descriptor de seguridad y del componente de la ACL, los crea y posteriormente asigna valores. Se sugiere al lector que siga los comentarios del programa como guía para comprender lo que hace.

Programa 5.19 Creación de un descriptor de seguridad con protección por defecto.

```
/* secobject.c – Creación de del descriptor de seguridad de un proceso y asignación de protección por defecto */
```

```
#include <windows.h>

int main (int argc, char** argv)
{
   /* Declaración de contenedores para longitudes */
   DWORD LongitudDACL;
   DWORD CodigoError;
   PSID SegProcSegId = NULL;
   /* Declaración de contenedores para descriptores de seguridad*/
   PACL pDACL;
   PSECURITY_DESCRIPTOR m_pSD = NULL;
   /* Definición de autoridad identificadora*/
   SID_IDENTIFIER_AUTHORITY siaWorld = SECURITY_WORLD_SID_AUTHORITY;
   int LongSidTemp;

   /* Asignación de memoria para descriptor de seguridad */
   m_pSD = malloc(sizeof(SECURITY_DESCRIPTOR));
   if (!m_pSD) {
      SetLastError(ERROR_NOT_ENOUGH_MEMORY);
      CodigoError = GetLastError();
      if (SegProcSegId) free (SegProcSegId);
      exit (-1);
   }/* Iniciación de valores en el descriptor de seguridad */
   if (!InitializeSecurityDescriptor(m_pSD,SECURITY_DESCRIPTOR_REVISION)) {
      CodigoError = GetLastError();
      if (SegProcSegId) free (SegProcSegId);
      exit (-1);
   }
/* Obtención de la longitud en bytes para almacenar un identificador de seguridad para una autoridad de
  identificación y del espacio de memoria correspondiente para el identificador*/
   LongSidTemp = GetSidLengthRequired(1);
   SegProcSegId = (PSID)malloc(LongSidTemp);
   if (!SegProcSegId) {
      SetLastError(ERROR_NOT_ENOUGH_MEMORY);
      CodigoError = GetLastError();
      if (SegProcSegId) free (SegProcSegId);
      exit (-1);
   }
   /* Obtención de espacio de memoria para la ACL */
LongitudDACL = sizeof (ACL) +sizeof (ACCESS_ALLOWED_ACE) - sizeof (DWORD)
   + LongSidTemp;
   pDACL = (PACL) malloc(LongitudDACL);
   if (!pDACL) {
      SetLastError(ERROR_NOT_ENOUGH_MEMORY);
      CodigoError = GetLastError();
      if (SegProcSegId) free (SegProcSegId);
      exit (-1);
   }
   /* Valores iniciales para la ACL y el SID. Si cualquiera falla se origina un error */
   if (!InitializeAcl(pDACL,LongitudDACL,ACL_REVISION)
       ||!InitializeSid(SegProcSegId,&siaWorld,1)) {
      CodigoError = GetLastError();
      if (SegProcSegId) free (SegProcSegId);
      exit (-1);
   }
   /* Extracción de la autoridad en el descriptor de seguridad */
   *(GetSidSubAuthority(SegProcSegId,0)) = SECURITY_WORLD_RID;
   /* Añadir una entrada de control de acceso a la ACL con valores por defecto. También se incluye el
     identificador de seguridad */
   if(!AddAccessAllowedAce(pDACL,ACL_REVISION,GENERIC_ALL|STANDARD_RIGHTS_ALL
      |SPECIFIC_RIGHTS_ALL,SegProcSegId)) {
      CodigoError = GetLastError();
      if (SegProcSegId) free (SegProcSegId);
      exit (-1);
   }
   /* Conectar la ACL creada con el descriptor de seguridad creado */
   if (!SetSecurityDescriptorDacl(m_pSD,TRUE,pDACL,FALSE)) {
```

```
        CodigoError = GetLastError();
        if (SegProcSegId) free (SegProcSegId);
        exit (-1);
    }
    printf ("\n Descriptor de seguridad con protección por defecto \n");
}
```

Como ejercicio se sugiere al lector que modifique este programa para asignar permisos específicos en el momento de la creación del descriptor de seguridad. Estos permisos se pueden pasar como parámetros del proceso.

5.18. LECTURAS RECOMENDADAS

Como se ha comentado a lo largo del capítulo, la mayoría de los libros generales de sistemas operativos (como [Silberschatz, 2006] y Stallings [Stallings, 2005]), realizan un tratamiento relativamente superficial del tema de E/S, centrándose, básicamente, en la gestión del almacenamiento, con la notable excepción de [Tanenbaum, 2001] y, sobre todo, [Tanenbaum, 2006], donde se incluye y analiza el código real de los manejadores del sistema operativo MINIX.

Para ampliar conocimientos sobre el tema de los manejadores de dispositivos, se recomienda el estudio de cómo se implementan en distintos sistemas operativos. Para Linux, se pueden consultar [Corbet, 2005] y [Bovet, 2005]. En el caso de UNIX, se recomiendan [McKusick, 1996], [McKusick, 2004], [Bach, 1999] y [Vahalia, 2006]. Por lo que se refiere a Windows, se pueden consultar [Baker, 2000], [Oney, 2002], Dekker [Dekker, 1999] y [Solomon, 2005].

En cuanto a los servicios de entrada/salida, en [Stevens, 1992] se presentan los servicios de UNIX y en [Hart, 2004] los de Windows.

Existe mucha bibliografía sobre sistemas de ficheros y directorios que puede servir como lectura complementaria a este libro. [Silberschatz 2005], [Stallings 2005] y [Tanenbaum 1997], son libros generales de sistemas operativos en los que se puede encontrar una explicación completa del tema con un enfoque docente.

[Grosshans 1986] contiene descripciones de las estructuras de datos que maneja el sistema de ficheros y presenta aspectos de acceso a ficheros. Con esta información, el lector puede tener una idea clara de lo que es un fichero y de cómo valorar un sistema de ficheros. [Folk 1987] incluye abundante información sobre estructuras de fichero, mantenimiento, búsqueda y gestión de ficheros. [Abernathy 1973] describe los principios de diseño básico de un sistema operativo, incluyendo el sistema de ficheros. En [McKusick 1996] se estudia el diseño del *Fast File System* con gran detalle, así como las técnicas de optimización usadas en el mismo. [Smith 1994] estudia la influencia de la estructura del sistema de ficheros sobre el rendimiento de las operaciones de entrada/salida. [Staelin 1988] estudia los tipos de patrones de acceso que usan las aplicaciones para acceder a los ficheros. Esta información es muy importante si se quiere optimizar el rendimiento del sistema de ficheros. Para otros mecanismos de incremento de prestaciones, consulte [Smith 1985], [Davy 1995] y [Ousterhout 1989].

Para programación de sistemas operativos puede consultar los libros de [Kernighan 1978] para aprender el lenguaje C y [Rockind 1985] para la programación de sistemas con UNIX y [Andrews 1996] para la programación de sistemas con Windows.

Para los ejemplos y casos de estudio se recomienda consultar [Beck 1996] para el caso de LINUX, [Goodheart 1994] para UNIX y [Solomon 1998] para Windows. Una explicación detalla del sistema de ficheros de Windows puede encontrarse en [Nagar 1997].

5.19. EJERCICIOS

1. Calcule las diferencias en tiempos de acceso entre los distintos niveles de la jerarquía de E/S. Razone la respuesta.

2. ¿Qué es el controlador de un dispositivo? ¿Cuáles son sus funciones?

3. ¿Es siempre mejor usar sistemas de E/S por interrupciones que programados? ¿Por qué?

4. ¿Qué ocurre cuando llega una interrupción de E/S? ¿Se ejecuta siempre el manejador del dispositivo? Razone la respuesta.

5. Indique dos ejemplos en los que sea mejor usar E/S no bloqueante que bloqueante.

6. ¿Qué problemas plantea a los usuarios la E/S bloqueante? ¿Y la no bloqueante?

7. ¿Se le ocurre alguna estructura para el sistema de E/S distinta de la propuesta en este capítulo? Razone la respuesta.

8. ¿Es mejor tener un sistema de E/S estructurado por capas o monolítico? Explique las ventajas y desventajas de ambas soluciones.

9. ¿En qué componentes del sistema de E/S se llevan a cabo las siguientes tareas?
a) Traducción de bloques lógicos a bloques del dispositivo.
b) Escritura de mandatos al controlador del dispositivo.
c) Traducir los mandatos de E/S a mandatos del dispositivo.
d) Mantener una cache de bloques de E/S.

10. ¿En qué consiste el DMA? ¿Para qué sirve?

11. En un centro de cálculo tienen un disco Winchester para dar soporte a la memoria virtual de un computador. ¿Tiene sentido? ¿Se podría hacer lo mismo con un disquete o una cinta? Razone la respuesta.

12. Suponga que un manejador de disco recibe peticiones de bloques de disco para las siguientes pistas: 2, 35, 46, 23, 90, 102, 3, 34. Si el disco tiene 150 pistas, el tiempo de búsqueda entre pistas consecutivas es de 4 ms. y el tiempo de búsqueda de la pista 0 a la 150 es de 8 ms., calcule los tiempos de búsqueda para los algoritmos de planificación de disco SSF, FCFS, SCAN y CSCAN.

13. Sea un disco con 63 sectores por pista, intercalado simple de sectores, tamaño de sector de 512 bytes y una velocidad de rotación de 3.000 RPM. ¿Cuánto tiempo costará leer una pista completa? Tenga en cuenta el tiempo de latencia.

14. El almacenamiento estable es un mecanismo *hardware*/*software* que permite realizar actualizaciones atómicas en almacenamiento secundario. ¿Cuál es el tamaño máximo de registro que este mecanismo permite actualizar atómicamente (sector, bloque, pista, ...)? Si la corriente eléctrica falla justo a mitad de una escritura en el segundo disco del almacenamiento estable, ¿cómo funciona el procedimiento de recuperación?

15. ¿Es lo mismo un disco RAM que una cache de disco? ¿Tienen el mismo efecto en la E/S? Razone la respuesta.

16. Suponga un controlador de disco SCSI, al que se pueden conectar hasta 8 dispositivos simultáneamente. El bus del controlador tiene 40 Mbytes/seg. de ancho de banda y puede solapar operaciones de búsqueda y transferencia, es decir, puede ordenar una búsqueda en un disco y transferir datos mientras se realiza la búsqueda. Si los discos tienen un ancho de banda medio de 2 Mbytes/seg. y el tiempo medio de búsqueda es de 6 ms., calcule el máximo número de dispositivos que el controlador podría explotar de forma eficiente si hay un 20% de operaciones de búsqueda y un 80% de transferencias, con un tamaño medio de 6 Kbytes, repartidas uniformemente por los 8 dispositivos.

17. ¿Cuáles son las principales diferencias, desde el punto de vista del sistema operativo, entre un sistema de copias de respaldo y un sistema de almacenamiento terciario complejo como HPSS?

18. Suponga que los dispositivos extraíbles, como las cintas, fueran tan caros como los discos. ¿Tendría sentido usar estos dispositivos en la jerarquía de almacenamiento?

19. Suponga un gran sistema de computación al que se ha añadido un sistema de almacenamiento terciario de alta capacidad. ¿De qué forma se puede saber si los ficheros están en el sistema secundario o en el terciario? ¿Se podría integrar todo el árbol de nombres? Razone la respuesta.

20. En algunos sistemas, como por ejemplo Linux, se almacena en una variable el número de interrupciones de reloj que se han producido desde el arranque del equipo, devolviéndolo en la llamada `times`. Si la frecuencia de reloj es de 100 Hz y se usa una variable de 32 bits, ¿cuánto tiempo tardará en desbordarse ese contador? ¿Qué consecuencias puede tener ese desbordamiento?

21. ¿Qué distintas cosas puede significar que una función obtenga un valor elevado en un perfil de ejecución de un programa?

22. Algunos sistemas permiten realizar perfiles de ejecución del propio sistema operativo. ¿De qué partes del código del sistema operativo no se podrán obtener perfiles?

23. Escriba el seudocódigo de una rutina de interrupción de reloj.

24. Suponga un sistema que no realiza la gestión de temporizadores directamente desde la rutina de interrupción, sino desde una rutina diferida. ¿En qué situaciones el contador de un temporizador puede tomar un valor negativo?

25. Analice para cada uno de estos programas si en su consumo de procesador predomina el tiempo gastado en modo usuario o en modo sistema.

i) Un compilador.
j) Un programa que copia un fichero.
k) Un intérprete de mandatos.
l) Un programa que comprime un fichero.
m) Un programa que resuelve una compleja fórmula matemática.

26. Enumere algunos ejemplos de situaciones problemáticas que podrían ocurrir si un usuario cambia la hora del sistema retrasándola. ¿Podría haber problemas si el cambio consiste en adelantarla?

27. Escriba el seudocódigo de las funciones de lectura, escritura y manejo de interrupciones para un terminal serie.

28. Lo mismo que el ejercicio anterior pero en el caso de un terminal proyectado en memoria.

29. Enumere ejemplos de tipos de programas que requieran que el terminal esté en modo carácter (modo crudo).

30. Escriba un programa usando servicios UNIX que lea un único carácter del terminal.

31. Realice el mismo programa utilizando servicios Windows.

32. Analice si es razonable permitir que múltiples procesos puedan leer simultáneamente del mismo terminal. ¿Y escribir?

33. Proponga métodos para intentar reducir lo máximo posible el número de veces que se copia un mensaje, tanto durante su procesamiento en la máquina emisora como en la máquina receptora.

34. En el apartado dedicado a la gestión de temporizadores, se esbozó el esquema usado en Linux. Recopile información sobre el mismo y escriba en seudo-código qué tratamiento se realiza con los temporizadores por cada interrupción de reloj.

35. Enumere qué funciones de gestión de dispositivos se han explicado a lo largo del capítulo que deberían ejecutarse dentro de rutinas diferidas. Proponga algún ejemplo adicional.

36. ¿Qué umbral sería razonable en el programa 8.7 si el dispositivo es muy lento? ¿Y si es muy rápido?

37. Repita el ejercicio para el programa 8.8.

38. Escriba una versión del *software* de gestión de la salida para un manejador de un dispositivo de

caracteres usando el mismo esquema que en el programa 8.3. ¿Sería también erróneo?

39. Escriba una versión del *software* de gestión de la salida para un manejador de un dispositivo de caracteres usando el mismo esquema que en el programa 8.4.

40. Escriba una versión del *software* de gestión de la salida para un manejador de un dispositivo de caracteres usando el mismo esquema que en el programa 8.5.

41. Analice las consecuencias de usar un umbral igual a 1 en el programa 8.8.

42. Analice las consecuencias de usar un umbral igual al tamaño del *buffer* en el programa 8.8.

43. Supóngase un hipotético dispositivo X de uso compartido, de solo lectura y dirigido por interrupciones. El manejador incluye 2 funciones: `LeerX`, que recibe la dirección de usuario especificada por el proceso donde éste desea leer el dato, y la rutina de interrupción. Se plantea el esqueleto de 4 versiones simplificadas, 2 de ellas erróneas, del manejador de este dispositivo (en el código, *Bloquear* incluye al proceso actual al final de una lista de bloqueo, *Desbloquear* pone como listo al primer proceso de la lista, *ResMem* reserva memoria del núcleo y *LibMem* la libera, *InserPet* incluye una petición al final de una cola de peticiones incluyendo la dirección donde copiar el dato y *ExtrPet* extrae la primera petición devolviendo su dirección).

Versión 1

```
tipo_lista_proc espera;
tipo_lista_proc list_opera;
dato *destino;
LeerX (dato *dir_usu) {
   Si (list_opera!= NULL)
      Bloquear(espera);
   Programar_disp_X();
   destino=dir_usu;
   Bloquear(list_opera);
   Si (espera != NULL)
      Desbloquear(espera);
}
InterrupciónX {
   *destino=reg_datos;
   Desbloquear(list_opera);
}
```

Versión 2

```
tipo_lista_proc espera;
tipo_lista_proc list_opera;
dato buf_int;
LeerX (dato *dir_usu) {
   Si (list_opera!= NULL)
      Bloquear(espera);
   Programar_disp_X();
   Bloquear(list_opera);
   *dir_usu=buf_int;
   Si (espera != NULL)
      Desbloquear(espera);
}
InterrupciónX {
    buf_int=reg_datos;
   Desbloquear(list_opera);
}
```

Versión 3

```
tipo_lista_proc lista;
tipo_cola_peticiones cola;
LeerX (void *dir_usu) {
   dato *p_buf_int;
   p_buf_int=ResMem();
   InserPet(cola,p_buf_int);
   Si (lista != NULL)
      Bloquear(lista);
   Sino {
      Programar_disp_X();
      Bloquear(lista);
   }
   *dir_usu=*p_buf_int;
   LibMem(p_buf_int);
}
InterrupciónX {
   dato *p_buf_int;
   p_buf_int=ExtrPet(cola);
   *p_buf_int=reg_datos;
   Desbloquear(lista);
   Si (lista != NULL)
      Programar_disp_X();
}
```

Versión 4

```
tipo_lista_proc lista;
tipo_cola_peticiones cola;
LeerX (void *dir_usu) {
   dato *p_buf_int;
   p_buf_int=ResMem();
   InserPet(cola,p_buf_int);
   Si (lista != NULL)
      Bloquear(lista);
   Sino {
      Programar_disp_X();
      Bloquear(lista);
   }
   *dir_usu=*p_buf_int;
   LibMem(p_buf_int);
}
InterrupciónX {
   dato *p_buf_int;
   p_buf_int=ExtrPet(cola);
   *p_buf_int=reg_datos;
   Desbloquear(lista);
   Si (lista != NULL) {
      Programar_disp_X();
      Bloquear(lista);
   }
}
```

n) Comparando las dos primeras versiones, ¿cuál es incorrecta y por qué? (observe que las diferencias entre ambas están marcadas en cursiva).

o) Comparando las dos últimas versiones, ¿cuál es incorrecta y por qué? (nuevamente, las diferencias están marcadas en cursiva).

p) Compare la eficiencia de las dos versiones correctas, analizando cuál genera menos cambios de contexto. Para ello, puede plantearse una traza de ejecución de varios procesos que leen de X simultáneamente.

q) ¿Cómo se podría usar la técnica de diferir interrupciones mediante la interrupción *software* en la segunda solución correcta? Muestre en seudo-código cómo se utilizaría esta técnica en ese caso.

44. ¿Se puede emular el método de acceso aleatorio con el secuencial? ¿Y viceversa? Explique las ventajas e inconvenientes de cada método.

45. ¿Cuál es la diferencia entre la semántica de compartición UNIX y la de versiones? ¿Podría emularse la semántica UNIX con la de versiones?

46. ¿Podrían establecerse enlaces al estilo de los UNIX en Windows ? ¿Por qué?

47. ¿Es UNIX sensible a las extensiones del nombre de fichero? ¿Lo es Windows ?

48. ¿Cuál es la diferencia entre nombre absoluto y relativo? Indique dos nombres relativos para `/users/miguel/datos`. Indique el directorio respecto al que son relativos.

49. ¿Cuál es la ventaja de usar un grafo acíclico frente a usar un árbol como estructura de los directorios? ¿Cuál puede ser su principal problema?

50. En UNIX existe una aplicación `mv` que permite renombrar un fichero. ¿Hay alguna diferencia entre implementarla usando `link` y `unlink` y hacerlo copiando el fichero origen al destino y luego borrando este último?

51. Modifique el programa 9.1 para que en lugar de copiar un fichero sobre otro, lea un fichero y lo saque por la salida estándar. Su programa es equivalente al mandato `cat` de UNIX.

52. Modifique el programa 9.2 para que en lugar de copiar un fichero sobre otro, lea un fichero y lo saque por la salida estándar. Su programa es equivalente al mandato `type` de Windows o MS-DOS.

53. Usando llamadas UNIX, programe un mandato que permita leer un fichero en porciones, especificando el formato de la porción. Para incrementar el rendimiento de su mandato, debe hacer lectura adelantada de la siguiente porción del fichero.

54. Modifique el programa 9.4 (`mi_ls`) para que, además de mostrar el nombre de las entradas del directorio, muestre algo equivalente a la salida del mandato `ls -l` de UNIX.

55. Haga lo mismo que en el ejercicio 11 para el programa 9.3.1 del entorno Windows .

56. ¿Qué método es mejor para mantener los mapas de bloques: mapas de bits o listas de bloques libres? Indique por qué. ¿Cuál necesita más espacio de almacenamiento? Explíquelo.

57. ¿Qué problema tiene usar bloques grandes o agrupaciones? ¿Cómo puede solucionarse?

58. ¿Qué es mejor en un sistema donde hay muchas escrituras: un sistema de ficheros convencional o uno de tipo LFS? ¿Y para lecturas aleatorias?

59. ¿Es conveniente mantener datos de fichero dentro del descriptor de fichero, como hace Windows ? ¿Es mejor tener un descriptor como el nodo-i de UNIX?

60. Tener una buena tasa de aciertos en la cache (% de bloques que se encuentran en la cache) es fundamental para optimizar la entrada/salida. Suponga que el tiempo medio de acceso de un disco es de 17 milisegundos y satisfacer una petición desde la cache cuesta 0.5 milisegundos. Elabore una fórmula para calcular el tiempo necesario para satisfacer una petición de n bloques. Considere la tasa de aciertos de la cache como un parámetro más de dicha fórmula. Calcule el tiempo de servicio para 5 bloques y una tasa de aciertos del 85%.

61. ¿Qué problemas de seguridad puede plantear la cache de nombres? ¿Cómo se pueden solucionar?

62. Imagine que está comprobando el estado del sistema de ficheros y observa que el nodo-i 342 tiene 4 referencias, pero aparece libre en el mapa de bits de nodos-i. ¿Cómo resolvería el problema?

63. Imagine el problema inverso al del ejercicio 19. El nodo-i no tiene referencias, pero aparece como ocupado. ¿Cómo lo solucionaría?

64. ¿Cómo se puede mejorar el tiempo de asignación de bloques de un dispositivo muy fragmentado que sólo admite acceso aleatorio? ¿Valdría su solución para dispositivos de acceso secuencial?

65. Determinar cuántos accesos físicos a disco serían necesarios, como mínimo, en un sistema UNIX para ejecutar la operación:
`fd = open("/lib/agenda/direcciones", O_RDONLY);`
Explique su respuesta. Suponga que la cache del sistema de ficheros está inicialmente vacía.

66. Realizar el ejercicio 22 para el fichero "`C:\lib\agenda\direcciones`". ¿Por qué el número de accesos puede ser tan distinto en ambos sistemas?

67. Se quiere diseñar un sistema de ficheros para un sistema operativo que dará servicio a un entorno del que se sabe lo siguiente:
– El tamaño medio de los ficheros es de 1,5 Kbytes.
– El número medio de bloques libres es el 7% del total.
– Se usan 16 bits para la dirección del bloque.
Para este sistema se selecciona un disco duro con bloques físicos de 1 Kbyte y con una capacidad igual a la del máximo bloque direccionable.

Teniendo en cuenta que se debe optimizar el uso del disco y la velocidad de acceso (por este orden) y que la memoria física de que se dispone es suficiente, conteste razonadamente a las siguientes preguntas:
a) ¿Cuál será el tamaño de bloque lógico más adecuado?
b) ¿Cuál será el método más adecuado para llevar el control de los bloques lógicos del disco?

68. Un proceso de usuario ejecuta operaciones de entrada/salida en las que pide los siguientes bloques de un sistema de ficheros:
`1 2 3 4 1 3 10 2 3 2 10 1`
Suponiendo que en la cache de bloques del sistema operativo caben `4` bloques y que inicialmente está vacía. Se pide:
a) Hacer una traza de la situación de los bloques de la cache para cada petición de bloque del proceso, suponiendo que se usa un política de reemplazo LRU (*Least Recently Used*).
b) Hacer una traza de la situación de los bloques de la cache para cada petición de bloque del proceso, suponiendo que se usa un política de reemplazo MRU (*Most Recently Used*).

69. En un sistema de ficheros tradicional de UNIX, cada partición tiene un superbloque donde se guarda información acerca de la estructura del sistema de ficheros. Sin este superbloque es imposible acceder al sistema de ficheros. ¿Qué problemas presenta una estructura de ficheros como la anterior? ¿Cómo podrían resolverse? Indique un sistema de ficheros en el que se han propuesto soluciones al problema anterior.

70. ¿Qué operaciones se hacen en UNIX para montar un sistema de ficheros sobre un directorio? ¿Y en Windows?

71. En Windows existe una utilidad `ln` dentro de las utilidades del sistema. Cree un fichero `fichero.orig` y use `ln` para crear un enlace físico entre dos ficheros de Windows, ejecutando el mandato:

```
ln   fichero.orig   fichero.dest
```

Use el explorador para ver la entrada de ambos ficheros. Borre el `fichero.orig` y describa lo que ocurre.

1. Los ficheros de metadatos de un sistema de ficheros de Windows son ficheros regulares, denominados `MFT`,pero están ocultos a los usuarios. Ejecute el mandato:

```
C:\>dir/A:h
```

e indique cuál es el tamaño del fichero de metadatos del dispositivo lógico `C:`.

2. ¿Cuánto espacio de disco se necesita para tener redundancia en un disco espejo? ¿Cuánto hace falta en un sistema RAID IV con 4 discos, incluyendo los de paridad?

3. ¿Cuál es la sobrecarga para las escrituras en un disco espejo? ¿Cuál es la sobrecarga para las escrituras en un sistema RAID con 3 discos de datos y 1 de paridad, asumiendo que se escribe de un golpe sobre todos los dispositivos?

6

COMUNICACIÓN Y SINCRONIZACIÓN DE PROCESOS

En este capítulo se presentan los problemas que surgen cuando los diferentes procesos que se ejecutan en un sistema compiten por recursos o se comunican entre sí. Esta situación, que se denomina concurrencia, ocurre cuando se produce la ejecución entrelazada en un mismo sistema de las instrucciones de diferentes procesos o threads. *En el capítulo se describen los problemas que plantea la ejecución concurrente de procesos y modelos clásicos de comunicación y sincronización entre procesos que ocurren en la vida real. A continuación, se introducen y analizan los principales mecanismos que ofrecen los sistemas operativos para la comunicación y sincronización de procesos, se muestra cómo utilizar éstos para resolver los problemas anteriores y se presenta el concepto de transacción atómica. También se explican los principales aspectos de diseño e implementación de los mecanismos de comunicación y sincronización, y cómo se realiza la sincronización y comunicación dentro del propio sistema operativo. El capítulo finaliza con la descripción de los principales servicios UNIX y Windows para la comunicación y sincronización entre procesos. Los temas que se cubren en este capítulo son:*

- *Concurrencia y tipos de procesos concurrentes.*
- *Modelos clásicos de comunicación y sincronización.*
- *Mecanismos que ofrecen los sistemas operativos para la comunicación y sincronización entre procesos.*
- *Transacciones.*
- *Aspectos de diseño e implementación de los mecanismos de comunicación y sincronización.*
- *La sincronización y comunicación dentro del propio sistema operativo.*
- *Servicios UNIX y Windows para la comunicación y sincronización entre procesos.*

6.1. CONCURRENCIA

La concurrencia, en términos generales, es la acción de juntarse en el mismo lugar o en el mismo tiempo. La concurrencia en el tiempo en los sistemas informáticos ocurre a muy diversos niveles, como se muestra en la figura 6.1. Dicha figura está ordenada de acuerdo al grano de concurrencia, empezando por la concurrencia a nivel de grano fino, como ocurre internamente en los procesadores, y finalizando con la concurrencia en los multicomputadores y aplicaciones paralelas.

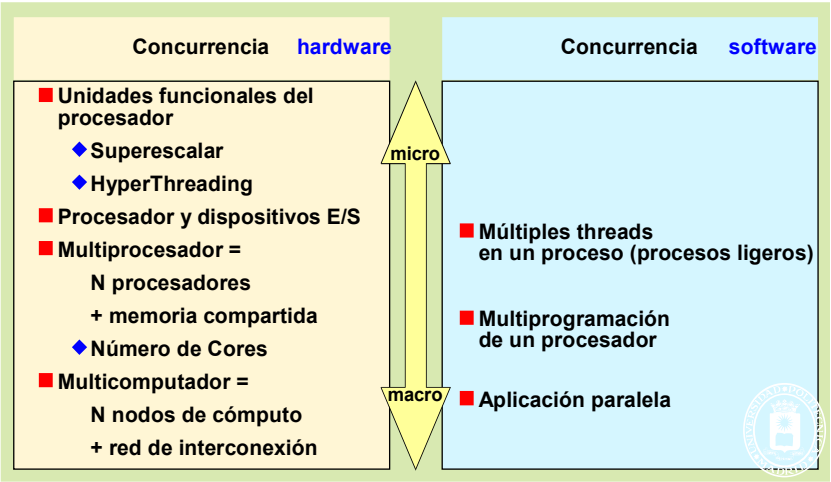


Figura 6.1 *La concurrencia ocurre a muy diversos niveles.*

Advertencia 6.1. La mayoría de las secciones de este capítulo se aplican tanto a procesos como a *threads*. Para evitar repetir constantemente "procesos o *threads*" utilizaremos el término de proceso para referirnos a ambos. Solamente cuando queramos diferenciar entre ambos utilizaremos los términos de proceso pesado y de *thread*.

Concurrencia *hardware*

La concurrencia *hardware* ocurre cuando se dispone de varias unidades *hardware* actuando simultáneamente.

Los procesadores actuales son altamente complejos, las técnicas de segmentación (*pipe-line*) y superescalares permiten realizar varias operaciones de forma simultánea, mientras que la técnica de multihilo (hyper o multi *threading*) permiten ejecutar de forma alternada dos o más flujos de instrucciones.

Por otro lado, los controladores de la mayoría de los periféricos trabajan de forma autónoma con acceso directo a memoria (DMA), permitiendo que el procesador ejecute simultáneamente otro programa, según se ha visto en el capítulo "5 E/S y Sistema de ficheros".

La mayoría de los computadores actuales son de tipo multiprocesador e incluso multihilo, por lo que, además del flujo de actividad de entrada salida, se cuenta con un flujo de ejecución por cada procesador (o por cada hilo de cada procesador multihilo). Un multiprocesador contiene una única memoria principal compartida por todos los procesadores, por lo que se dice que es un **sistema fuertemente acoplado** al permitir que procesos y *threads* compartan memoria.

Los grandes sistemas presentan una arquitectura multicomputador. En este caso, los nodos del sistema pueden ser de tipo multiprocesador, pero el acoplamiento entre nodos se hace mediante paso de menajes sobre una red de interconexión, por lo que se dice que los multicomputadores son **débilmente acoplados**.

Concurrencia *software*

En los **monoprocesadores** todos los procesos concurrentes se ejecutan sobre un único procesador. El sistema operativo se encarga de ir repartiendo el tiempo del procesador entre los distintos procesos, **intercalando** la ejecución de los mismos para dar así una *apariencia* de ejecución simultánea. En la figura 6.2 se presenta un ejemplo de ejecución de cuatro procesos en un sistema multiprogramado con un único procesador. Como puede verse, los procesos van avanzando su ejecución de forma aparentemente simultánea, pero sin coincidir en ningún momento sus fases de procesamiento.

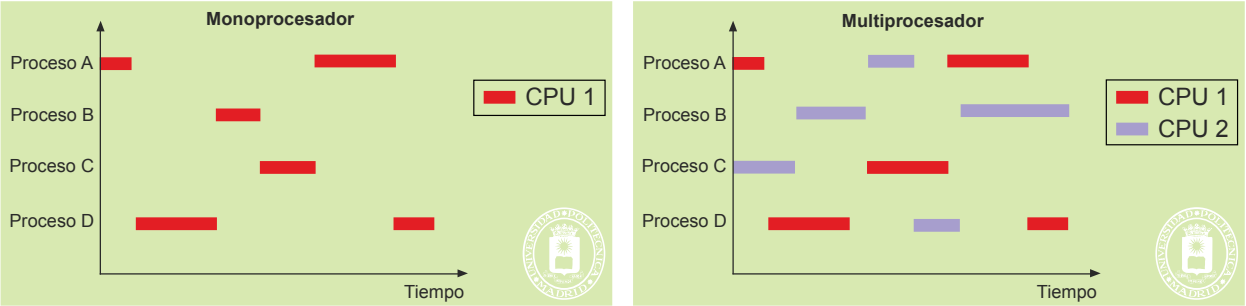


Figura 6.2 *Ejemplo de ejecución en un sistema multiprogramado con uno y dos procesadores.*

En los **multiprocesadores y multicomputadores** sí existe una verdadera ejecución simultánea de procesos al coincidir en el tiempo las fases de procesamiento de distintos procesos. En un instante dado se pueden ejecutar de forma simultánea tantos procesos como procesadores haya. En la figura 6.2 se muestra un ejemplo de ejecución de cuatro procesos en un multiprocesador con dos procesadores.

La **concurrencia *software*** ocurre cuando se produce la ejecución entrelazada en un mismo sistema de las instrucciones de diferentes procesos. La concurrencia será **aparente** siempre que haya más de un proceso por procesador, puesto que estos intercalan su ejecución, sin ejecutar al mismo tiempo. Sin embargo, la concurrencia será **real** cuando haya un proceso por procesador, puesto que existe superposición en el tiempo de ejecución. En el primer caso podemos hablar también de *pseudoparalelismo* y en el segundo de *paralelismo real*.

Aunque puede parecer que la intercalación y la superposición de la ejecución de procesos, es decir de la concurrencia aparente y real, presentan formas de ejecución distintas, ambas presentan los mismos problemas, que pueden resolverse utilizando los mismos mecanismos.

6.1.1. Ventajas de la concurrencia

Las principales razones que justifican la ejecución concurrente son las siguientes:

- Aunque la programación concurrente tiene su complejidad, f**acilita la programación** de aplicaciones complejas, al permitir que éstas se estructuren como un conjunto de procesos que cooperan entre sí para alcanzar un objetivo común. Por ejemplo, un compilador puede construirse mediante dos procesos: el compilador propiamente dicho, que se encarga de generar código ensamblador, y el proceso ensamblador, que obtiene código en lenguaje máquina a partir del ensamblador. Ambos procesos han de comunicarse entre sí.
- **Acelera los cálculos**. Si se quiere que una tarea se ejecute con mayor rapidez, lo que se puede hacer es dividirla en procesos, de forma que cada uno de ellos resuelva una parte más pequeña del problema. Se conseguirá resolver el problema en un tiempo menor si estos procesos se ejecutan con paralelismo real. Hay que hacer notar, sin embargo, que esta división a veces es difícil y no siempre es posible.
- **Posibilita el uso interactivo** a múltiples usuarios que trabajan de forma simultánea desde varios terminales.
- **Permite un mejor aprovechamiento de los recursos,** en especial del procesador, ya que, como se vio en el capítulo "2 Introducción a los sistemas operativos", se pueden aprovechar las fases de entrada/salida de unos procesos para realizar las fases de procesamiento de otros.

La concurrencia exige, como se verá en próximas secciones, coordinar el acceso a los recursos compartidos para evitar errores.

6.1.2. Tipos de procesos concurrentes

Los procesos que se ejecutan de forma concurrente se pueden clasificar como procesos independientes o cooperantes.

Un proceso **independiente** es aquél que ejecuta sin requerir la ayuda o cooperación de otros procesos. Un claro ejemplo de procesos independientes son los diferentes intérpretes de mandatos (*shells*) que se ejecutan de forma simultánea en un sistema.

Los procesos son **cooperantes** cuando están diseñados para trabajar conjuntamente en alguna actividad, para lo que deben ser capaces de comunicarse e interactuar entre sí. En el ejemplo anterior del compilador los dos procesos que lo conforman son procesos cooperantes, uno encargado de generar código ensamblador y otro encargado de obtener código en lenguaje máquina a partir del código ensamblador.

Tanto si los procesos son independientes como si son cooperantes, pueden producirse una serie de interacciones entre ellos. Estas interacciones pueden ser de dos tipos:

- Interacciones motivadas porque los procesos **comparten** o **compiten** por el acceso a recursos físicos o lógicos. Esta situación aparece en los distintos tipos de procesos anteriormente comentados. Por ejemplo, dos procesos totalmente independientes pueden competir por el acceso a disco. En este caso, el sistema

operativo deberá encargarse de que los dos procesos accedan ordenadamente al disco sin que se cree ningún conflicto. Esta situación también aparece cuando varios procesos desean modificar el contenido de un registro de una base de datos. Aquí, es el gestor de la base de datos el que se tendrá que encargar de ordenar los distintos accesos al registro.

- Interacción motivada porque los procesos se **comunican** y **sincronizan** entre sí para alcanzar un objetivo común. Los procesos compilador y ensamblador descritos anteriormente son dos procesos que deben comunicarse y sincronizarse con el fin de producir código en lenguaje máquina.

Estos dos tipos de interacciones obligan al sistema operativo a incluir unos servicios que permitan la comunicación y la sincronización entre procesos, servicios que se presentarán a lo largo de este capítulo.

1.1.2. Tipos de recursos compartidos

La concurrencia implica el uso de recursos compartidos. Estos recursos se pueden clasificar de acuerdo a los siguientes criterios:

- **Físicos** o **lógicos**. Ejemplos de recursos físicos son la memoria, el procesador, el disco o la red. Ejemplos de recursos lógicos son un fichero, un semáforo o una base de datos.
- **Reutilizable** o **consumible**. Un recurso es reutilizable si sigue existiendo después de que un proceso lo use, pero será consumible cuando desaparece una vez utilizado. Ejemplos de recursos consumibles son los mensajes, las señales o los semáforos.
- **Exclusivo** o **compartido**. Los procesos pueden compartir el uso de un recurso o lo deben usar en modo exclusivo o dedicado. Un recurso compartido está siempre disponible, mientras que el exclusivo sólo puede ser utilizado por un proceso en cada instante.
- **Simple** o **múltiple**. Es simple cuando hay un único ejemplar del recurso y múltiple cuando existen varias unidades del mismo.
- El recuso es **expropiable** cuando se le puede arrebatar el recurso al proceso que lo está utilizando.

6.1.3. Recursos compartidos y coordinación

El hecho de que la ejecución concurrente implique el empleo de recursos compartidos exige la necesidad de coordinar de forma adecuada el acceso a estos recursos. Esta coordinación la puede gestionar directamente el sistema operativo, de forma transparente a los usuarios, o puede ser gestionada de forma explícita por los propios procesos. La coordinación puede ser, por tanto:

Coordinación implícita.

Es la coordinación que realiza el propio **sistema operativo** como gestor de los recursos del sistema. Este tipo de coordinación es necesario para que los procesos independientes puedan utilizar los diferentes recursos del sistema de forma totalmente transparente, ignorando que los comparten y compiten por ellos. Así, por ejemplo, un proceso hará uso de la memoria asignada por el sistema operativo sin preocuparse de la memoria asignada a otros procesos.

El tipo de control de acceso que hace el sistema operativo depende del tipo de recurso. Por ejemplo, si varios procesos acceden simultáneamente a un fichero, el sistema operativo actuará de acuerdo a la semántica de coutilización del sistema de ficheros correspondiente, lo que garantiza que el fichero no se corromperá, pero no implica que su contenido final sea el deseado, si, por ejemplo, los procesos han escrito en él de forma desordenada.

Coordinación explícita.

Este tipo de coordinación es la que necesitan los procesos cooperantes. En este caso, diferentes procesos cooperan y coordinan sus acciones para obtener un fin común. Por tanto, los procesos cooperantes son conscientes del uso de los recursos compartidos.

Para que los diferentes procesos puedan coordinar y sincronizar el acceso a estos recursos es necesario que utilicen mecanismos de sincronización y comunicación proporcionados por el sistema operativo o por el lenguaje de programación. Existen lenguajes de programación como Ada o Java que incluyen mecanismos para el acceso coordinado a los recursos compartidos. Otros lenguajes de programación, como es el caso de C, no ofrecen dichos mecanismos, por lo que se debe recurrir a los servicios que ofrece el sistema operativo.

Gran parte de este capítulo se dedica a cómo realizar la coordinación explícita de los procesos concurrentes.

6.1.4. Resumen de los conceptos principales

Como resumen de esta sección de introducción a la concurrencia destacamos los tres conceptos siguientes:

- La concurrencia **implica** recursos compartidos. Estos recursos pueden ser:
  - Físicos: procesador, memoria, disco, red, ... → ¡TODO!
  - Lógicos: ficheros, interfaz de usuario, mecanismos de comunicación, etc.
- La concurrencia **permite**...

- Aprovechamiento del *hardware*: solapar el uso de los recursos físicos.
- Modularidad en las aplicaciones: descomposición cómoda del problema.

- La concurrencia **exige** coordinar el acceso a los recursos compartidos …..

  ….. Si no, los resultados pueden ser erróneos al ¡producirse condiciones de carrera!, como veremos más adelante.
  Para ello, se utilizarán unos mecanismos diseñados para este fin.

6.2. CONCEPTO DE ATOMICIDAD

En programación concurrente se dice que una operación o un conjunto de operaciones es atómico, linealizable, indivisible o ininterrumpible cuando, para el resto del sistema concurrente, se ejecuta aparentemente de forma instantánea. Lo que significa que queda aislado del resto del sistema concurrente y que sus pasos de ejecución no se van a entremezclar con los pasos del resto del sistema concurrente. En muchos casos, como en la transacciones que se analizan más adelante, la atomicidad, además, tiene la propiedad éxito-o-fallo, lo que significa que o se ejecuta de forma correcta toda la operación atómica, cambiando de forma exitosa el estado del sistema, o no tiene ningún efecto sobre el mismo.

Las **instrucciones de máquina** que ejecuta el computador **son atómicas**, puesto que las interrupciones solamente se admiten al final de cada instrucción[2]. Sin embargo, incluso la sentencia de alto nivel más simple NO lo es, como muestra, más abajo, la figura 6.4.

Para garantizar la atomicidad de un segmento de código es necesario utilizar unos mecanismos *software* de sincronización, que analizaremos a lo largo del capítulo. Para poder construir dichos mecanismos el *hardware* proporciona unas instrucciones de máquina que realizan de forma atómica dos operaciones. Ejemplos de esas instrucciones de máquina son:

- `Test_and_Set_Lock  Ra, M(d)`. Esta instrucción realiza de forma atómica las dos operaciones siguientes:
  - Ra ← M(d) (carga en el registro Ra el contenido de la posición de memoria de dirección d).
  - M(d) ← 1 (la posición de memoria de dirección d recibe un 1).
- `Swap Ra, M(d)` de forma atómica intercambia el valor del registro Ra con la posición memoria de dirección d.

Estas instrucciones máquina son el **soporte** para conseguir **atomicidad** en mecanismos *software* de **orden superior** que permitirán eliminar las condiciones de carrera (resolviendo la sección crítica), como veremos más adelante.

6.3. PROBLEMAS QUE PLANTEA LA CONCURRENCIA

Los dos problemas más habituales asociados con la concurrencia son las condiciones de carrera y los interbloqueos.

6.3.1. Condiciones de carrera

Este problema surge cuando varios procesos acceden de forma concurrente y sin coordinación a recursos compartidos. El término de carrera surge porque el resultado depende de las velocidades relativas de ejecución de los procesos involucrados. Para ilustrar este problema se van a presentar unos ejemplos en los que existe un fragmento de código que puede dar lugar a condiciones de carrera.

La figura 6.3 muestra un ejemplo de ejecución concurrente. Se trata de una base de datos bancaria en la que se considera la variable saldo de la cuenta del cliente Pepe, con un valor inicial de 120 €. Suponemos que Pepe está en un cajero haciendo una operación de retirada de fondos por importe de 20 €. De forma simultánea, le están haciendo una transferencia de 1.000 €.

2 Algunas instrucciones de máquina, como la copia múltiple de memoria a memoria, no son atómicas.

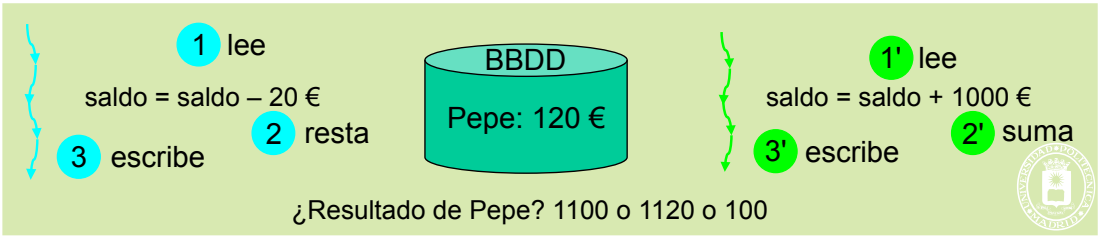


Figura 6.3 Acceso concurrente a la base de datos de una cuenta bancaria.

Cada una de las dos operaciones implica los tres pasos de: 1 leer el valor de la variable saldo, 2 restarle 20 o incrementarle 1.000 y 3 salvar el resultado. Es de observar que, dependiendo del orden en que ejecuten estas operaciones los procesos que atienden al cajero y a la transferencia, el resultado obtenido será diferente. Algunas posibles situaciones son las siguientes:

	Orden 1	**Orden 2**	**Orden 3**	**Orden 4**	**Orden 5**
	1	1'	1	1	1'
	1'	2'	2	2	2'
	2	1	1'	3	3'
	3	3'	3	1'	1
	2'	2	2'	2'	2
	3'	3	3'	3'	3
Resultado	1.100 €	100 €	1.120 €	1.100 €	1.100 €

Vemos que solamente se obtiene el resultado correcto cuando uno de los dos procesos realiza sus tres operaciones antes de que el otro lea el valor de la variable saldo, es decir, cuando no se entremezclan las operaciones de ambos procesos, lo que implica que los tres pasos mencionados se han de ejecutar de forma atómica.

El que puedan darse las situaciones erróneas anteriores se debe a que las rachas de ejecución de un proceso no son predecibles, puesto que dependen de las interrupciones que sufra el sistema, así como de los algoritmos de planificación que utilice el sistema operativo. Por tanto:

Advertencia 6.2. En principio, no se puede garantizar la velocidad relativa de los procesos concurrentes, es decir, no se puede predecir ni garantizar el orden relativo en que ejecutan los distintos accesos a los recursos compartidos. Para ello hay que utilizar mecanismos especiales que permiten garantizar la atomicidad.

La figura 6.4 muestra un ejemplo en el que dos *threads* ejecutan un código tan sencillo como `a++;` sobre una variable compartida a. Dado que la ejecución de dicho código requiere las tres instrucciones de máquina mostradas en la figura, dependiendo del orden relativo de ejecución de los *threads* el resultado será correcto o erróneo. Solamente se obtiene el valor correcto de 5 cuando cada *thread* ejecuta las tres instrucciones de máquina sin entremezclarse con las del otro, es decir, de forma atómica.

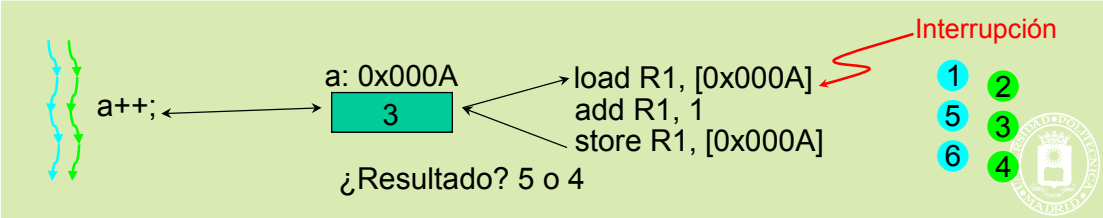


Figura 6.4 *Un código tan sencillo como `a++;` requiere varias instrucciones de máquina, por lo que presenta problemas de concurrencia. Si se produce una interrupción justo después del `load` de número 1 (thread azul) y se da control al otro thread (verde), el resultado final será erróneo.*

Consideremos, ahora, un sistema operativo que debe asignar un identificador de proceso (PID) a dos procesos en un sistema multiprocesador con dos procesadores. Esta situación se presenta en la figura 6.5. Cada vez que se crea un nuevo proceso el sistema operativo le asigna un identificador de proceso (PID). El valor del último PID asignado se almacena en un registro o posición de memoria A. Para asignar un nuevo PID el sistema operativo debe llevar a cabo las siguientes acciones:

- Leer de A el último PID asignado.
- Incrementar el valor del último PID. El nuevo valor será el PID a asignar al proceso.
- Almacenar el nuevo PID en A.

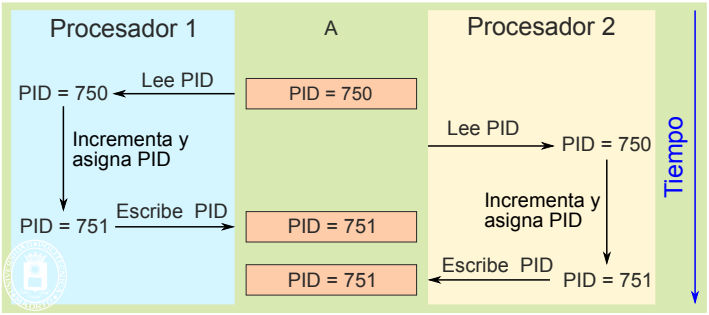


Figura 6.5 *Generación de PID en un sistema multiprocesador.*

Si las operaciones anteriores las ejecutase el sistema operativo en dos procesadores de forma simultánea sin ningún tipo de control, se podrían producir errores ya que se podría asignar el mismo PID a dos procesos distintos. Este problema se debe a que las acciones anteriormente descritas pueden producir, al igual que en los ejemplos anteriores, condiciones de carrera.

6.3.2. Sincronización

Los ejemplos de la sección anterior muestran la necesidad de sincronizar la ejecución de procesos, de forma que ejecuten siguiendo un orden adecuado para que las aplicaciones ejecuten correctamente. Existen dos formas de sincronización entre procesos concurrentes: la exclusión mutua y la sincronización condicional.

La **exclusión mutua** implica que solamente un **único** proceso ejecuta en cada instante la parte del programa que presenta condiciones de carrera. La exclusión mutua implica, por tanto, la atomicidad de esas partes del programa, que llamaremos secciones críticas, como veremos en la siguiente sección, de forma que no se entremezclen sus ejecuciones. Como veremos, la exclusión mutua exige **espera** cuando otro proceso está ejecutando la sección crítica.

La **sincronización condicional** implica que un proceso debe **esperar** a que se cumpla una cierta condición antes de poder continuar su ejecución. Por ejemplo, un proceso debe esperar a que otro genere una cierta información que necesita para seguir su ejecución.

Resumiendo, podemos decir que la sincronización introduce contención en los programas, puesto que han de esperar a que se den determinadas condiciones para poder ejecutar.

6.3.3. La sección crítica

En programación concurrente se denomina **sección crítica** al fragmento de código de un programa que accede a un recurso compartido que no debe ser accedido en cada instante por más de un flujo de ejecución (proceso o *thread*). Adicionalmente, se denomina **exclusión mutua** al requisito de garantizar que dos *threads* o procesos no puedan ejecutar simultáneamente en su sección crítica.

Una sección crítica es un segmento de código susceptible de contener una condición de carrera.

La figura 6.6 muestra tres procesos, uno con dos *threads* compartiendo una variable `a`. Los fragmentos de código que acceden a la variable constituyen las correspondientes secciones críticas.

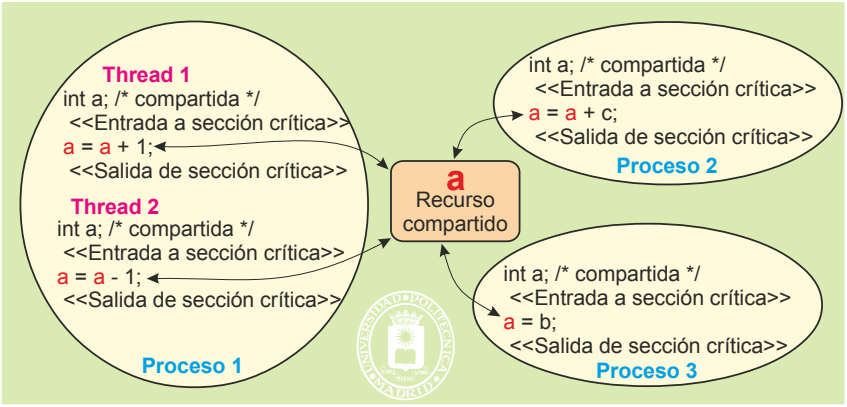


Figura 6.6 Procesos y threads compartiendo una variable `a`.

La figura 6.7 muestra 4 *threads* que comparten la variable `a`, sobre la que hacen un incremento. Para que el conjunto funcione correctamente es necesario que solamente uno de ellos ejecute dentro de su sección crítica. Se puede observar que los *threads* cían (2), verde (3) y amarillo (4), están bloqueados a la espera de que salga el rojo (1) de la sección crítica, en cuyo momento uno y solamente uno de los que esperan podrá entrar en la sección crítica.

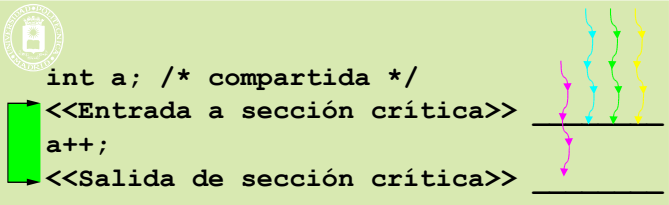


Figura 6.7: *Ejemplo de sección crítica.*

Para garantizar que los *threads* o procesos concurrentes ejecuten de forma ordenada y correcta, se ha de proteger la sección crítica con mecanismos de sincronización. Estos mecanismos deben garantizar las tres condiciones siguientes:

- **Exclusión mutua:** si un proceso está ejecutando código de la sección crítica, ningún otro lo podrá hacer. Esto implica que los *threads* o procesos deben esperar a que esté libre la sección crítica para poder entrar.
- **Progreso**: si ningún proceso se está ejecutando dentro de la sección crítica, la decisión de qué proceso entra en la sección se hará sobre los que desean entrar. Los procesos que no quieren entrar no pueden formar parte de esta decisión, por tanto, un *thread* o proceso fuera de la sección crítica no puede impedir que otro entre. Además, esta decisión debe realizarse en tiempo finito.
- **Espera acotada**: si y solamente si espero, entraré en tiempo finito. El mecanismo debe evitar la inanición, lo que ocurre si un *thread* o proceso queda en espera infinitamente.

Los mecanismos de sincronización detienen la ejecución, produciendo contención, lo que penaliza las prestaciones del sistema. Por ello, al programar aplicaciones concurrentes se debe:

- Retener los recursos el menor tiempo posible, para producir la menor contención posible.
- No dejar que los *threads* puedan quedar bloqueados dentro de la sección crítica.
- En definitiva: coordinar el acceso a recursos compartidos haciendo lo más ligero posible el código de las secciones críticas.

6.3.4. Interbloqueo

Un conjunto de procesos está en **interbloqueo** si cada proceso está esperando un recurso que sólo puede liberar (o generar, en el caso de recursos consumibles) otro proceso del conjunto.

No hay que confundir interbloqueo con inanición. La inanición ocurre por un problema de asignación de recursos no equitativa, en la que a un proceso no le llega nunca el turno.

Sobre los recursos se pueden considerar de forma genérica que los procesos disponen de las dos operaciones siguientes:

- **Solicitud** de un recurso.
  - Si el recurso no está disponible el proceso debe esperar a que el recurso se libere. La forma más eficiente de realizar esta espera es mediante espera pasiva, es decir, bloqueando la ejecución del proceso. Sin embargo, en algunos casos es necesario utilizar espera activa, en la que el proceso ejecuta un bucle de solicitud repetida del recurso hasta que se le concede.
  - Si el recurso está disponible, se le asigna.
- **Liberación** de un recurso. La liberación puede causar que se satisfaga una solicitud pendiente de otro proceso, provocando su desbloqueo.

Grafo de asignación de recursos

Una herramienta para analizar los interbloqueos es el grafo de asignación de recursos. Estos grafos tiene dos tipos de nodos: **nodos proceso**, que se representan con un círculo, y **nodos recurso**, que se representan mediante un cuadrado, como se muestra en la figura 6.8. En el caso de recursos múltiples, que no contemplaremos aquí, el recurso contiene un número que indica las unidades del recurso que existen.

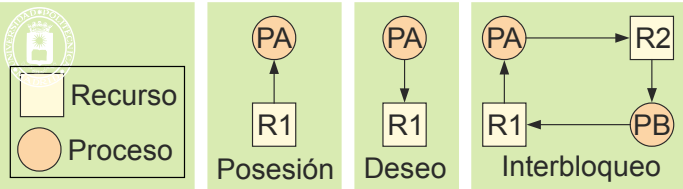


Figura 6.8 *Ejemplo de interbloqueo en el acceso a recursos compartidos.*

Además, existen dos tipos de flechas, las de posesión y las solicitud o deseo:

- El vector de posesión o de asignación va del recurso al proceso, indicando que el proceso tiene asignado dicho recurso.
- El vector de solicitud o deseo va desde el proceso al recurso deseado.

La figura 6.8 presenta el ejemplo de dos procesos PA y PB y dos recursos R1 y R2. Cada proceso necesita acceder a los dos recursos simultáneamente. La figura muestra que el proceso PA tiene posesión del recurso R1 y el proceso PB tiene posesión del recurso R2. Cada proceso tiene uno de los recursos, pero está esperando por el recurso que está en posesión del otro proceso. Esta situación da lugar a un interbloqueo permanente de los dos procesos, puesto que ninguno de los dos puede continuar su ejecución.

En el caso de recursos simples, las condiciones necesarias para que se produzca un interbloqueo son las siguientes:

- **Usu exclusivo**. Los recursos implicados se usan en exclusión mutua. Un recurso compartido está siempre disponible, por lo que no puede estar involucrado en interbloqueos.
- **Retención y espera**. Cuando no se puede satisfacer la petición de un proceso, éste se bloquea manteniendo los recursos que tenía previamente asignados. Se trata de una condición que refleja una forma de asignación que se corresponde con la usada prácticamente en todos los sistemas reales.
- **Sin expropiación**. No hay posibilidad de expropiar los recursos que tiene asignado un proceso. Un proceso sólo libera sus recursos voluntariamente.
- **Espera circular**. Debe existir una lista circular de procesos, tal que cada proceso en la lista esté esperando por uno o más recursos que tiene asignados el siguiente proceso. Esta espera se puede observar en el diagrama de asignación de recursos como un ciclo.

Tratamiento del interbloqueo

Las estrategias utilizadas para tratar el interbloqueo son las siguientes:

- **Algoritmo del avestruz**. Este algoritmo consiste en **ignorar los interbloqueos**. Aunque parezca sorprendente *a priori*, muchos sistemas operativos usan frecuentemente esta estrategia de "esconder la cabeza en el suelo o debajo del ala". Esta estrategia se justifica por la baja probabilidad de que se produzcan interbloqueos en la asignación implícita de recursos que hace el sistema operativo, unido a la complejidad o limitaciones que introducen el resto de las estrategias.
- **Detección y recuperación**: Los procesos realizan sus peticiones sin ninguna restricción pudiendo, por tanto, producirse interbloqueos. Se debe supervisar el estado del sistema para detectar el interbloqueo mediante algún algoritmo de detección. Nótese que la aplicación de este algoritmo supondrá un coste que puede afectar al rendimiento del sistema. Cuando se detecta, hay que eliminarlo mediante algún procedimiento de recuperación que, normalmente, conlleva una pérdida del trabajo realizado hasta ese momento por algunos de los procesos implicados.
  Las **transacciones**, que se analizan en la sección "6.8 Transacciones", permiten resolver el problema de los interbloqueos de la siguiente forma: cuando la transacción no puede obtener un recurso, porque está ocupado, la transacción fracasa, liberando todos los recursos que tiene asignados. De esta forma es imposible que se produzca la espera circular.
- **Prevención**: Este tipo de estrategias intenta eliminar el problema de raíz, fijando una serie de restricciones en el sistema sobre el uso de los recursos que aseguran que no se pueden producir interbloqueos. Nótese que estas restricciones se aplican a todos los procesos por igual, con independencia de qué recursos use cada uno de ellos. Esta estrategia suele implicar una infrautilización de los recursos, puesto que un proceso, debido a las restricciones establecidas en el sistema, puede verse obligado a reservar un recurso mucho antes de necesitarlo.
- **Evitación**: Esta estrategia evita el interbloqueo basándose en un conocimiento *a priori* de qué recursos va a usar cada proceso. Este conocimiento permite definir algoritmos que aseguren que no se produce un interbloqueo. Como ocurre con las estrategias de detección, a la hora de aplicar esta técnica, es necesario analizar la repercusión que tiene la ejecución del algoritmo de evitación sobre el rendimiento del sistema. Además, como sucede con la prevención, generalmente provoca una infrautilización de los recursos.

Veamos seguidamente ejemplos en los que se produce interbloqueo.

Supongamos dos procesos P1 y P2 que comparten dos semáforos S y Q (el concepto de semáforo se verá en la sección "6.7.2 Semáforos"). Supongamos que ejecutan el siguiente fragmento de código:

```
P0              P1
sem_wait(S)     sem_wait(Q)
sem_wait(Q)     sem_wait(S)  <=!!  ¡¡ Se produce interbloqueo!!
...             ...
sem_post(S)     sem_post(Q)
sem_post(Q)     sem_post(S)
```

Se generaría una situación de interbloqueo en el caso de que el proceso P0 espere al semáforo Q, mientras que P1 espere al semáforo S, estando ambos ocupados. Dependiendo del orden de ejecución de estos procesos, esta situación puede ser más o menos improbable. Los errores poco probables son muy complicados de depurar, puesto que suele ser muy difícil reproducir las condiciones de ejecución que los producen. El problema del ejemplo se evita reordenando el acceso a los recursos, por ejemplo, el orden en el que P1 solicita los semáforos. Sin embargo, en códigos más complejos puede ser difícil encontrar una reordenación adecuada.

Supongamos ahora el siguiente código:

```
…
pipe(pp);
read(pp[0],...);    ⇐!!
/* NO termina */
/* ¡Quedan escritores! */
…
```

El proceso crea un *pipe* e inmediatamente se pone a leer de él. Como no se ha escrito nada en el *pipe* el proceso queda bloqueado indefinidamente, puesto que el propio proceso sigue con el descriptor de fichero pp[1] abierto.

Finalmente, la figura 6.41, página 360, muestra otra situación de interbloqueo, que se produce por mal diseño del programa.

Livelock

El *livelock* es una forma de inanición similar al interbloqueo, con la diferencia de que los estados de los procesos envueltos en el *livelock* cambian constantemente con respecto al de los otros, sin que ninguno progrese.

Un ejemplo de *livelock* se produce si dos personas se encuentran en un pasillo estrecho y, amablemente, se corren para el mismo lado para dejar pasar al otro, pero terminan oscilando de un lado al otro simultáneamente sin que ninguno pueda avanzar.

6.4. DISEÑO DE APLICACIONES CONCURRENTES

En esta sección se plantea una sistemática para abordar el diseño de aplicaciones concurrentes. Consideramos el caso más común en el que el recurso que se comparte es información.

6.4.1. Pasos de diseño

Información compartida

El primer paso consiste en determinar qué información se comparte. Para ello hay que definir el tipo de dato y el número de instancias del mismo.

- Según la persistencia de la información podemos tener:
  - Datos consumibles. Es decir datos que una vez utilizados ya no sirven y su soporte se puede sobrescribir. Un ejemplo de dato consumible es un mensaje, que una vez leído se puede sobrescribir.
  - Datos permanentes, que tienen validez durante toda la ejecución de los procesos concurrentes. Ejemplo de datos permanentes son los encontrados en una base de datos.
- Según el tipo de dato podemos tener:
  - Variables simples.
  - Registros o estructuras de tamaño fijo.
  - Registros o estructuras de tamaño variable.
- En el caso de tener varias instancias de información compartida, éstas se pueden organizar en una estructura común, que puede ser, entre otras alternativas, una de las siguientes:
  - **Tabla** o **vector**. Esta solución suele ser conveniente cuando las informaciones compartidas son permanentes. Por ejemplo la lista de referencias de los libros de una biblioteca.
  - ***Buffer* circular**. Esta solución es especialmente apropiada cuando la información compartida es consumible. Por ejemplo, los paquetes de información que tiene que manejar un encaminador. Estos paquetes son recibidos y almacenados por un proceso receptor para su posterior envío por el proceso emisor. Una vez enviado (consumido) el paquete ya no tiene validez dentro del encaminador.
  - **Lista enlazada**.

Soporte de la información

El siguiente paso es definir el soporte que va a tener dicha información. Las alternativas son las siguientes:

- **Variables comunes** definidas como globales. Esta solución es válida si se trata de *threads* de un mismo proceso.
- Región de **memoria compartida**. Esta solución es válida para procesos locales, es decir que ejecuten en un mismo computador.
- **Fichero**. Esta alternativa es válida para procesos y ficheros locales y remotos.
- **Mecanismo de comunicación** con *buffer*. Por ejemplo un *pipe*. En realidad, internamente el *pipe* no es más que un *buffer* circular.

Determinar las condiciones de sincronización y la información de control

En este paso hay que definir las condiciones por las que un proceso debe esperar para poder seguir su ejecución, hay que definir lo que llamaremos las **condiciones de entrada**. Dicha definición implica determinar las variables de control necesarias, así como los valores que las mismas deben tener para cumplir las condiciones de entrada.

También hay que definir las modificaciones que se han de producir en las variables de control antes de que el proceso acceda al recurso compartido.

Finalmente, hay que definir el proceso de **salida**, es decir, las acciones que se deben realizar al finalizar el acceso al recurso compartido.

Los dos tipos de información de control más frecuentes son:

- La **variable booleana** que, según su estado, permita el acceso a la información compartida o no lo permita.
- El **contador** que, según su valor, permita o no el acceso.

La información de control es, a su vez, información compartida, que se debe acceder con exclusión mutua. Por tanto, existirán unas secciones críticas, dentro de las cuales se realice el acceso a las mismas.

Definir los mecanismos de sincronización y comunicación

Primero, hay que seleccionar el tipo o los tipos de mecanismos que se van a utilizar. Según veremos en este capítulo existen varios tipos de mecanismos de sincronización y comunicación con distintos campos de aplicación.

Seguidamente, hay que definir las instancias concretas utilizadas, planteando el diseño detallado de las aplicaciones concurrentes.

En general, estos mecanismos son atómicos, lo que implica que garantizan la exclusión mutua.

6.4.2. Esquemas de acceso a recursos compartidos

Siguiendo lo indicado en la sección "6.3.2 Sincronización", la figura 6.9 se presentan dos esquemas básicos de acceso al recurso protegido.

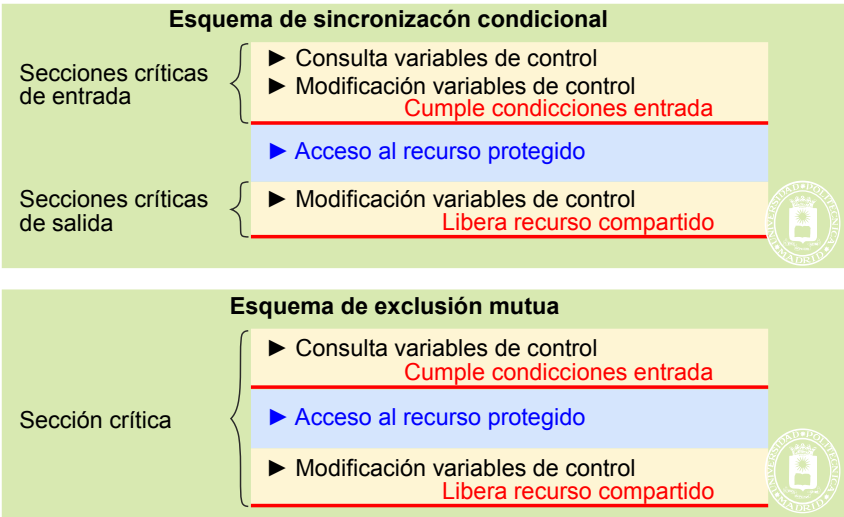


Figura 6.9 *Esquemas básicos de acceso al recurso protegido.*

En el **esquema de sincronización condicional** se contemplan las tres fases siguientes:

- Fase de **sincronización** o **entrada**. En esta fase se analizan las variables de control para ver si se cumple la condición de sincronización. También es posible que se modifique alguna variable de control, para indicar que el proceso va a proseguir su ejecución. El acceso para consulta y modificación ha de estar protegido, lo que exige encapsular esta fase en una o varias secciones críticas.
  El proceso permanece en esta fase hasta que se cumplan las condiciones de sincronización.
- Fase de **acceso al recurso protegido**. El acceso puede ser **compartido**, como ocurre en el caso de los procesos que son solamente lectores, pero puede ser **exclusivo** , caso en el que solamente podrá haber un proceso haciendo simultáneamente el acceso.
- Fase de **salida**. Terminado el acceso al recurso protegido, se entra en una fase de actualización de las variables de control para reflejar que el proceso ha terminado con el acceso protegido, por lo que otro proceso en espera podría continuar su ejecución. Esta fase también debe ser protegida por la o las correspondientes secciones críticas.

En el **esquema de exclusión mutua** las tres fases anteriores se incluyen en una sola sección crítica de forma que solamente es necesario la modificación de las variables de control en el código de salida. Esta solución no sirve para acceso compartido.

6.5. MODELOS DE COMUNICACIÓN Y SINCRONIZACIÓN

Las necesidades de comunicación y sincronización entre procesos se plantean en una serie de situaciones clásicas de comunicación y sincronización. Estos modelos están presentes en numerosos escenarios reales, por lo que son una base interesante para plantear el diseño de aplicaciones concurrentes.

6.5.1. Productor-consumidor. Modela comunicación

El modelo productor-consumidor es uno de los modelos más habituales en aplicaciones concurrentes. En este tipo de problemas uno o más procesos, que se denominan *productores,* generan cierto tipo de datos, que llamaremos despachos, que son utilizados o consumidos por otros procesos, que se denominan *consumidores*.

En esta clase de modelos es necesario disponer de algún mecanismo de comunicación y/o almacenamiento que permita a los procesos productor y consumidor intercambiar los despachos. Ambos procesos, además, deben sincronizar su acceso a dicho mecanismo para que la interacción entre ellos no sea problemática: cuando el mecanismo se llena, el o los procesos productores deberán quedarse bloqueados hasta que haya espacio para seguir insertando despachos. A su vez, el o los procesos consumidores deberán quedarse bloqueados cuando el mecanismo esté vacío, ya que, en este caso, no podrán continuar su ejecución, al no disponer de despachos que consumir. Por lo tanto, este tipo de modelo requiere servicios para que los procesos puedan comunicarse y servicios para que se sincronicen a la hora de acceder al mecanismo de comunicación.

El recurso compartido son los despachos que genera el productor y que sirven de entrada al consumidor. Es un tipo de recurso consumible, puesto que uno vez leído por el consumidor se puede desechar.

En la figura 6.10 se representa la estructura genérica de este tipo de procesos, así como la solución muy popular de emplear un *buffer* circular para realizar la comunicación entre los procesos productores y los consumidores

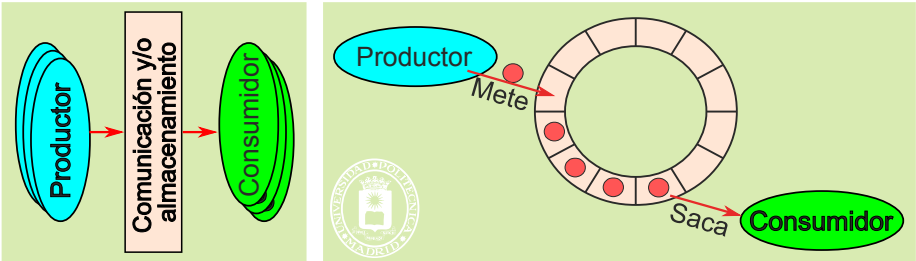


Figura 6.10 *Estructura general de procesos productores-consumidores y esquema con buffer circular.*

Solución mediante *buffer* circular

Con gran frecuencia se utiliza un buffer circular para almacenar los despachos. En este tipo de problemas es necesario evitar que ocurra alguna de las siguientes situaciones:

- Un consumidor saque despachos cuando el *buffer* está vacío.
- Un productor coloque despachos en el *buffer* cuando éste se encuentra lleno.
- Un productor sobrescriba un despacho que todavía no ha sido sacado del *buffer.*
- Un consumidor saque despachos del *buffer* que ya fueron sacados con anterioridad.
- Un consumidor saque un despacho determinado mientras un productor lo está insertando.

El almacenamiento compartido es el mencionado *buffer* circular. Las variables de control típicas dependen de si los despachos son de tamaño fijo o variable:

- Caso de despachos de tamaño fijo. El *buffer* permitirá almacenar N despachos. Las variables de control pueden ser:
  - Contador con el número de huecos libres en el *buffer* circular.
  - Puntero del primer hueco libre para introducir un nuevo despacho. Una vez escrito el nuevo despacho se debe incrementar en 1 dicho puntero. Además, hay que restar 1 al contador de huecos libres.
  - Puntero del despacho más antiguo, que será el que debe leer el consumidor. Una vez leído el despacho se debe incrementar en 1 dicho puntero. Además, hay que sumar 1 al contador de huecos libres.
- Caso de despachos de tamaño variable. El *buffer* tendrá un tamaño de P bytes y cada despacho deberá indicar su tamaño. Las variables de control pueden ser:
  - Espacio libre del *buffer* expresado en bytes.
  - Puntero al primer byte libre, para que el productor escriba, a partir de ese punto, el nuevo despacho. Una vez escrito el despacho, hay que incrementar el puntero de acuerdo al tamaño del mismo. Además, hay que restar ese mismo valor del espacio libre del *buffer.*
  - Puntero del primer byte del despacho más antiguo, para que el consumidor pueda leerlo. Una vez leído el despacho, hay que incrementar el puntero de acuerdo al tamaño del mismo. Además, hay que restar ese mismo valor del espacio libre del *buffer.*

Solución con mecanismo de comunicación con *buffer*

Otro esquema usual es conectar productor y consumidor mediante un mecanismo de comunicación con *buffer*, como podría ser un *pipe*. El problema queda prácticamente resuelto por el mecanismo de comunicación, gracias a que estos mecanismos bloquean al emisor cuando están llenos y también bloquean al receptor cuando están vacíos.

Por un lado, el o los productores irán enviando los despachos a través del mecanismo de comunicación. Dicho mecanismo bloquea a los emisores si no tiene espacio libre en su *buffer*.

Por otro lado, el o los consumidores irán leyendo despachos del mecanismo de comunicación. Dicho mecanismo bloquea a los consumidores si no tiene datos disponibles.

Es de observar que, en este caso, no son necesarias variables de control, puesto que el propio mecanismo de comunicación se encarga de sincronizar productores y consumidores.

6.5.2. Lectores-escritores. Modela acceso a recurso compartido

En este modelo existe un determinado almacenamiento (véase la figura 6.11), que va a ser utilizado y compartido por una serie de procesos concurrentes. Algunos de estos procesos sólo van a acceder al almacenamiento sin modificarlo, mientras que otros van a acceder al almacenamiento para modificar su contenido. Esta actualización implica leerlo, modificar su contenido y escribirlo. A los primeros procesos se les denomina *lectores*, y a los segundos se les denomina *escritores*. En este tipo de problemas existe la siguiente serie de restricciones que han de cumplirse:

- Sólo se permite que un escritor tenga acceso al almacenamiento al mismo tiempo. Además, mientras el escritor esté accediendo al almacenamiento, ningún otro proceso lector ni escritor podrá acceder a él. El acceso del escritor es exclusivo. De esta forma se evita que un lector lea el almacenamiento a medio modificar o que se mezclen las modificaciones de dos escritores.
- Se permite, sin embargo, que múltiples lectores tengan acceso al almacenamiento mientras no haya ningún escritor, ya que ellos nunca van a modificar el contenido del almacenamiento.

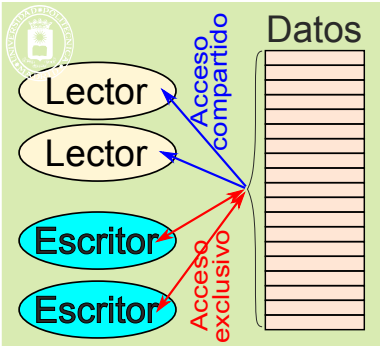


Figura 6.11 *Procesos lectores y escritores, mostrando el tipo de acceso que requieren.*

En este tipo de aplicaciones es necesario disponer de servicios de sincronización que permitan que los procesos lectores y escritores se sincronicen adecuadamente en el acceso al almacenamiento.

Este modelo es típico en aplicaciones de bases de datos. Por ejemplo, en la base de datos de una biblioteca, varios usuarios pueden estar consultando el estado de las reservas de un determinado libro y otros pueden estar interesados en reservar el libro. El sistema ha de conseguir que aquellos usuarios interesados en reservar el libro (usuarios escritores) lo hagan de forma exclusiva.

Las características generales de este tipo de problema son las siguientes

- El almacenamiento compartido en este tipo de problemas suele ser el de un conjunto de registros, que pueden estar almacenados en memoria, en un fichero, en una base de datos, etc.
- Una primera e importante decisión es determinar la granularidad del control, que determina el elemento del almacenamiento sobre el que se hace acceso concurrente. Se pueden contemplar las dos situaciones siguientes:
  - Grano grueso. Por ejemplo, considerar todo el almacenamiento como elemento concurrente. En este caso, sólo se podría tener un único escritor en cada instante. La solución es sencilla, pero produce una gran contención si existen muchos escritores.
  - Grano fino. Por ejemplo, considerar cada registro del almacenamiento como elemento concurrente. En este caso, se podría tener simultáneamente un escritor por cada registro del almacenamiento. La contención será mucho menor, pero la memoria necesaria para las variables de control viene determinada por el número de registros del almacenamiento.
- La pareja de variables de control típicas por cada elemento de acceso concurrente son las siguientes:
  - Número de lectores que están accediendo simultáneamente al elemento concurrente.
  - Número de escritores que están accediendo simultáneamente al elemento concurrente. Esta variable solamente admite el valor 0 y 1, por lo que también se puede considerar una variable booleana que indique si hay un escritor accediendo.

En el caso de grano fino a nivel de registro, hace falta una pareja de variables por registro, lo que necesita más memoria par el control de acceso, pero reduce la contención.

6.5.3. Filósofos comensales. Modela el acceso a recursos limitados

Este modelo aparece en aquellas situaciones en las que existen varios procesos accediendo a un conjunto de recursos limitados. Para modelar esta situación utilizaremos el problema de los filósofos comensales, que fue propuesto por Dijkstra. En este problema existen varios filósofos (véase la figura 6.12) que se pasan toda su vida pensando y comiendo, actividades que realizan de forma independiente unos de otros. Los filósofos comparten una misma mesa con comida en el centro. Cada filósofo tiene su propio plato y dispone de dos palillos, uno situado a su derecha y otro a su izquierda. Para poder comer, cada filósofo necesita acceder a los dos palillos simultáneamente. Cuando tiene los dos palillos come sin soltarlos en ningún momento y cuando acaba de comer vuelve a dejar los palillos en la mesa, disponiéndose de nuevo a pensar. En este tipo de problemas hay que verificar dos cosas:

- Que ningún filósofo se muera de hambre, hay que asegurar que todo filósofo que quiere comer en algún momento come y no espera de forma indefinida.
- Que no se producen interbloqueos, que no se da una situación en la que todos los filósofos esperan de forma indefinida por un palillo que tiene otro filósofo.
- Que no se produce un *livelock*. Esto podría ocurrir si el filósofo que no puede conseguir el segundo palillo soltase el primero, para permitir que otro comiese. En efecto, si todos ellos, repetidamente y simultáneamente, cogen el palillo de la izquierda y lo sueltan al no poder coger el de la derecha se produce un *livelock*.

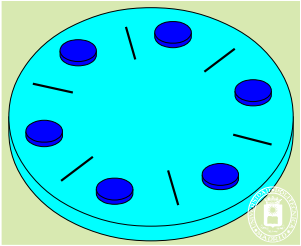


Figura 6.12 *El problema de los filósofos comensales.*

En la vida real los filósofos son procesos que necesitan para su ejecución un conjunto limitado de recursos de forma simultánea.

6.5.4. Modelo cliente-servidor

En el modelo cliente-servidor, los procesos llamados servidores ofrecen una serie de servicios a otros procesos que se denominan clientes (véase la figura 6.13). El proceso servidor puede residir en la misma máquina que el cliente o en una distinta, en cuyo caso la comunicación deberá realizarse a través de una red de interconexión. Muchas aplicaciones y servicios de red, como el correo electrónico, la transferencia de ficheros o la Web, se basan en este modelo.

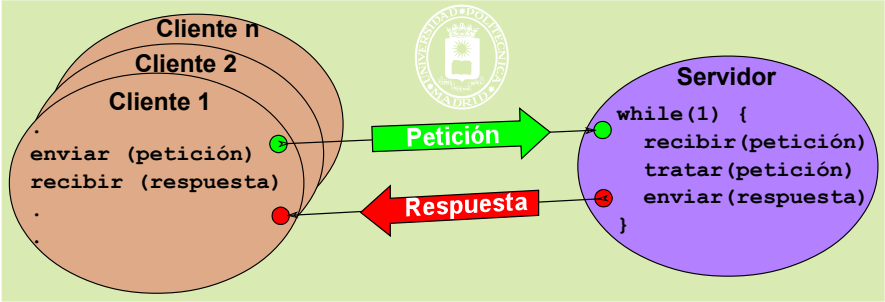


Figura 6.13 *Comunicación cliente-servidor.*

En este tipo de aplicaciones es necesario que el sistema operativo ofrezca servicios que permitan comunicarse a los procesos cliente y servidor. Cuando los procesos ejecutan en la misma máquina, se pueden emplear técnicas basadas en memoria compartida o ficheros. Sin embargo, el modelo cliente-servidor suele emplearse en aplicaciones que se ejecutan en computadores que se encuentran conectados mediante una red y que no comparten memoria, por lo que se usan técnicas de comunicación basadas en paso de mensajes (el paso de mensajes se analiza más adelante). Aparte de resolver el problema básico de comunicación, en este tipo de aplicaciones pueden existir diversos procesos clientes accediendo de forma concurrente al servidor, por ello es necesario disponer en el servidor de mecanismos que permitan ejecutar las peticiones concurrentes de los diversos clientes de forma coordinada y sin problemas.

6.5.5. Modelo de comunicación entre pares "Peer-to-peer" (P2P)

El modelo cliente-servidor clásico es centralizado, en el que existen uno o varios nodos servidores que atienden a los nodos clientes. Sin embargo, en el modelo P2P todos los nodos son iguales, actuando simultáneamente como clientes y servidores, por lo que se utilizan los términos "entre iguales" o "entre pares".

Como muestra la figura 6.14 los nodos están interconectados mediante una red mallada, normalmente superpuesta sobre Internet.

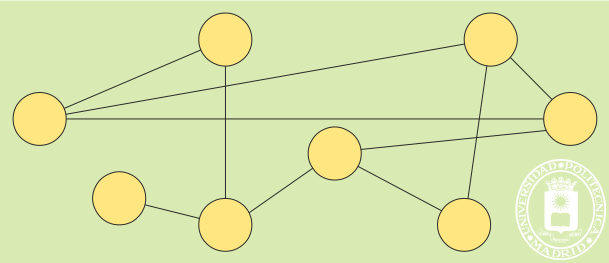


Figura 6.14 *Los nodos de una red P2P se conectan en red mallada actuando cada nodo como cliente y como servidor.*

El recurso compartido suele ser información, estando dicho recurso distribuido entre los nodos de la red P2P. De acuerdo a la organización de la información se distinguen dos tipos de redes:

- Red **no estructurada**. En estas redes, que son las más comunes, cada miembro almacena la información que quiere, durante el tiempo que quiere.
- Red **estructurada**: En estas redes la información está repartida entre los nodos según determinados criterios. Se exige, por tanto, una administración central que asigne la información al nodo. Este podría ser el caso de una red implantada por una empresa para su uso interno.

Estas redes platean el reto de la gestión del directorio (para saber qué información está disponible) y del encaminamiento (para saber dónde se puede encontrar). Las alternativas son las siguientes:

- Gestión **centralizada**. El problema es que deja de ser un sistema puro P2P, puesto que se requieren unos nodos especiales para albergar y servir esta información. Ejemplos: Napster y Audiogalaxy.
- Gestión totalmente **distribuida**. Esta solución da lugar a un sistema P2P puro, pero es menos eficiente a la hora de buscar contenidos. Ejemplos: Ares Galaxy, Gnutella, Freenet y Kademlia.
- Gestión **híbrida**. Platean una solución intermedia entre centralizado y totalmente distribuido. Ejemplos: Bittorrent, eDonkey2000, eMule y Direct Connect.

Las características de las redes P2P son las siguientes:

- **Escalabilidad**. En los sistemas cliente-servidor el crecimiento viene limitado por la capacidad del servidor central. En las redes P2P, en principio no hay límite, cuantos más nodos estén conectados mejor. De hecho hay redes con cientos de millones de nodos.
- **Robustez**. En los sistemas P2P puros con gestión distribuida, no hay ningún punto singular de fallo, es decir, no hay ningún elemento del sistema cuyo fallo impida su funcionamiento. Si falla algún elemento no pasa nada, puesto que la información está replicada en multitud de nodos, por tanto, cuantos más nodos haya, habrá más replicación de la información y el sistema será más robusto.
- **Descentralizado**. En los sistemas P2P puros no hay ningún nodo con una función especial, por lo que es un sistema totalmente descentralizado.
- **Costes distribuidos**. Cada participante de la red contribuye con los costes de su nodo, por lo que los costes son totalmente distribuidos.
- **Anonimato**. En las redes P2P se pude conseguir el anonimato del autor de un contenido, del editor, del lector, del servidor que lo alberga y de la petición para encontrarlo.
- **Seguridad**. La seguridad no es fácil de conseguir en un sistema P2P. Es difícil evitar nodos maliciosos, contenidos incorrectos o infectados, el espionaje de las comunicaciones, etc.

6.6. MECANISMOS DE COMUNICACIÓN

Para resolver los problemas que plantea la ejecución concurrente de procesos, el sistema operativo ofrece una serie de servicios que permiten a los procesos comunicarse y sincronizarse.

Los mecanismos de comunicación hacen posible que los procesos intercambien datos entre ellos. Los principales mecanismos de comunicación que ofrecen los sistemas operativos son los siguientes:

- Variables en memoria compartida.
- Ficheros.
- Tuberías.
- Paso de mensajes.
- *Sockets*

En las siguientes secciones se describen cada uno de estos mecanismos.

Aclaración 6.1. En general, y como se verá en las próximas secciones, un mecanismo de comunicación también se puede utilizar para sincronizar procesos.

En una comunicación intervienen las tres entidades siguientes:

- El **mensaje** que se envía a través de los mecanismos de comunicación.
- El **emisor** que es la entidad que envía el mensaje. El emisor dispone de un punto final (*endpoint*) que permite el envío de mensajes.
- El **receptor** que es la entidad que recibe el mensaje. El receptor dispone de un punto final (*endpoint*) que permite la recepción de mensajes.

Direccionamiento

Tanto el emisor como el receptor tienen una dirección que permite identificarles unívocamente. Las características más importantes del direccionamiento son las siguientes:

- **Tipo**. Existen los siguientes tipos de direccionamiento.
  - **Sin nombre**: Para utilizar el mecanismo se utiliza el identificador obtenido al crear el mecanismo. Este identificador ha se ser heredado por los otros procesos que lo deseen utilizar.
  - **Nombre textual** o simbólico. Por ejemplo: "/home/jfelipe/mififo" o "www.fi.upm.es". El esquema de nombrado que se sigue es jerárquico en árbol, por las ventajas que tiene dicho esquema, tal y como se vio en la sección "5.7 Directorios".
  - **Nombre físico**. Por ejemplo: "IP = 138.100.8.100 TCP = 80"
- **Ámbito**. Espacio en el que se aplica el esquema de direccionamiento. El ámbito puede ser:
  - Procesos **emparentados** (creador y sus descendientes).
  - **Local**: Procesos que ejecutan de el mismo computador, bajo un único sistema operativo.
  - **Remoto**: Procesos en máquinas distintas, cada una bajo su propio sistema operativo.

Servidor de nombres

El servidor de nombres es un proceso servidor al que se le presenta un nombre textual y lo convierte en el correspondiente nombre físico. El servidor de nombres puede ser:

- **Local**: Servidor de nombres del servidor de ficheros.
- **Remoto**: Servidor DNS de Internet.

Características de los mecanismos de comunicación

Las características del mecanismo de comunicación al crearlo (o abrirlo) son las siguientes:

- Según la forma de **nombrarlo** en la solicitud de creación:
  - **Sin nombre**. No se especifica ningún nombre, como ocurre con el *pipe*.
  - Con **nombre local**. Se especifica un nombre de la jerarquía de nombres del sistema de ficheros local, lo que ocurre con el FIFO.
  - Con **nombre físico**. Se especifica el nombre físico o de red, como ocurre con el *socket*.
- Según el **identificador** devuelto para utilizar el mecanismo
  - Cuando el mecanismo lo provee el sistema operativo, la solicitud de creación devuelve un descriptor de fichero (UNIX) o un manejador (Windows), de igual forma que al abrir un fichero.
  - Cuando el mecanismo lo provee una biblioteca, el identificador devuelto será un identificador propio de la biblioteca, que no tiene nada que ver con los identificadores suministrados por el sistema operativo.

Las características del mecanismo de comunicación al usarlo a través del identificador obtenido en su creación son las siguientes:

- Según el **flujo de datos** que permita el mecanismo, éste puede ser:
  - **Unidireccional**. El mecanismo solamente permite comunicación del extremo emisor al extremo receptor. Si se desea comunicación bidireccional es necesario establecer dos mecanismos, uno en cada sentido.
  - **Bidireccional**. En este caso, los dos extremos de la comunicación son a la vez emisores y receptores, por lo que la información puede fluir en ambos sentidos.
- Dependiendo de la **capacidad de memoria** del mecanismo se tienen las alternativas siguientes:
  - **Sin *buffering***. El mecanismo no tiene memoria, por lo que no puede almacenar mensajes.
  - **Con *buffering***. El mecanismo puede almacenar mensajes según la capacidad de la memoria que se le asigne.

- Dependiendo de si **bloquea** o no a los procesos cuando no se puede realizar la comunicación, el mecanismo puede ser:
  - **Síncrono** o bloqueante. El mecanismo bloquea al proceso hasta que pueda enviar o recibir el mensaje.
  - **Asíncrono** o no bloqueante. Si no se puede enviar o recibir el mensaje, el mecanismo no bloquea al proceso y le devuelve un aviso indicando la situación encontrada.

6.6.1. Comunicación remota: Formato de red

No todas las arquitecturas de procesador representan los números enteros de la misma forma, puesto que unos utilizan formato *little-endian*[3], mientras que otros usan *big-endian*.

Cuando se transmiten bytes esta diferencia no presenta ningún problema, pero si se transmiten enteros sí que surgen problemas. Especialmente delicado es el caso de las direcciones físicas, que suelen expresarse con enteros, puesto que si se envían en formato *little-endian*, los computadores que usan *big-endian* las interpretarán erróneamente, con lo que el mensaje no llegará al destinatario correcto.

Por esta razón, se ha definido un formato de red independiente de la arquitectura del procesador. Por ejemplo, el campo **`sockaddr_in.sin_port`**, que indica el número de puerto de un *socket*, debe estar en formato red, como veremos al analizar los *sockets*.

Para transmitir correctamente los enteros hay que codificarlos en formato de red y decodificarlos del formato de red. Para ello, se utilizan las funciones siguientes:

- **`htonl()`**. Convierte un entero largo (32 bits) del formato del computador donde se ejecuta la función al formato de red.
- **`htons()`**. Convierte un entero corto (16 bits) del formato del computador donde se ejecuta la función al formato de red.
- **`htonl()`**. Convierte un entero largo (32 bits) del formato de red al del formato del computador en el que se ejecuta la función.
- **`htons()`**. Convierte un entero corto (16 bits) del formato de red al del formato del computador en el que se ejecuta la función.

La comunicación de datos de otros tipos básicos simples, como son los datos en coma flotante, o de estructuras exige otros métodos de conversión que no son estándar. Por simplicidad, muchos protocolos usan texto como formato independiente y transmiten otros datos sin formato específico en binario (8 bits)

6.6.2. Memoria compartida

La memoria compartida es un paradigma que permite comunicar a procesos que ejecutan en la misma máquina, bien sea un monoprocesador o un multiprocesador. Con este modelo de comunicación un proceso almacena un valor en una determinada variable, y otro proceso puede acceder a ese valor sin más que consultar la variable. De esta forma se consigue que los dos procesos puedan comunicarse entre ellos. La memoria compartida no ofrece sincronización, por lo que es necesario utilizar un mecanismo adicional para ello.

Los *threads* que se crean dentro de un proceso comparten memoria de forma natural, y utilizan ésta como mecanismo de comunicación. En el ejemplo del productor-consumidor con semáforos del programa 6.8, página 357, se emplea esta técnica para comunicar al proceso productor y consumidor a través de un *buffer* común.

Cuando se quiere emplear memoria compartida entre procesos pesados (creados con `fork` en UNIX), es necesario recurrir a servicios ofrecidos por el sistema operativo, de forma que, los procesos que se quieren comunicar, creen un segmento de memoria compartida al que ambos pueden acceder a través de posiciones de memoria situadas dentro de su espacio de direcciones.

En la figura 6.15 se representa el concepto de segmento de memoria compartida, ya presentado en el capítulo "4 Gestión de memoria". Algún proceso debe encargarse de crear ese segmento, mientras que el resto de procesos que quieran utilizarlo simplemente tienen que acceder a él. Cada uno de los procesos puede acceder a este segmento de memoria compartida, pero utilizando direcciones relativas al comienzo de la región.

3 En el formato *little-endian* el byte menos significativo del dato, por ejemplo de un entero, ocupa la dirección de memoria de menor valor. Por el contrario, en el formato *big-endian* el byte menos significativo ocupa la dirección de memoria de mayor valor.

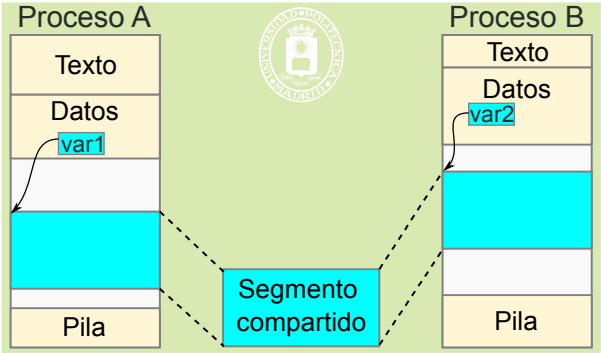


Figura 6.15 *Memoria compartida entre procesos. La ubicación de la región compartida puede ser diferente en cada proceso, por lo que solamente deben utilizarse direcciones relativas al comienzo de la región. El proceso A accede al segmento de memoria compartida utilizando la dirección almacenada en la variable de tipo puntero* `var1`*, mientras que el proceso B lo hace a través de la dirección almacenada en la variable de tipo puntero* `var2`*. La variable* `var1` *es una variable del espacio de direcciones del proceso A y* `var2` *es una variable del espacio de direcciones del proceso B.*

6.6.3. Comunicación mediante ficheros

Un fichero es un mecanismo que puede emplearse para comunicar procesos. Por ejemplo, un proceso puede escribir datos en un fichero y otro puede leerlos. El empleo de ficheros como mecanismo de comunicación presenta las siguientes ventajas:

- Permite comunicar a un número potencialmente ilimitado de procesos. Basta con que los procesos tengan permisos para acceder al fichero.
- Los servidores de ficheros ofrecen servicios sencillos y fáciles de utilizar.
- Ofrece un modelo muy desacoplado de los procesos y fácil de usar, permitiendo la comunicación, por ejemplo, de aplicaciones para las que no se dispone del código, por lo que no se pueden modificar.
- Ofrece persistencia en los datos que se quieren comunicar. El hecho de que los datos a comunicar se almacenen en ficheros hace que los datos sobrevivan a la ejecución o posibles fallos de los procesos.

Sin embargo, este mecanismo presenta una serie de inconvenientes que hacen que no sea un mecanismo de comunicación ampliamente utilizado. Se trata de un mecanismo empleado fundamentalmente para compartir información. Los inconvenientes son:

- Es un mecanismo bastante poco eficiente, puesto que la escritura y lectura en disco es lenta.
- Necesitan algún otro mecanismo que permita que los procesos se sincronicen en el acceso a los datos almacenados en un fichero. Por ejemplo, es necesario contar con mecanismos que permitan indicar a un proceso cuándo puede leer los datos de un fichero. Aunque para realizar esta sincronización se puede emplear cualquiera de los mecanismos que se van a ver en este capítulo, la mayoría de los sistemas de ficheros ofrecen cerrojos sobre ficheros (que analizaremos más adelante) que permiten bloquear el acceso al fichero o a partes del fichero.
- El almacenamiento puede venir limitado por el tamaño del dispositivo subyacente. Otros mecanismos de comunicación, al no almacenar toda la información, no presentan este problema.

6.6.4. Tubería o *pipe*

Una tubería o *pipe* es un mecanismo que tiene las siguientes propiedades

- Es un mecanismo de **comunicación sin nombre**.
- Por lo que solamente se puede usar entre procesos que **heredan** el mecanismo.
- Se **identifica** mediante dos descriptores de fichero (UNIX) o dos manejadores (Windows).
- Cada proceso debe **cerrar** los descriptores o manejadores no utilizados.
- Es un mecanismo con ***buffering*** (típicamente tiene un *buffer* de 4 KiB) y **unidireccional**.

Conceptualmente, cada proceso ve la tubería como un conducto con dos extremos, uno de los cuales se utiliza para escribir o insertar datos y el otro para extraer o leer datos de la tubería. Las operaciones de lectura y escritura se realizan con los mismos servicios que para los ficheros, pero con importantes peculiaridades que se detallan más adelante. Además, se garantiza que lecturas y escrituras sean **atómicas**, siempre que no se llene el *buffer* interno de la tubería.

El flujo de datos de las tuberías es unidireccional y con funcionamiento *first-in first-out*, esto quiere decir que los datos se extraen de la tubería (mediante la operación de lectura) en el mismo orden en el que se insertaron (mediante la operación de escritura). Aunque lo más común es utilizar una tubería para comunicar dos procesos, puede haber múltiples procesos lectores y escritores. La figura 6.16 representa cuatro procesos que se comunican de forma unidireccional utilizando una tubería. Cuando se desea disponer de un flujo de datos bidireccional es necesario crear dos tuberías como se muestra en la figura 6.17.

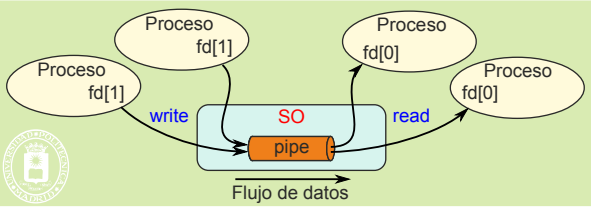


Figura 6.16 *Comunicación unidireccional empleando una tubería.*

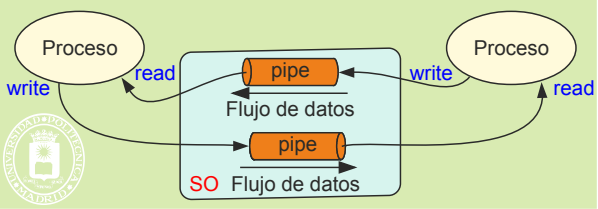


Figura 6.17 *Comunicación bidireccional empleando dos tuberías.*

Escritura en una tubería

Una escritura sobre una tubería introduce los datos de forma ordenada, uno detrás de otro. La semántica de esta operación es la siguiente:

- Si la tubería se encuentra llena o se llena durante la escritura, la operación **bloquea** al proceso escritor hasta que se pueda completar.
- Si no hay ningún proceso con la tubería abierta para lectura, la operación de escritura devuelve un error, que en el caso de UNIX consiste en recibir una señal SIGPIPE.
- Una operación de escritura sobre una tubería se realiza de forma **atómica**, es decir, si dos procesos intentan escribir de forma simultánea en una tubería, sólo uno de ellos lo hará, el otro se bloqueará hasta que finalice la primera escritura.

Lectura de una tubería

Una lectura de una tubería obtiene los datos almacenados en la misma en el orden en que fueron introducidos. Además, estos datos se eliminan de la tubería. Las operaciones de lectura siguen la siguiente semántica:

- Si la tubería está vacía, la llamada **bloquea** el proceso hasta que algún proceso escriba datos en la misma.
- Si la tubería almacena M bytes y se quieren leer n bytes, entonces:
  - Si $M \geq n$, la llamada devuelve n bytes y elimina de la tubería los datos leídos.
  - Si $M < n$, la llamada devuelve M bytes y elimina los datos disponibles en la tubería.
- Si no hay escritores y la tubería está vacía, la operación devuelve fin de fichero. Es decir, devuelve 0 bytes leídos y no bloquea al proceso. De esta forma el proceso lector sabe que ya no habrá nunca datos en la tubería.
- Al igual que las escrituras, las operaciones de lectura sobre una tubería son **atómicas**.

Como puede apreciarse, una lectura de una tubería nunca bloquea al proceso si hay datos disponibles en la misma.

Aclaración 6.2. En general, la atomicidad en las operaciones de lectura y escritura sobre una tubería se asegura siempre que el número de datos involucrados en las anteriores operaciones quepa en el tamaño del *buffer* interno de la misma.

Desde el punto de vista de su implementación, una tubería es un pseudofichero mantenido en memoria por el sistema operativo. Dentro del sistema operativo, una tubería se implementa normalmente en memoria como un *buffer* circular con un tamaño típico de 4 KiB. El sistema operativo necesita mantener para cada tubería los tres valores siguientes: la dirección de memoria donde comienza el *buffer* circular (`base`), la dirección de memoria del primer byte a leer en las operaciones de lectura (`inicio`) y la cantidad de bytes almacenados en la tubería (`longitud`). Con esta información, las escrituras se realizan en la posición de memoria dada por `base+longitud`. En la figura 6.18 se puede apreciar la implementación descrita.

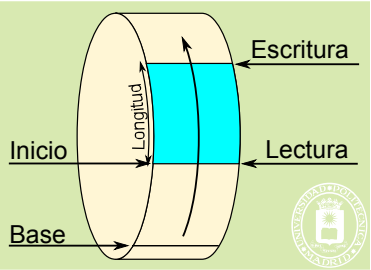


Figura 6.18 *Implementación típica de una tubería mediante un buffer circular.*

Tuberías con nombre

Existe una versión de tuberías con nombre denominada FIFO en el caso de UNIX y las tuberías con nombre en el caso de Windows.

En el caso de los FIFO, el comportamiento es el mismo que el de las tuberías sin nombre, con la salvedad de que pueden ser utilizados por procesos no emparentados.

Sin embargo, las tuberías con nombre de Windows son muy distintas, puesto que tienen las siguientes características: a) sólo utilizan un manejador tanto para leer como para escribir, b) están orientadas a mensajes, c) son bidireccionales, d) puede haber múltiples instancias independientes de la misma tubería y e) se pueden utilizar en sistemas conectados a una red.

6.6.5. *Sockets.* Comunicación remota

El concepto de *socket* y de programación con *sockets* se desarrolló en los 80 como el "Berkeley Sockets Interface" dentro del entorno de UNIX, para la comunicación entre procesos pesados residentes en máquinas remotas. Un socket se define como un punto final (*endpoint*) de comunicaciones. Cada *socket* tiene una dirección mediante la cual se le identifica. Dos procesos se comunican mediante una pareja de *sockets*, un *socket* en cada proceso.

En general, los *sockets* se utilizan en una arquitectura cliente servidor, como se muestra en la figura 6.19. La secuencia de funcionamiento es la siguiente:

- El proceso servidor espera peticiones de los procesos clientes leyendo de su *socket* (*socket* servidor).
- El proceso cliente envía, a través de su *socket* (*socket* cliente) una solicitud a un *socket* servidor.
- El proceso servidor acepta la solicitud y se establece una asociación entre el *socket* servidor y el *socket* cliente. Esta asociación puede ser duradera, de forma que permite el intercambio de una secuencia de mensajes o peticiones entre cliente y servidor, o temporal, de forma que solamente permite el envío de una petición o mensaje y la correspondiente respuesta del servidor.

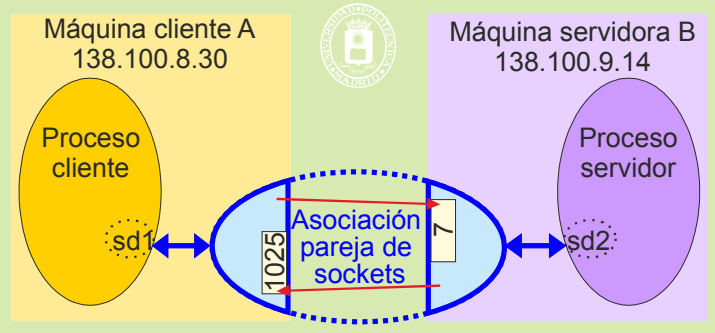


Figura 6.19 *Comunicación cliente servidor mediante una pareja de sockets.*

El "Berkeley Sockets Interface" se ha convertido en un estándar de facto para el desarrollo de aplicaciones de red, estando disponibles, por supuesto en UNIX y Linux, y en gran cantidad de otros sistemas operativos. Por ejemplo, los *sockets* del Windows (WinSock) están basados en la especificación de Berkeley. En este texto seguiremos, por tanto, esta especificación.

Las características más importantes de los *sockets* son las siguientes:

- Es un mecanismo soportado por el sistema operativo para comunicación dentro de un dominio. Veremos un poco más adelante que existen distintos dominios de comunicación totalmente independientes, de forma que un *socket* de un dominio no puede comunicarse con un *socket* de otro dominio.
- Es un mecanismo **con dirección**. El tipo de dirección depende del dominio.
- Se utiliza en forma de asociación de pareja de *sockets* para conseguir comunicación **bidireccional**.
- Los *sockets* tienen ***buffering***, de forma que almacenan mensajes.
- Ofrecen comportamiento **bloqueante** o no bloqueante. El comportamiento por defecto es el bloqueante, pero se puede cambiar a no bloqueante, para ello hay que cambiar un bit de opción, lo que se puede hacer en UNIX con el servicio fcntl().

Dominio del *socket*

El dominio especifica la familia de direcciones en la que se establece el *socket*. Algunos dominios son los siguientes:

- **AF_INET**: Este dominio está basado en direcciones del protocolo de Internet IPV4[4].
- **AF_INEF6**: Este dominio está basado en direcciones del protocolo de Internet IPV6. Junto con el anterior son los más utilizados.

4 Una transmisión IP está caracterizada por los cinco parámetros siguientes:
Protocolo, que puede ser UDP (5% de los casos), TCP (95% de los casos) u otro (uso muy limitado).
Dirección IP del computador origen más número de puerto origen utilizado.
Dirección IP del computador destino más número de puerto destino utilizado.

- **AF_BTH**: Este dominio está basado en la familia de direcciones Bluetooth.
- **AF_UNIX**. Es un dominio basado en nombres del sistema de ficheros. Se utiliza para comunicación entre procesos locales de un sistema UNIX.

Es de destacar que los servicios de los *socket* son independientes del dominio, la única diferencia se encuentra en el tipo de direcciones utilizado.

Tipo de *socket*

Los dos tipos de *sockets* más importantes son el *stream* y el *datagrama*.

El ***socket stream*** (SOCK_STREAM) es similar a una conversación telefónica en la que los participantes van alternando sus mensajes. Tiene las siguientes propiedades:

- Está orientado a flujo de datos.
- Establece una comunicación **con conexión**. La asociación entre el *socket* cliente y el servidor es duradera, de forma que se puede establecer un diálogo cliente-servidor con una secuencia de solicitudes o mensajes con sus respectivas respuestas. La asociación se termina de forma explícita cerrando el *socket* cliente.
- **Es fiable**. Asegura que los mensajes llegan, que su contenido es correcto y que llegan en el orden en que fueron enviados.

El ***socket datagrama*** (SOCK_DGRAM) es similar a la correspondencia postal. Tiene las siguientes propiedades:

- Está orientado a mensajes individuales.
- Es **sin conexión**. El *socket* cliente se asocia al *socket* servidor solamente para el intercambio de un mensaje. Enviada la respuesta por parte del servidor, se rompe la asociación. Si el cliente desea enviar otra solicitud al servidor se ha de establecer una nueva asociación.
- **No es fiable**: puede haber pérdida de mensajes, éstos pueden llegar en desorden e, incluso, pueden llegar duplicados.

Protocolo de comunicaciones que utiliza el *socket*

Los protocolos de comunicaciones que se pueden seleccionar para el *socket* dependen de su dominio y de su tipo. Para los dominios AF_INET y AF_INET6 se dispone, entre otros, de los dos protocolos siguientes:

IPPROTO_TCP, para *sockets* de tipo *stream*.
IPPROTO_UDP, para *sockets* de tipo *datagrama*.

Dirección del socket

Para establecer la dirección de un socket hay que utilizar una estructura de tipo `sockaddr`. Esta estructura depende del dominio del socket. Existe una estructura genérica de dirección, cuya declaración es:

```
struct sockaddr {
   sa_family_t sa_family;
   char sa_data[14];
};
```

Dominio AF_INET

Para el dominio AF_INET (protocolo IPv4) la estructura `sockaddr` se declara como:

```
struct sockaddr_in {
   short           sin_family;   // e.g. AF_INET.
   unsigned short  sin_port;     // e.g. htons(3490).
   struct in_addr  sin_addr;     // la estructura struct in_addr, se declara más abajo.
   char            sin_zero[8];  // no se usa, poner a cero.
};

struct in_addr {
   unsigned long   s_addr;  // convertir la dirección con with inet_aton()
};
```

Como los usuarios conocen la dirección en formato de texto, por ejemplo: "138.100.8.100" o "laurel.datsi.fi.upm.es", hay que convertir dicha dirección al formato de red. Para ello, se pueden utilizar las dos funciones siguientes:

- Para convertir de formato decimal-punto (e.g. 138.100.8.100) se puede utilizar la función:

```
int inet_aton (char *str, struct in_addr *dir);   // La función devuelve la dirección en formato red.
```

- Para convertir de formato nombre-punto (e.g. "laurel.datsi.fi.upm.es") se puede utilizar la función:

```
struct hostent *gethostbyname (char *str); // La estructura devuelta contiene la dirección en formato red.
```

Dominio AF_INET6

Para el protocolo AF_INET6 (IPv6) la estructura sockaddr se declara como:

```
struct sockaddr_in6 {
```

```
    sa_family_t     sin6_family;      /* AF_INET6 */
    in_port_t       sin6_port;        /* port number */
    uint32_t        sin6_flowinfo;    /* IPv6 flow information */
    struct in6_addr sin6_addr;        /* IPv6 address */
    uint32_t        sin6_scope_id;    /* Scope ID (new in 2.4) */
};

struct in6_addr {
    unsigned char   s6_addr[16];      /* IPv6 address */
};
```

Para convertir de los formatos decimal-punto y nombre-punto se utilizan las mismas funciones que para el caso AF_INET, pero utilizando como argumento una estructura in6_addr.

6.7. MECANISMOS DE SINCRONIZACIÓN

En los problemas de sincronización, un proceso debe esperar la ocurrencia de un determinado evento o condición. Así, por ejemplo, en problemas de tipo productor-consumidor el proceso consumidor debe esperar mientras no haya datos que consumir. Para que los procesos puedan sincronizarse es necesario disponer de servicios que permitan bloquear o suspender, bajo determinadas circunstancias, la ejecución de un proceso. Los principales mecanismos específicos de sincronización que ofrecen los sistemas operativos son:

- Señales.
- Semáforos.
- *Mutex* y variables condicionales (específico para *threads*).
- Cerrojos en ficheros.

6.7.1. Sincronización mediante señales

Las señales, descritas en el capítulo "3 Procesos" , pueden utilizarse para sincronizar procesos. Si, por ejemplo, se utilizan señales UNIX, un proceso puede bloquearse en el servicio `pause` esperando la recepción de una señal. Esta señal puede ser enviada por otro proceso mediante el servicio `kill`. Con este mecanismo se consigue que unos procesos esperen a que se cumpla una determinada condición y que otros los despierten cuando se cumple la condición que les permite continuar su ejecución.

El empleo de señales, sin embargo, no es un mecanismo muy apropiado para sincronizar procesos, debido a las siguientes razones:

- Las señales tienen un comportamiento asíncrono. Un proceso puede recibir una señal en cualquier punto de su ejecución, aunque no esté esperando su recepción.
- Las señales no se encolan, salvo las de real-time que sí se encolan. Si hay una señal pendiente de entrega a un proceso y se recibe una señal del mismo tipo, la primera se perderá. Sólo se entregará la última. Esto hace que se puedan perder eventos de sincronización importantes.

6.7.2. Semáforos

Un semáforo [Dijkstra 1965] es un mecanismo de sincronización que se utiliza generalmente en sistemas con memoria compartida, bien sea un monoprocesador o un multiprocesador. Su uso en un multicomputador depende del sistema operativo en particular. El semáforo es un mecanismo para sincronizar procesos pesados, aunque también pude usarse para *threads*.

Un semáforo tiene las siguientes propiedades:

- Contiene un campo entero **contador**, al que se le puede asignar un **valor inicial no negativo**. Dicho contador se utiliza como una de las **variables de control** para resolver el problema de concurrencia. El significado del valor del contador es el siguiente:
  - Si es mayor que 0 significa el número de boletos de entrada que el semáforo tiene disponibles.
  - Si es negativo significa el número de procesos que están esperando en el semáforo a que alguien genere un boleto.
  - Si es cero significa que no tiene boletos pero que nadie espera.
- Existen dos **operaciones atómicas**: **`sem_wait`** (también llamada `down` o `p`) y **`sem_post`** (también llamada `up`, `siganal` o `v`). Las definiciones de estas dos operaciones son las siguientes:
  - **`sem_wait`**. Este servicio se utiliza para solicitar un boleto de entrada del semáforo. Si el semáforo tiene boletos disponible (el contador tiene un valor mayor que 0) le da uno y le deja seguir ejecutando. Si no hay boletos disponibles, el semáforo bloquea al proceso, hasta que alguien produzca un bo-

leto. Para realizar esta función, el semáforo decrementa el valor de su contador y actúa según sea el valor obtenido de acuerdo a la secuencia siguiente:

```
sem_wait(s) {
    <<decrementar contador s>>;
    if (<<contador>> < 0)
        <<esperar en el semáforo>>
}
```

Atómico

- **`sem_post`**. Este servicio se utiliza para crear un boleto de entrada del semáforo. Si no hay procesos esperando, simplemente se incrementa el número de boletos disponibles. Sin embargo, si hay procesos esperando, el semáforo selecciona uno, al que le da el boleto para que siga ejecutando. Para realizar esta función, el semáforo incrementa el valor de su contador y actúa según sea el valor obtenido de acuerdo a la secuencia siguiente:

```
sem_post(s) {
    <<incrementar contador>>;
    if (<<contador>> <= 0)
    <<que continúe uno>>
}
```

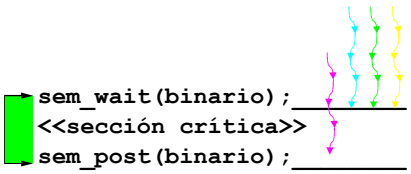
Atómico

El semáforo es un recurso **consumible** y **sin posesión**. Los boletos del semáforo los produce cualquier proceso que ejecute el servicio `sem_post`, sin tener que haber hecho previamente un `sem_wait`, y el boleto que recibe el proceso se consume al seguir ejecutando.

Semáforo binario y sección crítica con semáforos

El **semáforo binario** es un semáforo cuyo máximo valor de contador es 1. Es, por tanto, un semáforo que solamente deja pasar a un proceso.

La construcción de una sección crítica con un semáforo binario es muy simple, como se muestra a continuación:



El valor que tiene que tomar el semáforo inicialmente es 1, de esta forma, sólo se permite que un único proceso acceda a la sección crítica. Si el valor inicial del semáforo fuera, por ejemplo, 2, entonces dos procesos podrían ejecutar la llamada `sem_wait` sin bloquearse y, por tanto, se permitiría que ambos se ejecutaran de forma simultánea dentro de la sección crítica. Además, los procesos solamente deben ejecutar el `sem_post` al salir de la sección crítica.

En la figura 6.20 se representa la ejecución de tres procesos (P_0, P_1 y P_2) que intentan acceder a una sección crítica mediante un semáforo binario.

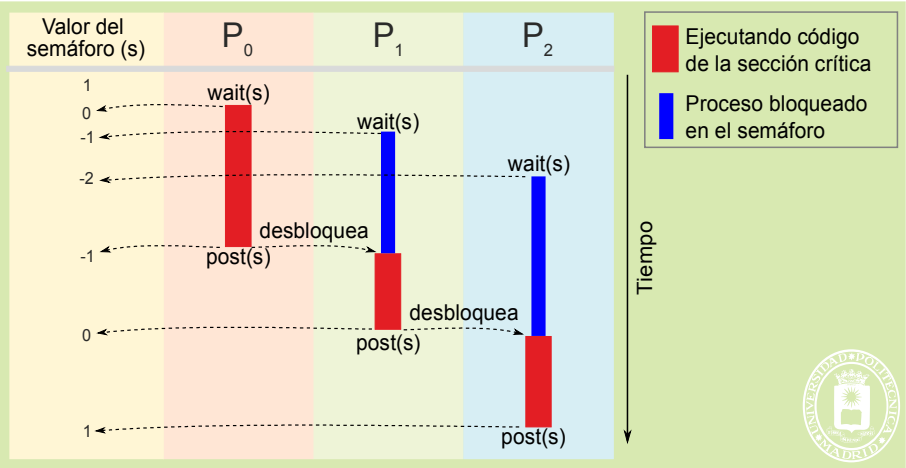


Figura 6.20 *Sección crítica con un semáforo binario.*

6.7.3. *Mutex* y variables condicionales

Los *mutex* y las variables condicionales son mecanismos especialmente concebidos para la sincronización de *threads*.

El *mutex* está expresamente diseñado para resolver la exclusión mutua mientras que las variables condicionales están expresamente diseñadas para resolver la sincronización condicional.

Mutex

Un *mutex* es el mecanismo de sincronización de *threads* más sencillo y eficiente. Se emplea para obtener asegurar la exclusión mutua sobre secciones críticas.

El comportamiento del *mutex* es similar al de un semáforo binario pero con las siguientes diferencias.

- El *mutex* es como una llave que se posee para entrar, y que hay que devolver para que otro *thread* pueda utilizarla. Por tanto, es un **recurso reutilizable** (en contraposición con el semáforo que es consumible), y **sin memoria**, puesto que carece del contador que caracteriza al semáforo.
- Es **con posesión.** Solamente puede devolver la llave el que la obtuvo anteriormente.

Sobre un *mutex* se pueden realizar dos operaciones **atómicas** básicas:

- **`lock`**: El *thread* que ejecuta esta operación intenta obtener el *mutex* y seguir ejecutando. Si el *mutex* lo tiene otro *thread*, el *thread* que realiza la operación espera a que alguien devuelva el *mutex*. En caso contrario obtiene el *mutex* y sigue ejecutando. Esta operación se realiza de acuerdo a la secuencia siguiente:

```
mutex_lock(m) {
    if (<<no hay llave>>)
        <<esperar llave>>;
    <<abrir, entrar, cerrar y llevármela>>;
}
```

Atómico

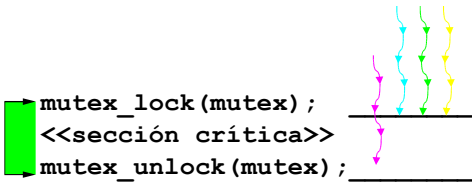
- **`unlock`**: Esta operación la ejecuta el *thread* para devolver el *mutex*. Si existen *threads* esperando por él, se escogerá uno de ellos, que será el nuevo proceso que adquiera el *mutex*. La operación `unlock` sobre un *mutex* sólo la puede ejecutar el *thread* que adquirió con anterioridad el *mutex* mediante la operación `lock`. Esto es diferente a lo que ocurre con las operaciones `sem_wait` y `sem_post` sobre un semáforo. Esta operación se realiza de acuerdo a la secuencia siguiente:

```
mutex_unlock(m) {
    if (<<alguien espera>>)
        <<entregar llave>>;
    else
        <<devolver llave a cerradura>>;
}
```

Atómico

Sección crítica con *mutex*

El siguiente segmento de pseudocódigo utiliza un *mutex* para proteger una sección crítica.



En la figura 6.21 se representa de forma gráfica una situación en la que dos *threads* intentan acceder de forma simultánea a ejecutar código de una sección crítica utilizando un *mutex* para protegerla.

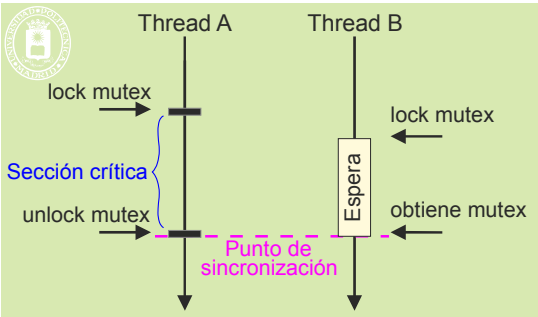


Figura 6.21 *Ejemplo de* mutex *en una sección crítica.*

Dado que las operaciones `lock` y `unlock` son atómicas, sólo un *thread* conseguirá bloquear el *mutex* y podrá continuar su ejecución dentro de la sección crítica. El segundo *thread* esperará hasta que el primero libere el *mutex* mediante la operación `unlock`.

Variable condicional

Una variable condicional es una variable de sincronización que está asociada a un *mutex* y que se utiliza para bloquear a un *thread* hasta que ocurra algún suceso. Los eventos de Windows son similares a las variables condicionales, sin embargo, los eventos no se encuentran asociados a ningún *mutex*, como ocurre con las variables condicionales.
Las variables condicionales tienen tres operaciones para esperar y señalizar:

- **`c_wait(c,m)`**: esta operación solamente la puede ejecutar un *thread* que esté en posesión del *mutex* `m`. El *thread* queda bloqueado a la espera de que llegue el aviso de la variable condicional `c` y se libera el *mutex*, de forma que algún otro *thread* lo pueda adquirir. El bloqueo del *thread* y la liberación del *mutex* se realizan de forma atómica.
  Es de destacar que el valor de la variable condicional `c` puede haber cambiado de valor desde que se recibe su aviso hasta que se consigue nuevamente el *mutex*.
  Esta operación se realiza de acuerdo a la secuencia siguiente:

```
c_wait(c,m) {
   mutex_unlock(m);          } Atómico
   <<esperar aviso>>;
   mutex_lock(m);            } Atómico
}
```

- **`c_signal(c)`**: envía un aviso a un *thread* bloqueado en la variable condicional `c`. El *thread* que se despierta, completa la operación c_wait, por lo que compite de nuevo por el *mutex*. Si no existe ningún *thread* esperando el aviso se pierde. Esta operación se realiza de acuerdo a la secuencia siguiente:

```
c_signal(c) {
   if (<<alguien espera>>)
      <<avisar de que siga>>;   } Atómico
   else (<<se pierde>>)
}
```

- **`c_broadcast(c)`**: envía un aviso a todos los *threads* bloqueados en la variable condicional `c`. Como se ha comentado en el caso anterior, cada *thread* que se despierta, completa la operación c_wait, por lo que compite de nuevo por el *mutex*. Esta operación se realiza de acuerdo a la secuencia siguiente:

```
c_broadcast(c) {
   while(<<alguien espera>>)
      <<avisar que siga>>;   } Atómico
}
```

Construcción del código para una sincronización condicional

Bloque de enrada

Mediante una combinación de un *mutex* y una variable de condición se puede construir el código para una sincronización condicional (lo que sería el código de la sección crítica de entrada del esquema de sincronización condicional de la figura 6.9).

```
mutex_lock(mimutex);
while (<<no puedo continuar (condición variable control)>>)
      c_wait(cond, mimutex);
<<modifica variables de control>>
mutex_unlock(mimutex);
```

Es fundamental resaltar la necesidad del **while** en el código anterior. En el siguiente código se ha sustituido el `c_wait` por su definición, para apreciar dicha necesidad. Se puede observar que el *thread* espera el aviso fuera de una sección crítica y, cuando recibe el aviso, lo que hace es capturar el *mutex* con un `mutex_lock`. Como puede haber otros *threads* compitiendo por conseguir el *mutex*, puede producirse el hecho de que otro *thread* consiga el *mutex* antes y que dicho *thread* modifique las variables de control, anulando la condición de continuar. El `while` obliga a recalcular la condición de continuar, lo que se hace en exclusión mutua al estar en posesión del *mutex*.

```
mutex_lock(mimutex);
while (<<no puedo continuar (condición variable control)>>)
      c_wait(cond, mimutex) {
         mutex_unlock(mimutex);   } Atómico
         <<esperar aviso>>;
         mutex_lock(mimutex);     } Atómico
      }
<<modifica variables de control>>
mutex_unlock(mimutex);
```

Bloque de salida

El código para el bloque de salida incluye la función de despertar a un posible *thread* que esté esperando en el correspondiente `c_wait`. Por tanto queda como sigue:

```
mutex_lock(mimutex);
<<modifica variables de control>>
c_signal(cond);
mutex_unlock(mimutex);
```

El empleo de *mutex* y variables condicionales que se ha presentado es similar al concepto de monitor [Hoare 1974], y en concreto a la definición de monitor dada para el lenguaje Mesa [Lampson 1980].

En la figura 6.22 se representa de forma gráfica el uso de *mutex* y variables condicionales entre dos *threads* tal y como se ha descrito anteriormente.

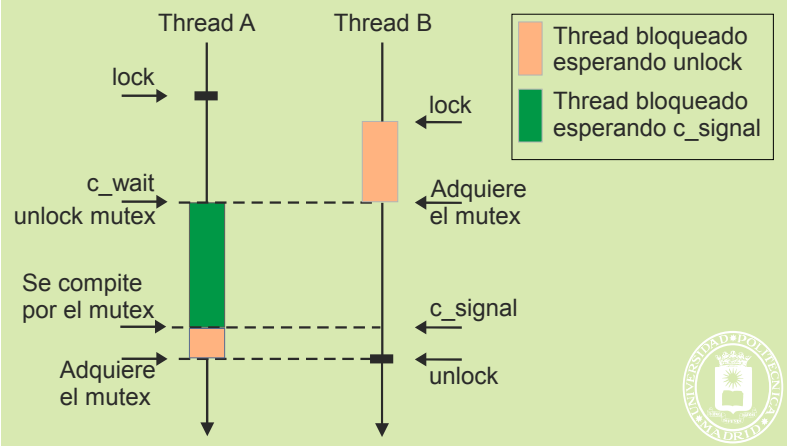


Figura 6.22 *Ejemplo de* mutex *y variables condicionales.*

La combinación de los dos bloques descritos anteriormente permite plantear la siguiente solución general de sincronización en la que el acceso al recurso compartido puede ser múltiple.

```
mutex_lock(mimutex);
while (<<no puedo continuar (condición variable control)>>)
      c_wait(cond, mimutex);
<<modifica variables de control>>
mutex_unlock(mimutex);

<<acceso al recurso compartido. El acceso puede ser múltiple>>

mutex_lock(mimutex);
<<modifica variables de control>>
c_signal(cond);
mutex_unlock(mimutex);
```

6.7.4. Cerrojos sobre ficheros

Los servicios del sistema operativo para gestionar cerrojos sobre ficheros permiten coordinar la coutilización de ficheros por parte de procesos cooperantes independientes. Los cerrojos se establecen sobre la totalidad o parte de un fichero, dependiendo de la granularidad del control que se desee utilizar. Mientras menor sea el grano del cerrojo, es decir, cuanto menores sean las regiones del fichero sobre las que se establezcan los cerrojos, menor será la contención que se producirá sobre los procesos cooperantes.

Existen dos modalidades de cerrojos:

- Cerrojo **consultivo** (*advisory*). Estos cerrojos permiten a los procesos que cooperan sincronizarse, pero no afectan a los procesos que los usan. El sistema operativo no prohíbe las operaciones de lectura y escritura sobre el fichero aunque no se tenga posesión de un cerrojo sobre la zona afectada.
- Cerrojo **obligatorio** (mandatory). Estos cerrojos afectan a todos los procesos, puesto que el sistema operativo prohíbe las lecturas y escrituras si existe un cerrojo, de acuerdo al tipo de cerrojo, según se explica a continuación. Su uso es peligroso, puesto que puede dejar un fichero bloqueado por un cerrojo.

Existen los dos tipos de cerrojos siguientes:

- **Compartido**. Las características del cerrojo compartido son las siguientes:
  - No puede solapar con otro exclusivo.
  - Sólo se puede (cerrojo obligatorio) o se debe (cerrojo consultivo) hacer `read` de la región que tiene el cerrojo.

- **Exclusivo**: Las características del cerrojo exclusivo son las siguientes:
  - No puede solapar con ningún otro, ya sea compartido o exclusivo.
  - Se puede hacer `read` y `write`.

La tabla 6.1 muestra las alternativas de concesión de cerrojos.

Tabla 6.1 *Concesión de cerrojos*

	¿Se puede conceder si se solapa con otro de tipo exclusivo?	¿Se puede conceder si se solapa con otro de tipo compartido?	¿Se puede conceder si no hay ninguno que solape?
Cerrojo pedido de tipo exclusivo	NO	NO	SI
Cerrojo pedido de tipo compartido	NO	SI	SI

Para especificar la zona del fichero sobre la que se establece el cerrojo hay que especificar el inicio, es decir, el número del primer byte de la zona, así como su tamaño.

Linux soporta cerrojos consultivos sobre regiones (servicio `fcntl`). Aunque también soporta cerrojos obligatorios sobre regiones no son recomendables. Además el servicio `flock` permite establecer un cerrojo sobre todo un fichero. En Windows hay que diferenciar entre los cerrojos de coutilización que se establecen sobre todo el fichero en el momento de hacer su apertura de los cerrojos de byte que son cerrojos consultivos sobre regiones (servicios `LockFile`, `LockFileEx`, `UnlockFile` y `UnlockFileEx`).

6.7.5. Paso de mensajes

Todos los mecanismos vistos hasta el momento necesitan que los procesos que quieren intervenir en la comunicación o quieren sincronizarse se ejecuten en la misma máquina. Cuando se quiere comunicar y sincronizar procesos que ejecutan en máquinas distintas es necesario recurrir al *paso de mensajes*. En este tipo de comunicación los procesos intercambian *mensajes* entre ellos a través de un *enlace de comunicación*. Es obvio que este esquema también puede emplearse para comunicar y sincronizar procesos que se ejecutan en la misma máquina, en cuyo caso los mensajes son locales a la máquina donde ejecutan los procesos.

Los procesos se comunican mediante dos operaciones básicas:

- **`send`**`(dirección destino, mensaje)`, envía un mensaje al proceso destino.
- **`receive`**`(dirección origen, mensaje)`, recibe un mensaje del proceso origen.

De acuerdo con estas dos operaciones, las tuberías se pueden considerar en cierta medida como un mecanismo de comunicación basado en paso de mensajes. Los procesos pueden enviar un mensaje a otro proceso por medio de una operación de escritura y puede recibir mensajes de otros mediante una operación de lectura. En este caso el enlace que se utiliza para comunicar a los procesos es la propia tubería.

Existen múltiples implementaciones de sistemas con paso de mensajes. Los *sockets* que se describen más adelante constituyen uno de los modelos más utilizados en aplicaciones distribuidas. A continuación se describen algunos aspectos de diseño relativos a este tipo de sistemas.

Tamaño del mensaje

Los mensajes que envía un proceso a otro pueden ser de tamaño fijo o variable. En caso de mensajes de longitud fija la implementación es más sencilla. Sin embargo, si el tamaño es muy grande es ineficiente, puesto que se desaprovecha ancho de banda y espacio de almacenamiento, y si es pequeño dificulta la tarea del programador ya que puede obligar a éste a descomponer los mensajes grandes en mensajes de longitud fija más pequeños.

Comunicación directa e indirecta

La comunicación es **directa** cuando cada proceso que desea enviar o recibir un mensaje de otro debe nombrar de forma explícita al proceso receptor o emisor del mensaje. En este esquema de comunicación, las operaciones básicas `send` y `receive` se definen de la siguiente manera:

- **`send`**`(P, mensaje)`, envía un mensaje al proceso P.
- **`receive`**`(Q, mensaje)`, espera la recepción de un mensaje por parte del proceso Q.

Existen modalidades de paso de mensajes con comunicación directa que permiten especificar al receptor la posibilidad de recibir un mensaje de cualquier proceso. En este caso la operación `receive` se define de la siguiente forma:

receive(ANY, mensaje);

La comunicación es **indirecta** cuando los mensajes no se envían directamente del emisor al receptor, sino a unas estructuras de datos que se denominan *colas de mensajes* o *puertos*. Una cola de mensajes es una estructura a

la que los procesos pueden enviar mensajes y de la que se pueden extraer mensajes. Cuando dos procesos quieren comunicarse entre ellos, el emisor sitúa el mensaje en la cola y el receptor lo extrae de ella. Sobre una cola de mensajes puede haber múltiples emisores y receptores. La figura 6.23 presenta la forma de comunicación utilizado colas de mensajes.

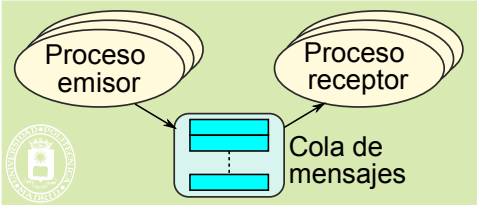


Figura 6.23 *Uso de una cola de mensajes en la comunicación entre procesos.*

Un puerto es una estructura similar a una cola de mensajes. Sin embargo, un puerto se encuentra asociado a un proceso, siendo éste el único que puede recibir datos de él. En este caso, cuando dos procesos quieren comunicarse entre sí, el receptor crea un puerto y el emisor envía mensajes al puerto del receptor. La figura 6.24 presenta la forma de comunicación utilizado puertos.

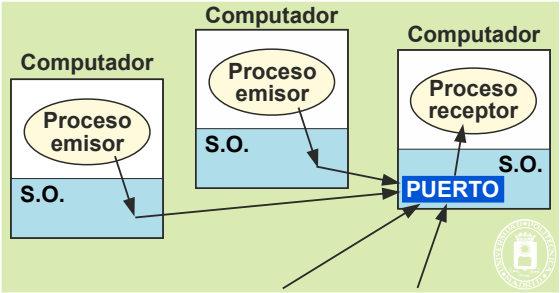


Figura 6.24 *Utilización de un puerto para comunicación entre procesos.*

Utilizando comunicación indirecta las operaciones `send` y `receive` toman la siguiente forma:

- **`send`**`(Q, mensaje)`, envía un mensaje a la cola o al puerto Q.
- **`receive`**`(Q, mensaje)`, recibe un mensaje de la cola o del puerto Q.

Cualquiera que sea el método utilizado, el paso de mensajes siempre se realiza en exclusión mutua. Si dos procesos ejecutan de forma simultánea una operación `send`, los mensajes no se entrelazan, primero se envía uno y a continuación el otro. De igual forma, si dos procesos desean recibir un mensaje de una cola, sólo se entregará el mensaje a uno de ellos.

Sincronización

La comunicación entre dos procesos es síncrona cuando los dos procesos han de ejecutar los servicios de comunicación al mismo tiempo, es decir, el emisor debe estar ejecutando la operación `send` y el receptor ha de estar ejecutando la operación `receive`. La comunicación es asíncrona en caso contrario.

En general, son tres las combinaciones más habituales que implementan los distintos tipos de paso de mensajes.

- **Envío y recepción bloqueante**. En este caso tanto el emisor como el receptor se bloquean hasta que tenga lugar la entrega del mensaje. Esta es una técnica de paso de mensajes totalmente síncrona que se conoce como *cita*. El primero que llega a la cita, ya sea el emisor o el receptor, ha de esperar hasta que llegue el otro.
- **Envío no bloqueante y recepción bloqueante**. Esta es la combinación generalmente más utilizada. El emisor no se bloquea y por lo tanto puede continuar su ejecución depositando el mensaje en el mecanismo de comunicación, sin embargo, el receptor se bloquea hasta que le llega.
- **Envío y recepción no bloqueante**. Se corresponde con una comunicación totalmente asíncrona en la que nadie espera. En este tipo de comunicación es necesario disponer de servicios que permitan al receptor saber si se ha recibido un mensaje.

Una operación de recepción bloqueante permite bloquear el proceso que la ejecuta hasta la recepción del mensaje. Esta característica permite emplear los mecanismos de paso de mensajes para sincronizar procesos, ya que una sincronización siempre implica un bloqueo.

Almacenamiento

Este aspecto hace referencia a la capacidad del enlace de comunicaciones. El enlace, y por tanto el paso de mensajes puede:

- **No tener capacidad** (sin almacenamiento) para almacenar mensajes. En este caso, el mecanismo utilizado como enlace de comunicación no puede almacenar ningún mensaje y por lo tanto la comunicación entre los procesos emisor y receptor debe ser síncrona, es decir, el emisor sólo puede continuar cuando el receptor haya recibido el mensaje.

- **Tener capacidad** (con almacenamiento) para almacenar mensajes. En este caso, la cola de mensajes o el puerto al que se envían los mensajes pueden tener un cierto tamaño para almacenar mensajes a la espera de su recepción. Si la cola no está llena al enviar un mensaje, se guarda en ella y el emisor puede continuar su ejecución sin necesidad de esperar. Sin embargo, si la cola ya está llena, el emisor deberá bloquearse hasta que haya espacio disponible en la cola para insertar el mensaje.

A continuación se describen algunas situaciones en las que se puede utilizar el mecanismo de paso de mensajes.

Ámbito de uso

Existen diferentes mecanismos basados en paso de mensajes. Algunos de ellos sólo pueden utilizarse para comunicar procesos que ejecutan en el mismo computador. Este es el caso de las colas de mensajes UNIX que se describirán más adelante. Otros, sin embargo, permiten comunicar procesos que ejecutan en computadores distintos. Así, por ejemplo, los *sockets*, que veremos en la sección "6.6.5 Sockets. Comunicación remota", constituyen un mecanismo de paso de mensajes que se puede utilizar para comunicar procesos que se ejecutan en computadores distintos.

Secciones críticas con paso de mensajes

Para resolver el problema de la sección crítica utilizando paso de mensajes se va a recurrir al empleo de colas de mensajes con una solución similar a la utilizada con las tuberías. Se creará una cola de mensajes con capacidad para almacenar un único mensaje que hará las funciones de testigo. Cuando un proceso quiere acceder al código de la sección crítica ejecutará la función `receive` para extraer el mensaje que hace de testigo de la cola. Si la cola está vacía, el proceso se bloquea ya que en este caso el testigo lo posee otro proceso.

Cuando el proceso finaliza la ejecución del código de la sección crítica inserta de nuevo el mensaje en la cola mediante la operación `send`.

Inicialmente alguno de los procesos que van a ejecutar el código de la sección crítica deberá crear la cola e insertar el testigo inicial ejecutando algún fragmento con la siguiente estructura:

```
< Crear la cola de mensajes >
send(cola, testigo);     /* insertar en la cola el testigo */
```

Una vez que todos los procesos tienen acceso a la cola, sincronizan su acceso a la sección crítica ejecutando el siguiente fragmento de código:

```
receive(cola, testigo);
< Código de la sección crítica >
send(cola, testigo);
```

De esta forma el primer proceso que ejecuta la operación `receive` extrae el mensaje de la cola y la vacía, de tal manera que el resto de procesos se bloqueará hasta que de nuevo vuelva a haber un mensaje disponible en la cola. Este mensaje lo inserta el proceso que lo extrajo mediante la operación `send`. De esta forma alguno de los procesos bloqueados se despertará y volverá a extraer el mensaje de la cola vaciándola.

Productor-consumidor con colas de mensajes

A continuación se presenta una posible solución al problema del productor-consumidor utilizando paso de mensajes. Esta solución vuelve a ser similar a la que se presentó en la sección empleando tuberías.

Cuando el proceso productor produce un elemento, lo envía al proceso consumidor mediante la operación `send`. Lo ideal es que esta operación sea no bloqueante para que pueda seguir produciendo elementos. En el caso de que no haya espacio suficiente para almacenar el mensaje el productor se bloqueará.

Cuando el proceso consumidor desea procesar un nuevo elemento ejecuta la operación `receive`. Si no hay datos disponibles el proceso se bloquea hasta que el productor cree algún nuevo elemento.

La estructura de los procesos productor y consumidor se muestra en el programa 6.1.

Programa 6.1 Procesos productor-consumidor utilizando paso de mensajes.

```
Productor() {

    for(;;) {
        < Producir un dato >
        send(Consumidor, dato);
    }
}

Consumidor() {

    for(;;) {
        receive(Productor, dato);
        < Consumir el dato >
    }
```

```
}
```

Recursos limitados con paso de mensajes

En esta sección se va a resolver el problema de los filósofos comensales utilizando un modelo de paso de mensajes basado en colas de mensajes. Se supondrá que la operación `receive` sobre una cola de mensajes vacía bloquea al proceso que la ejecuta hasta que la cola vuelve a tener algún mensaje. En este caso se retira el mensaje y el proceso continúa la ejecución. La solución que se va a mostrar a continuación se basa en la solución propuesta en el programa 6.6 que resolvía el problema utilizando semáforos. En este caso se va a modelar cada palillo y la mesa con una cola de mensajes. Para asegurar el funcionamiento de la solución es importante que cada cola de mensaje almacene inicialmente un mensaje y la cola de mensajes que modela la mesa debe almacenar inicialmente cuatro mensajes. El contenido de estos mensajes no es importante para la resolución del problema. Para asegurar esto se puede recurrir a un proceso controlador que se encargue de crear las colas e insertar los mensajes correspondientes en ellas. Una vez hecho esto, puede comenzar la ejecución de los procesos que modelan a los filósofos. La estructura del proceso controlador y de cada uno de los filósofos se muestra en el programa 6.2. Como se puede ver en este programa se utiliza un vector de colas para representar a cada una de las colas que modelan cada palillo.

Programa 6.2 Problema de los filósofos comensales utilizando colas de mensajes.

```
#define NUM_FILOSOFOS    6
Controlador(){
    Crear la cola con nombre mesa;
    for (i = 0; i < NUM_FILOSOFOS -1; i++)
            send(mesa, m);

    for(i = 0; i < NUM_FILOSOFOS; i++){
        crear la cola palillo[i];
        send(palillo[i], m);
    }
}

Filósofo(k){
    for(;;){
        receive(mesa, m);
        receive(palillo[k], m);
        receive(palillo[k+1] % NUM_FILOSOFOS, m);

        come();

        send(palillo[k], m);
        send(palillo[k+1] % NUM_FILOSOFOS, m);
        send(mesa, m);

        piensa();
    }
}
```

Paso de mensajes en el modelo cliente-servidor

El empleo más típico del paso de mensajes se encuentra en los esquemas cliente-servidor. En este tipo de situaciones el proceso servidor se encuentra en un bucle infinito esperando la recepción de las peticiones de los clientes (operación `receive`). Los clientes solicitan un determinado servicio enviando un mensaje al servidor (operación `send`).

Cuando el servidor recibe el mensaje de un cliente lo procesa, sirve la petición y devuelve el resultado mediante una operación `send`. Según se sirva la petición, los servidores se pueden clasificar de la siguiente manera:

- **Servidores secuenciales**. En este caso es el propio servidor el que se encarga de satisfacer la petición del cliente y devolverle los resultados. Con este tipo de servidores sólo se puede atender a un cliente de forma simultánea.
- **Servidores concurrentes**. En este tipo de servidores, cuando llega una petición de un cliente, el servidor crea un proceso hijo o *thread* que se encarga de servir al cliente y devolverle los resultados. Con esta estructura mientras un proceso hijo está atendiendo a un cliente, el proceso servidor puede seguir esperando la recepción de nuevas peticiones por parte de otros clientes. De esta forma se consigue atender a más de un cliente de forma simultánea.

En la figura 6.25 se representa de forma gráfica el funcionamiento de ambos tipos de servidores.

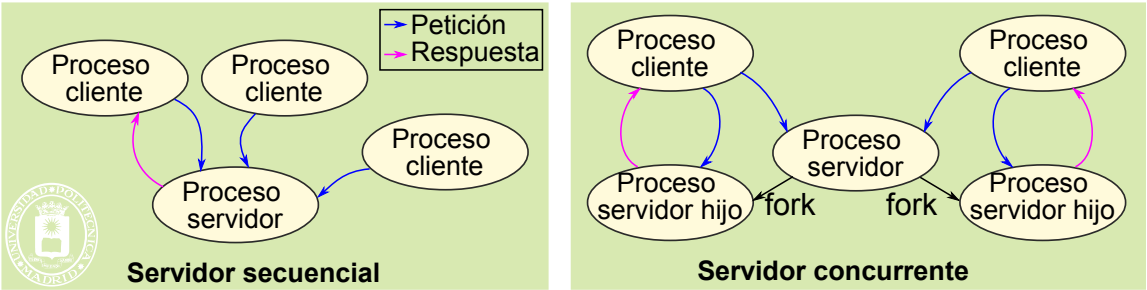


Figura 6.25 *Estructura de un servidor secuencial y concurrente.*

6.7.6. Empleo más adecuado de los mecanismos de comunicación y sincronización

A modo de resumen, en esta sección se van a indicar los mecanismos más adecuados para comunicar y sincronizar procesos que se ejecutan en diferentes escenarios:

- *threads*. Para comunicar datos entre diferentes *threads* de un mismo proceso el mecanismo más habitual es el empleo de memoria compartida. Para la sincronización se recurre a *mutex* y variables condicionales.
- Procesos emparentados. Para aquellos procesos emparentados, como los procesos que en UNIX se crean a partir de la llamada `fork`, el mecanismo más habitual para sincronizarlos es el empleo de semáforos sin nombre. Para la comunicación entre ellos los mecanismos más adecuados son las tuberías y la memoria compartida.
- Procesos no emparentados que se ejecutan en el mismo computador. En este caso la sincronización puede realizarse mediante semáforos con nombre y la comunicación mediante tuberías con nombre o mecanismos de paso de mensajes locales.
- Procesos que se ejecutan en diferentes computadores. Lo más habitual es el empleo de los *sockets*.

Además de los esquemas descritos anteriormente, el fichero es un mecanismo que se puede utilizar en todos los escenarios anteriores para comunicar datos entre diferentes procesos. En un escenario en el que los procesos ejecutan en máquinas distintas, siempre es posible transferir un fichero de un computador a otro para comunicar datos.

La figura 6.26 presenta en el espacio nivel/acoplamiento, los distintos mecanismos de almacenamiento, aviso, sincronización y comunicación, indicando su uso y tipo de proceso para los que son adecuados.

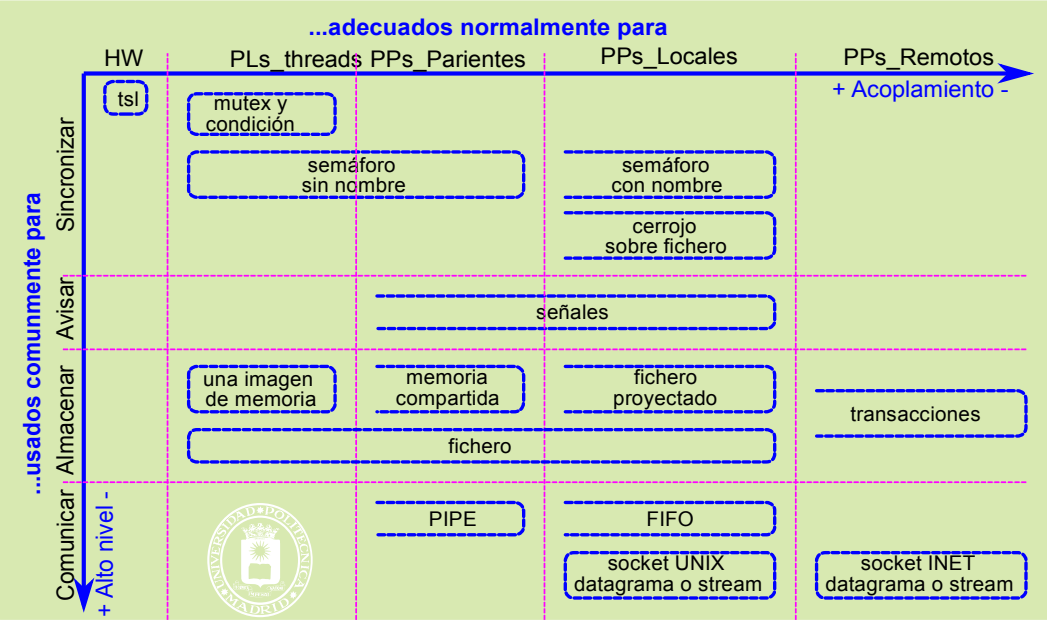


Figura 6.26 *Utilización de los mecanismos de almacenamiento, aviso, sincronización y comunicación.*

6.8. TRANSACCIONES

El concepto de transacción en un sistema informático depende del contexto en el que se emplee, por ejemplo:

- Para un usuario una transacción es una operación de tipo solicitud-respuesta, como puede ser una operación de compra de un activo, una operación de transferencia de fondos o una operación en un cajero.

- Para el auditor del sistema informático es la ejecución de la operación. Para ello se deberán generar y mantener los correspondientes registros históricos de las transacciones ejecutadas en el sistema informático.
- Para el programador de la aplicación es un programa con unas características especiales que analizaremos, más adelante. En el resto de la sección nos centraremos en este concepto de transacción.

La necesidad de las transacciones surge porque los mecanismos de comunicación y sincronización vistos hasta el momento, no ofrecen dos características que son fundamentales para muchas aplicaciones, puesto que:

- Ni ofrecen **recuperación** de fallos en el sistema informático.
- Ni garantizan el que los datos o recursos que se comparten siempre queden en un **estado consistente**.

Para ilustrar esto, considere el clásico ejemplo de una aplicación bancaria que utiliza un fichero para almacenar los datos de las cuentas de los clientes. Considere que sobre este fichero se realizan de forma concurrente dos operaciones. La primera realiza una transferencia de 100 euros de la cuenta A a la B y la segunda realiza una transferencia de 50 euros de la cuenta B a la C. Para realizar estas dos transferencias se realizan las siguientes operaciones sobre el fichero:

Operación 1: Transferir (100, A, B)
Leer el saldo de A: saldoA
Leer el saldo de B: saldoB
saldoA = saldoA – 100
saldoB = saldoB + 100
escribir el nuevo saldo (saldoA) en A
escribir el nuevo saldo (saldoB) en B

Operación 2: Transferir(50 , B, C)
Leer el saldo de B: saldoB
Leer el saldo de C: saldoC
saldoB = saldoB – 50
saldoC = saldoC + 50
escribir el nuevo saldo (saldoB) en B
escribir el nuevo saldo (saldoC) en C

Para que ambas operaciones se realicen de forma correcta es necesario, en primer lugar, que se realicen de forma atómica, es decir, de forma indivisible. Si no fuera así, el estado del fichero que almacena las cuentas podría quedar en un estado inconsistente. Esta atomicidad podría resolverse, por ejemplo, mediante el empleo de semáforos. Sin embargo, el empleo de un semáforo no resuelve otro de los problemas que puede aparecer en este escenario. Imagine que la operación 1 se está realizando de forma atómica y que el proceso que realiza la operación falla después de realizar la primera operación de escritura sobre la cuenta A y antes de actualizar el nuevo saldo en la cuenta B. Si esto fuera así, el fichero quedaría en un estado inconsistente puesto que se habría retirado una cantidad de dinero que no se ha ingresado en ningún sitio. Este problema hace que sea necesario otro tipo de mecanismo que asegure la atomicidad y la actualización consistente de los datos o recursos que se comparten. Las transacciones permiten resolver estos problemas.

En este contexto, una **transacción** es una secuencia de operaciones que se realizan de forma atómica y que transforman el estado de los recursos o datos sobre los que se aplican de forma consistente. Para ello, una transacción debe satisfacer las siguientes características:

- **Atomicidad**. La transacción debe realizarse de forma atómica, incluso frente a fallos del sistema. Esto implica que la transacción no sólo debe ejecutarse de forma indivisible, sino que, además, debe realizarse completamente. En caso de que ocurriese un fallo en alguna de las operaciones de la transacción y ésta no pudiera completarse, las operaciones realizadas hasta el momento no deben tener ningún efecto sobre el sistema.
- **Consistencia**. El conjunto de operaciones que incluye la transacción deben cambiar el estado de los recursos o datos sobre los que se aplica de forma correcta sin que se produzcan inconsistencias.
- **Aislamiento** o **serialización**. Las transacciones que ejecutan de forma concurrente no deben interferir unas con otras y deben aparecer como una detrás de otra.
- **Persistencia**. Una vez que la transacción se ha completado el estado debe ser permanente.

Estas cuatro características se suelen referir con el acrónimo **ACID** (*Atomicity, Consistency, Isolation and Durability*).

Para implementar las transacciones se necesitan los siguientes elementos:

- Un gestor de transacciones que ofrezca unas primitivas de construcción de transacciones.
- Un almacenamiento estable. El almacenamiento estable garantiza la atomicidad de las escrituras. Esto lo puede hacer la aplicación escribiendo la información por duplicado o lo puede hacer el controlador de RAID.

6.8.1. Gestor de transacciones

Para poder hacer uso de transacciones es necesario el empleo de un gestor de transacciones. El sistema operativo no suele incorporar un gestor de transacciones por lo que debe incorporarse como un elemento adicional. Sin embargo, los gestores de base de datos sí suelen incorporarlo.

Un gestor de transacciones simplifica el desarrollo de aplicaciones que hacen uso de transacciones, ofreciendo un conjunto de servicios o primitivas de transacción:

- `Begintrans`. Mediante esta llamada la aplicación le indica al gestor de transacciones el comienzo de la transacción.

- `Endtrans o Commit`. Esta llamada indica el fin de la transacción. El gestor de transacciones devuelve para esta operación dos posibles valores: *completada* en caso de que la transacción se haya ejecutado con éxito o *abortada*, indicando que no se ha realizado ninguna operación de la transacción.
- `Aborttrans`. Esta llamada aborta la transacción en curso, devolviendo el estado del sistema a aquel que tenía justo antes de comenzar la transacción.

Para poder implementar las transacciones, el gestor de transacciones debe incorporar servicios para asegurar la atomicidad. Para ello, debe incorporar mecanismos de control de concurrencia. Las alternativas más habituales son:

- Cerrojos sobre los datos. En este caso, el gestor de transacciones utiliza cerrojos, con semántica similar a la de los *flock*, que utiliza cuando se accede a un dato y que se libera cuando finaliza la transacción.
- Control de concurrencia optimista. En este caso las transacciones se realizan de principio a fin sin emplear ningún bloqueo. Al final de la transacción se comprueba si ha habido algún conflicto en el acceso a recursos compartidos. Si es así, la transacción se aborta.
- Marcas de tiempo. Con marcas de tiempo, el gestor de transacciones registra los instantes en los que se realizan las operaciones de lectura y escritura sobre un dato o recurso compartido. Cada transacción compara su propia marca de tiempo con la de los datos, para determinar si la operación se puede realizar o no. Si no se puede realizar, la transacción se aborta.

La figura 6.27 muestra las relaciones entre los elementos involucrados en una transacción. La secuencia de eventos es la siguiente:

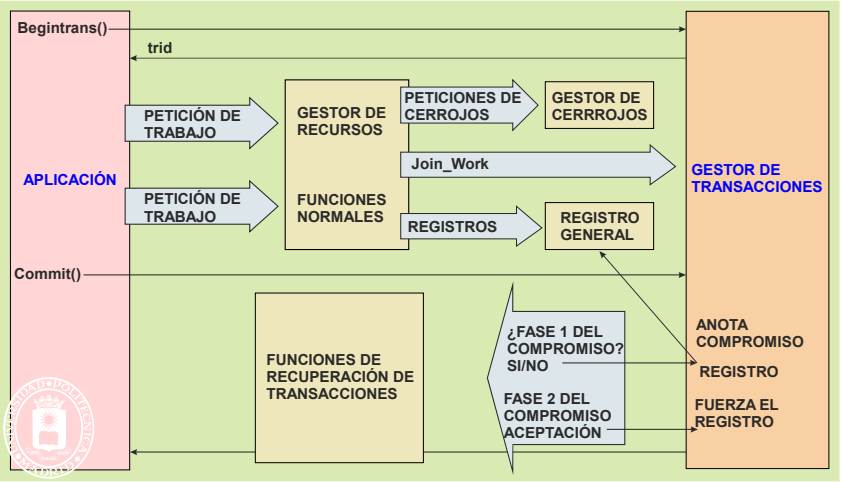


Figura 6.27 *Relaciones entre los distintos componentes involucrados en una transacción.*

1. La aplicación hace la llamada `Begintrans` al gestor de transacciones, que le devuelve un identificador de transacción (`trid`).
2. La aplicación típicamente hace peticiones a servidores como puede ser a una base de datos. En estas peticiones especifica el `trid`. El servidor ejecute la llamada `Join_Work` con el argumento `trid`, indicando al gestor de transacciones que se une a dicha transacción.
3. Todo el trabajo de la transacción se va anotando en el registro general, que debe residir en un almacenamiento estable.
4. Cuando la aplicación hace la llamada `Commit` el gestor de transacciones ejecuta el protocolo de cierre de transacción. Dependiendo de cómo se cierre la transacción el valor devuelto será de completada o abortada.
5. En caso de que la aplicación realice la llamada `Aborttrans`, por ejemplo, porque no consigue un recurso que necesita, el gestor de transacciones termina la transacción descartando todo el trabajo realizado y devolviendo un "abortada".

El protocolo de cierre de transacción es el **protocolo de compromiso en dos fases**, similar al de una boda, que tiene la siguiente secuencia:

Oficiante	**Contrayentes**
¿YY aceptas a XX?	
	SI
¿XX aceptas a YY?	
	SI
Quedáis casados	

El protocolo se muestra en la figura 6.28. Durante la primera fase, que comienza con la primitiva `Commit`, el gestor de transacciones prepara la transacción, para lo cual registra la preparación y solicita a todas las partes involucradas en la transacción la correspondiente preparación. Si todas las respuestas son positivas, pasa a la fase siguiente, en caso contrario aborta la transacción. En la siguiente fase envía el comprometido a todas las partes involucradas, que deben responder con el terminado. Cuando recibe todas las respuestas de terminación, da por finalizada la transacción.

Es de observar que si el sistema o una parte del sistema caen durante la fase 1 la transacción se aborta, desechando todos los posibles cambios. Pero, si el sistema cae durante la fase 2, al reiniciarse el sistema se puede com-

pletar la transacción sin problemas, puesto que el registro mantiene toda la información correspondiente a la transacción.

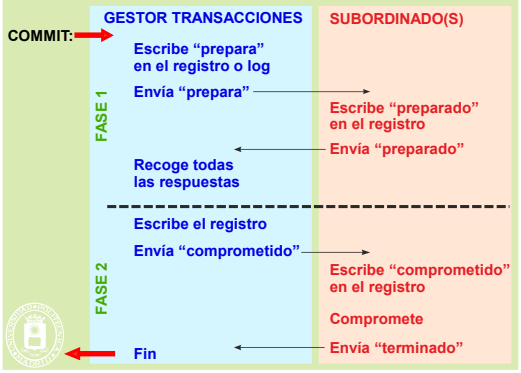


Figura 6.28 *Protocolo de dos fases utilizado en el cierre de una transacción.*

Si la transacción **aborta**, se ejecuta un procedimiento de recuperación que deja el estado del sistema intacto, es decir, se descartan todos los cambios tentativos realizados hasta el momento y se deja el estado del sistema en el aquel que tenía justo antes de comenzar la transacción. Esto se puede hacer de forma parecida al COW visto en el tema de memoria. Como muestra la figura 6.29, los bloques de información modificada se desdoblan manteniéndose la copia original y la modificada. Solamente al hacer el `Commit` se descartan los bloques originales. Por el contrario, en un `Aborttrans` se descartan los bloques modificados.

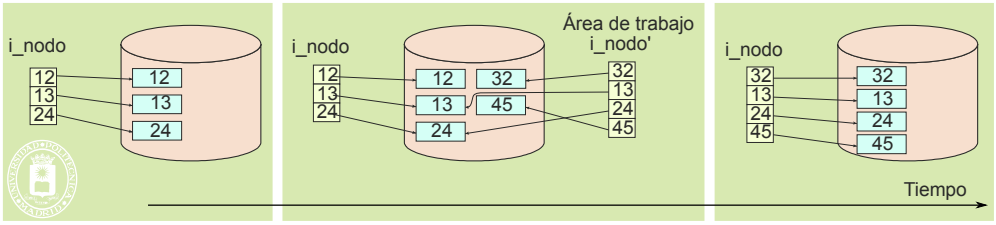


Figura 6.29 *Utilización de desdoblamiento de bloques para área de trabajo de una transacción.*

Con mucha frecuencia la aplicación requiere los servicios de gestores de recursos (por ejemplo bases de datos) que residen en diferentes máquinas, debiendo todo ello quedar englobado en una transacción. Nace, por tanto, el concepto de transacción distribuida que engloba el trabajo realizado en varios sistemas, como muestra la figura 6.30.

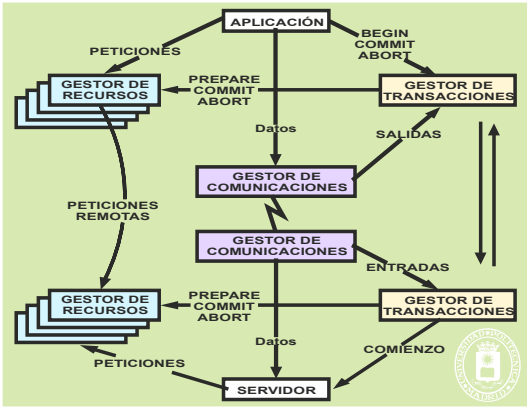


Figura 6.30 *Una transacción distribuida incluye la coordinación del trabajo realizado por gestores de recursos residentes en sistemas remotos. Para ello, es necesario disponer de un gestor de transacciones en cada sistema, así como la coordinación entre ellos.*

Por otro lado, la figura 6.31 muestra la evolución en el tiempo de las interacciones entre el usuario, el proceso cliente que le atiende, el gestor de transacciones y los servidores que se acceden.

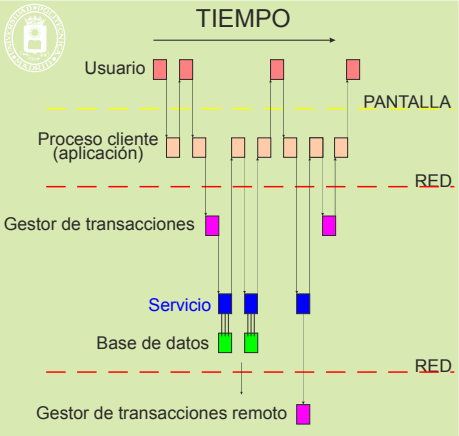


Figura 6.31 *Ejemplo de ejecución de una transacción interactiva. El usuario interacciona con un proceso cliente, que interacciona con el gestor de transacciones y los servidores.*

X/Open Distributed Transaction Processing Standard (X/Open DTP)

Definido por el consorcio de empresas Open Group el estándar X/Open DTP define una arquitectura de comunicaciones con la siguiente funcionalidad:

- Ejecución concurrente de aplicaciones sobre recursos compartidos.
- Coordinación de las transacciones de todas las aplicaciones.
- Definición de los componentes, interfaces y protocolos que constituye la arquitectura y posibilitan la portabilidad de las aplicaciones.
- Atomicidad de los sistemas de transacciones.
- Control mono-hilo y secuenciamiento de las llamadas a funciones.

El estándar X/Open DTP XA define las interfaces de programación de aplicaciones que utiliza un gestor de recursos para comunicar con el gestor de transacciones, de forma que pueda: a) unirse a una transacción, b) ejecutar el `commit` de dos fases y c) recuperarse de una transacción fallida.

6.8.2. Transacciones e interbloqueo

Las transacciones permiten, además, resolver el problema de interbloqueo, actuando de la siguiente forma: En su ejecución, la transacción va obteniendo los recursos que necesita, por ejemplo, los registros de una base de datos que debe leer y modificar. Si puede obtener todos los recursos necesarios, y no ocurre ningún error, la transacción termina satisfactoriamente. Por el contrario, si se encuentra que algún registro que necesita está bloqueado por otra transacción, la transacción aborta liberando todos los recursos asignados hasta ese momento. La entidad que ha lanzado la transacción deberá, si es necesario, volver a lanzar dicha transacción más tarde, con la esperanza de que ya no se reproduzca el problema.

6.9. ASPECTOS DE DISEÑO

En esta sección se describen los principales aspectos de diseño e implementación subyacentes a los mecanismos de sincronización. También se describen la sincronización y comunicación dentro del propio sistema operativo.

6.9.1. Soporte *hardware* para la sincronización

Como se ha visto, la sincronización entre procesos, como ocurre por ejemplo en la exclusión mutua en el acceso a un recurso compartido, puede implicar una espera en la ejecución de los procesos. Para asegurar este bloqueo el sistema operativo necesita mecanismos *hardware*. A continuación se describen los principales mecanismos que ofrece el *hardware* para implementar bloqueos.

Deshabilitar las interrupciones

Una forma sencilla de asegurar la sincronización en un computador con una única UCP cuando varios procesos intentan acceder a una sección crítica es deshabilitar las interrupciones cuando un proceso entra a ejecutar código de la sección crítica. En efecto, si las interrupciones están deshabilitadas mientras un proceso ejecuta código de la sección crítica, éste no podrá ser interrumpido y por tanto podrá ejecutar todo el código de la sección crítica de forma atómica sin ser expulsado. Esto evitará la aparición de condiciones de carrera. Los diferentes procesos que acceden a una sección crítica podrían utilizar la siguiente estructura para resolver el acceso a la misma.

Deshabilitar interrupciones
Sección crítica

Habilitar interrupciones

Esta solución presenta varios problemas. En primer lugar, esta solución no es aceptable para sincronizar procesos de usuario. ¿Qué ocurriría si un proceso permaneciera indefinidamente dentro de la sección crítica por un error? En este caso, como las interrupciones están inhabilitadas no se podría interrumpir la ejecución del proceso y habría que reiniciar el sistema. En segundo lugar, no es una solución adecuada para un sistema multiprocesador, puesto que deshabilitar las interrupciones sólo afecta al procesador donde se ejecuta la instrucción que deshabilita las interrupciones. Por último, aunque deshabilitar las interrupciones es una técnica que puede utilizarse para sincronizar acciones dentro del núcleo del sistema operativo (como se verá en la sección 6.9.4 "Sincronización dentro del sistema operativo"), sólo sería razonable su uso cuando la sección crítica incluyese muy pocas instrucciones. Si la sección crítica dura mucho, podrían perderse interrupciones importantes.

Instrucciones de máquina especiales

Todos los procesadores incluyen una o más instrucciones de máquina especiales que se pueden utilizar para implementar mecanismos de espera. Estas instrucciones, que son atómicas como la gran mayoría de las instrucciones máquina, se basan en la ejecución atómica (su ejecución no puede ser interrumpida) de dos acciones, normalmente lectura y escritura, sobre una posición de memoria. Las instrucciones más representativas son las siguientes:

Instrucción *test-and-set*

Esta instrucción máquina se aplica sobre una posición de memoria. El formato de esta instrucción es `T&S Reg, var`. La instrucción realiza, de forma atómica, dos acciones: lee el contenido de una posición de memoria (variable `var` en la instrucción anterior) en un registro del computador (registro `Reg`) y escribe un uno en la posición de memoria (variable `var`). Utilizando esta instrucción se puede implementar el acceso a una sección crítica mediante la siguiente estructura de código:

```
Bucle:   T&S .R1, /cerrojo
         CMP .R1, #1                ; compara .R1 con 1
         BZ     $Bucle              ; si .R1 es 1 saltar a Bucle
         <Sección crítica >
         LD .R1, #0
         ST    .R1, /cerrojo        ; almacena un 0 en cerrojo
```

La variable `cerrojo` se comparte entre todos los procesos y su valor inicial debe 0. Como se verá en la siguiente sección el proceso realiza una espera activa para poder entrar a ejecutar el código de la sección crítica.

Instrucción *swap*

Esta instrucción, *`swap`* `.Reg, var`, se aplica sobre un registro (`.Reg`) y una posición de memoria (var) y de forma atómica intercambia los valores del registro y de la posición de memoria. Si se utiliza la instrucción *`swap`*, el problema de la sección crítica puede resolverse de la siguiente manera:

```
         LD        .R1, #1
Bucle:   SWAP      .R1, /cerrojo
         CMP     .R1, #1            ; compara .R1 con 1
         BZ        $Bucle           ; si .R1 es 1 saltar a Bucle
         <Sección crítica>
         LD .R1, #0
         ST    .R1, /cerrojo        ; almacena un 0 en cerrojo
```

La variable `cerrojo` se comparte entre todos los procesos y su valor inicial debe ser 0. El registro `.R1` es un registro y por tanto es local a cada proceso.

Instrucciones *hardware* en multiprocesadores

En el caso de un multiprocesador, las instrucciones *hardware* de sincronización tienen que garantizar que el acceso atómico por parte de un procesador a una posición de memoria compartida impide al resto de procesadores el acceso a esa posición de memoria mientras dura el acceso. En el caso de un multiprocesador las instrucciones atómicas para sincronización deben apoyarse en el *hardware* del sistema de memoria del multiprocesador y en especial en el *hardware* para el mantenimiento de la coherencia en el sistema de memoria [5].

El empleo en multiprocesadores de instrucciones clásicas de sincronización, como las vistas anteriormente (`test-and-set` y *`swap`*) para resolver el acceso a una sección crítica, generan, mientras el cerrojo está ocupado,

5 Un multiprocesador, como se vio en el capítulo "1 Conceptos arquitectónicos del computador", consta de un conjunto de procesadores que comparten una memoria y en el que normalmente todos estos componentes están conectados por un bus. Para minimizar la contención sobre el bus y mejorar la escalabilidad, los multiprocesadores introducen una memoria cache local para cada procesador. Esta solución reduce el tráfico en el bus pero introduce un problema de coherencia cuando se actualizan variables compartidas que se almacenan en las memorias caches. Para resolver estos problemas los multiprocesadores utilizan *protocolos de coherencia de cache*, que resuelven el problema de la coherencia haciendo que cada escritura sea visible a todos los procesadores.

lecturas consecutivas desde un procesador sobre la variable cerrojo, lo que provoca un excesivo tráfico en el bus del multiprocesador debido a los protocolos de mantenimiento de la coherencia. Para reducir el número de transacciones en el bus, se han ideado otras alternativas a las instrucciones *hardware* tradicionales. Una solución es el empleo de la instrucción **`test-and-set` con retardo**. En este caso cuando un proceso se encuentra dentro de la sección crítica, el resto no ejecuta la instrucción `test-and-set` de forma continua, sino introduciendo un retardo después de aquellos intentos de adquisición del cerrojo que no tengan éxito.

Otra alternativa en multiprocesadores es el ejemplo del par de instrucciones **`load-locked`** y **`store conditional`** (`LL`/`SC`). La instrucción de carga (`LL .Reg, var`) realiza, al igual que una instrucción de carga convencional, la carga de una posición de memoria en un registro pero guarda información de que se ha accedido a la posición de memoria reservando el bloque de cache en el que está la palabra. El `store` condicional (`SC .Reg, var`) intenta actualizar el contenido de una posición de memoria (posición `var`). La actualización sólo tiene éxito si se mantiene la reserva, es decir, almacena un determinado valor en una posición de memoria sólo si desde que se ejecutó la instrucción `LL` no se ha realizado sobre la variable ningún otro `store` condicional. En caso contrario, no se escribe ni se generan invalidaciones por parte del protocolo de coherencia de caches. Con este par de instrucciones se puede implementar el acceso a una sección crítica de la siguiente forma, asumiendo que la variable `cerrojo` toma valor inicial 0:

```
Bucle:  LL      .R1, /cerrojo
        CMP .R1, #1             ; compara .R1 con 1
        BZ      $Bucle          ; si .R1 es 1 saltar a Bucle
        LD      .R2, #1
        SC      .R2, /cerrojo
        CMP .R2, #0
        BZ      Bucle
        <Sección crítica>
        LD .R1, #0
        ST    .R1, /cerrojo     ; almacena un 0 en cerrojo
```

6.9.2. Espera activa

En la sección anterior se ha visto cómo implementar el acceso a una sección crítica utilizando instrucciones máquina especiales. El empleo de estas instrucciones de máquina permite construir **cerrojos**, un mecanismo que proporciona una forma de asegurar la exclusión mutua. Sobre un cerrojo se utilizan dos funciones, similares a las empleadas sobre los *mutex*:

- **`lock(cerrojo)`**. Esta operación intenta bloquear el cerrojo. Si el cerrojo ya está bloqueado por otro proceso, el proceso que realiza la operación se queda esperando de forma activa. En caso contrario se cierra el cerrojo y el proceso puede continuar.
- **`unlock(cerrojo)`**. Esta operación abre el cerrojo, permitiendo que alguno de los procesos que se encuentra esperando en la función `lock` lo adquiera y continué su ejecución.

Utilizando la primitiva `test-and-set` se pueden implementar estas funciones de la siguiente forma:

```
lock(cerrojo) {
    Bucle:  T&S .R1, cerrojo
            CMP .R1, #1         ; compara .R1 con 1
            BZ      $Bucle      ; si .R1 es 1 saltar a Bucle
}

unlock(cerrojo) {
        LD .R1, #0
        ST    .R1, /cerrojo     ; almacena un 0 en cerrojo
}
```

Basándose en este esquema, se puede construir una sección crítica de la siguiente forma (asumiendo que `cerrojo` tiene valor inicial 0):

```
lock(cerrojo);
Sección Crítica
unlock(cerrojo);
```

Este tipo de cerrojo así implementado se conoce como *spin-lock* y se basa en el uso de una variable como un cerrojo. La variable, como se puede ver en la estructura anterior, se consulta en un bucle de forma continua hasta que tenga un valor que indique que el cerrojo está abierto. Esta consulta debe realizarse como se ha comentado anteriormente mediante una primitiva atómica de consulta y actualización de memoria, como es la instrucción `test-and-set` en este caso.

Como se puede ver en la solución planteada, el bloqueo de un proceso se basa en un bucle que consulta de forma continua y atómica el valor de una posición de memoria. A este tipo de bloqueo se le denomina **espera activa** o **espera cíclica**. Cuando se emplea espera activa para bloquear a los procesos, estos deben ejecutar un ciclo continuo

hasta que puedan continuar. La espera activa es obviamente un problema en un sistema multiprogramado real, ya que se desperdician ciclos del procesador en aquellos procesos que no pueden ejecutar y que no realizan ningún trabajo útil. Este malgasto de ciclos es especialmente importante en un sistema con una sola UCP, puesto que un proceso que no puede entrar en la sección crítica no puede continuar su ejecución y por otra parte impide que el proceso que se encuentra dentro de la sección crítica avance y libere la sección lo antes posible.

La espera activa es un mecanismo, sin embargo, que se suele emplear para la sincronización dentro del propio sistema operativo en sistemas multiprocesadores (véase la sección 6.9.4 "Sincronización dentro del sistema operativo").

6.9.3. Espera pasiva o bloqueo

Para resolver el problema que plantea la espera activa, los sistemas operativos recurren al bloqueo del proceso que no puede continuar su ejecución. En este caso, cuando un proceso no puede continuar la ejecución el sistema operativo lo **bloquea** suspendiendo su ejecución. A este modelo de espera se le denomina **espera pasiva**. La operación de bloqueo coloca al proceso es una cola de espera. Este bloqueo transfiere el control al planificador del sistema operativo, que seleccionará otro proceso para su ejecución. De esta forma no se desperdician ciclos de procesador en ejecutar operaciones inútiles. Este mecanismo es el que emplea el sistema operativo para implementar las operaciones de sincronización sobre *mutex* o semáforos. En estas situaciones normalmente cada recurso (semáforo, *mutex*, etc.) lleva asociado una cola de espera. Cuando un proceso se bloquea en una operación sobre el recurso, el sistema operativo suspende su ejecución e inserta al proceso en la cola de espera asociada al recurso.

Cuando un proceso libera el recurso, el sistema operativo extrae uno de los procesos de la lista de procesos bloqueados y cambia su estado a listo para ejecutar.

Se va a ilustrar el empleo de la espera pasiva en la implementación de un semáforo (el empleo para cualquier otro mecanismo de sincronización sería similar). Para implementar un semáforo, o cualquier otro tipo de mecanismo de sincronización, utilizando espera pasiva, basta con asignar al semáforo (o al mecanismo de sincronización concreto) una lista de procesos, en la que se irán introduciendo los procesos que se bloqueen. Una forma sencilla de implementar esta lista de procesos es mediante una lista cuyos elementos apunten a las entradas correspondientes de la tabla de procesos.

Cuando un proceso debe bloquearse en el semáforo, el sistema operativo realiza los siguientes pasos:

- Inserta al proceso en la lista de procesos bloqueados en el semáforo.
- Cambia el estado del proceso a bloqueado.
- Llama al planificador para elegir otro proceso a ejecutar y a continuación al activador.

En la figura 6.32 se muestra lo que ocurre cuando un proceso (con identificador de proceso 11) ejecuta una operación `wait` sobre un semáforo con valor –1 (figura 6.32.a). Como el proceso debe bloquearse, el sistema operativo añade este proceso a la lista de procesos bloqueados en el semáforo y pone su estado como bloqueado (figura 6.32.b).

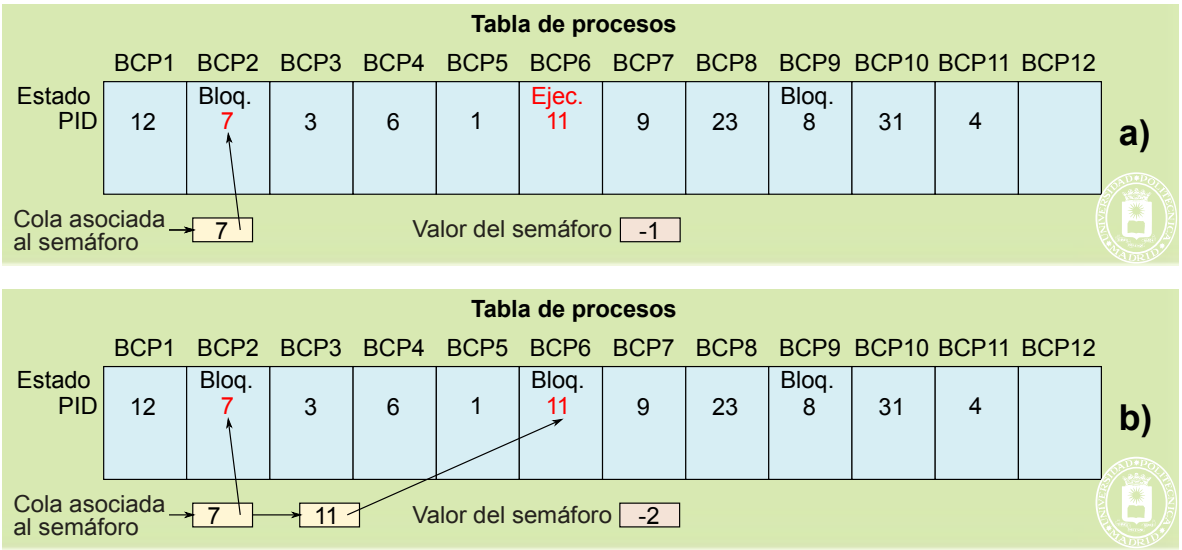


Figura 6.32 *Acciones realizadas por el sistema operativo cuando hay que bloquear un proceso en un semáforo (el proceso 11 ejecuta un* `wait`*).*

Cuando se realiza una operación `signal` sobre el semáforo, el sistema operativo realiza las siguientes acciones:

- Extrae al primer proceso de la lista de procesos bloqueados en el semáforo.
- Cambia el estado del proceso a listo para ejecutar.

- Llama al planificador para elegir a otro proceso (que puede ser el que llama a `signal` o el que se despierta) y a continuación al activador.

Estas acciones se describen en la figura 6.33.

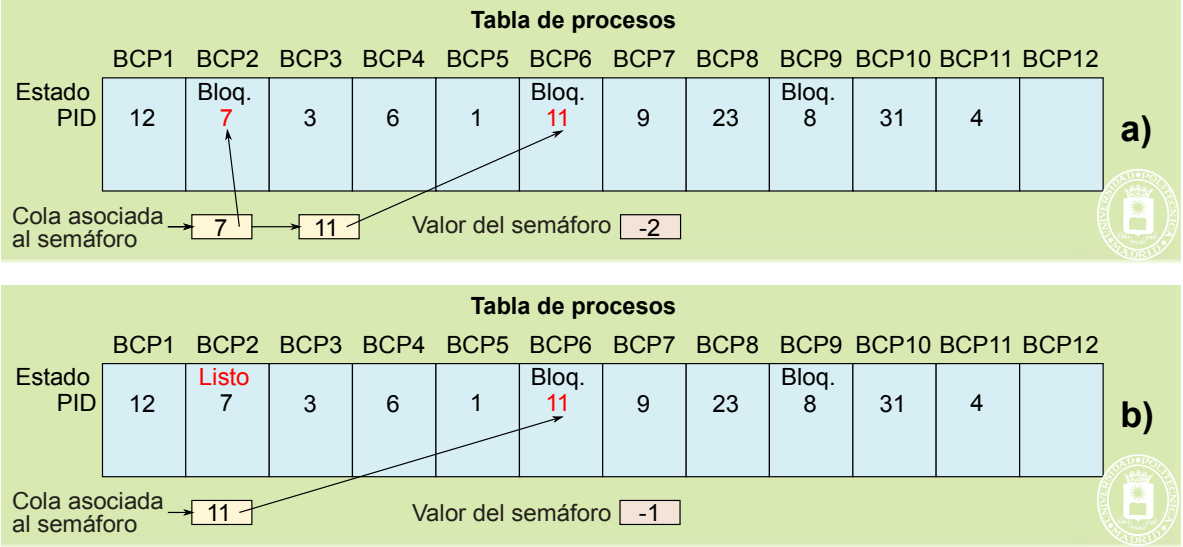


Figura 6.33 *Acciones realizadas por el sistema operativo en una operación* `signal` *sobre un semáforo.*

Aparte de estas operaciones, un aspecto importante en la implementación de un semáforo (y de otros mecanismos de sincronización), es que las operaciones `wait` y `signal` deben ejecutarse de forma atómica, es decir, en exclusión mutua. Para conseguir esta exclusión mutua, los sistemas operativos recurren a las instrucciones *hardware* de sincronización que permiten resolver el problema de la sección crítica. Así por ejemplo, utilizando una operación `lock`, como la descrita en la sección anterior, sobre un *spin-lock*, un semáforo vendría definido por una estructura que almacena: el valor del semáforo, la lista de procesos bloqueados y el *spin-lock* a utilizar. La definición de las operaciones `wait` y `signal` en este caso sería la siguiente:

```
wait(s){
   lock(spin-lock);
   s = s - 1;
   if (s < 0){
      unlock(spin-lock);
      Bloquear el proceso;
   }
   else
      unlock(spin-lock);
}
signal(s){
   lock(spin-lock);
   s = s + 1;
   if ( s <= 0)
      Desbloquear un proceso bloqueado en la operación wait;
   unlock(spin-lock);
}
```

Con la implementación anterior no se ha eliminado del todo la espera activa pero, sin embargo, se ha reducido a porciones de código muy pequeñas dentro de las operaciones `wait` y `signal`. Lo importante es que los procesos cuando se bloquean en la operación `wait`, no realizan espera activa.

6.9.4. Sincronización dentro del sistema operativo

El sistema operativo es un programa con un alto grado de concurrencia y de asincronía. En cada momento se pueden producir múltiples eventos de forma asíncrona, lo que da lugar a problemas de sincronización muy complejos. Este hecho ha causado que los sistemas operativos sean tradicionalmente un módulo *software* con una tasa de errores apreciable. Hay que resaltar además que la mayoría de los problemas clásicos de sincronización que aparecen en la literatura sobre la programación concurrente provienen del ámbito de los sistemas operativos.

En el caso de un sistema monoprocesador, se presentan dos tipos de problemas de concurrencia dentro del sistema operativo, que se estudian a continuación analizando sus posibles soluciones.

El primer escenario sucede cuando ocurre una interrupción con un nivel de prioridad superior, mientras se está ejecutando código del sistema operativo vinculado con el tratamiento de una llamada, una excepción o una interrup-

ción. Se activará la rutina de tratamiento correspondiente, que puede entrar en conflicto con la labor que ha dejado a medias el flujo de ejecución interrumpido. Piense, por ejemplo, en una llamada o rutina de interrupción que manipula la lista de procesos listos que es interrumpida por una interrupción que también modifica esta estructura de datos. Puede ocurrir una condición de carrera que deje corrupta la lista.

Supóngase una hipotética función que inserta un BCP al final de una lista usando el siguiente fragmento de código:

```
insertar_ultimo(lista, BCP){
   lista->ultimo->siguiente = BCP;
   lista->ultimo = BCP;
}
```

Considere que dentro de una llamada al sistema se desbloquea un proceso y se usa la rutina de inserción anterior para incorporarlo en la cola de listos. Podría ser, por ejemplo, una llamada *unlock* que deja libre un *mutex*, desbloqueando uno de los procesos que estuviera esperando para tomar posesión del *mutex*:

```
unlock(mimutex) {
   .....
   Si (hay procesos esperando) {
      Seleccionar un proceso P;
      insertar_ultimo(lista_listos, P);
   }
   .....
}
```

Asimismo, suponga una rutina de interrupción, como, por ejemplo, la del terminal, que puede desbloquear un proceso que estuviera esperando un determinado evento. La rutina de interrupción podría insertar el proceso bloqueado en la cola de listos usando esa misma función:

```
int_terminal {
   .....
   Si (hay procesos esperando) {
      Seleccionar un proceso Q;
      insertar_ultimo(lista_listos, Q);
   }
   .....
}
```

Supóngase que, mientras se está ejecutando la rutina *insertar_ultimo* dentro de *unlock* para insertar un proceso *P* al final de la cola de listos, habiéndose ejecutado sólo la primera sentencia, llega una interrupción del terminal que desbloquea a un proceso *Q* incorporándolo a la lista de listos. Cuando retorna la rutina de interrupción, se completa la inserción realizada por *unlock*, pero la lista de listos queda corrupta:

- El puntero al último elemento que hay en la cabecera de la lista quedará apuntando a *P*.
- Sin embargo, ningún elemento de la lista hará referencia a *P* como su sucesor.

En el segundo escenario, sucede que, mientras se está realizando una llamada al sistema, se produce un cambio de contexto a otro proceso (por ejemplo, debido a que se ha terminado la rodaja del proceso actual). Este proceso a su vez puede ejecutar una llamada al sistema que entre en conflicto con la llamada previamente interrumpida. Se produce, por tanto, la ejecución concurrente de dos llamadas al sistema. Esta situación puede causar una condición de carrera. Así, por ejemplo, dos llamadas concurrentes que intenten crear un proceso en un sistema que usa una tabla de procesos podrían acabar obteniendo el mismo BCP libre. Supóngase que el siguiente fragmento corresponde con el código de la llamada al sistema que crea un proceso:

```
crear_proceso(...) {
   ...........
   pos = BuscarBCPLibre();
   tabla_procesos[pos].ocupada = true;/* marca la entrada como ocupada */
   ...........
}
```

Podría ocurrir un cambio de contexto involuntario (por ejemplo, por fin de rodaja) justo después de que la función *BuscarBCPLibre* se haya ejecutado pero antes de poner la entrada correspondiente de la tabla de procesos como ocupada. El proceso activado por el cambio de contexto involuntario podría invocar también la llamada *crear_proceso*, con lo que habría dos llamadas ejecutándose concurrentemente. Esta segunda invocación de *crear_proceso* causaría una condición de carrera, ya que seleccionaría la misma posición de la tabla de procesos que la llamada interrumpida.

De los escenarios planteados, se deduce la necesidad de crear secciones críticas dentro de la ejecución del sistema operativo. Evidentemente, éste no es problema nuevo. Es el mismo que aparece en cualquier programa concurrente como se ha visto en secciones anteriores de este capítulo, que se resuelve con mecanismos de sincronización como semáforos o *mutex*. Sin embargo, en este caso se presentan dos dificultades adicionales:

- Estos mecanismos de sincronización se basan en que sus primitivas se ejecutan de forma atómica. ¿Cómo se puede conseguir este comportamiento dentro del sistema operativo?
- Desde una rutina de interrupción, como se ha explicado reiteradamente, no se puede usar una primitiva que cause un bloqueo.

A continuación, se estudia el problema distinguiendo si se trata de un sistema con un núcleo expulsable o no expulsable. Después, se analizará qué dificultades adicionales se presentan cuando se usa un sistema multiprocesador.

Sincronización en un núcleo no expulsable

El modo de operación de los sistemas con un núcleo no expulsable facilita considerablemente el tratamiento de los problemas de sincronización al limitar el grado de concurrencia en el sistema. A continuación, se analizan los dos escenarios problemáticos planteados.

En primer lugar, se trata el problema de concurrencia que surge cuando una rutina del sistema operativo es interrumpida por una rutina de interrupción de mayor prioridad. La solución habitual es elevar el nivel de interrupción del procesador durante el fragmento correspondiente para evitar la activación de la rutina de interrupción conflictiva. Es importante resaltar que se debería elevar el nivel interrupción del procesador justo lo requerido y minimizar el fragmento de código durante el cual se ha elevado explícitamente dicho nivel. Así, por ejemplo, en un fragmento de código del sistema operativo donde se manipula el *buffer* de un terminal, bastaría con inhibir la interrupción del terminal, pudiendo seguir habilitadas otras interrupciones, como, por ejemplo, la del reloj.

En el ejemplo planteado anteriormente, donde la estructura conflictiva era la cola de procesos listos, habrá que elevar el nivel al máximo para prohibir todas las interrupciones, puesto que se trata de una estructura de datos que manipulan todas ellas[6].

En cuanto a los problemas de sincronización entre llamadas concurrentes, con este tipo de núcleo no existe este problema ya que no se permite la ejecución concurrente de llamadas: una llamada al sistema continúa su ejecución hasta que termina o causa el bloqueo del proceso, pueden activarse rutinas de tratamiento de interrupción, pero no causarán un cambio de contexto. Por tanto, no existirá este problema de sincronización en este tipo de núcleos no expulsables. En el ejemplo planteado, la rutina *crear_proceso* podrá marcar como ocupada la entrada encontrada sin ninguna interferencia.

Hay que resaltar que, si el proceso se bloquea, cuando continúe su ejecución posteriormente, la situación en el sistema puede haber cambiado significativamente. Una condición que se cumplía antes del bloqueo puede haberse dejado de satisfacer cuando se reanuda la ejecución, debido a que ha podido cambiar por la ejecución de otros procesos en ese intervalo de tiempo. La llamada al sistema debería comprobarla de nuevo y tomar las acciones correctivas oportunas (por ejemplo, volver a bloquearse esperando que el estado sea el adecuado). Asimismo, si una llamada bloqueante necesita asegurarse de que otras llamadas no utilizan un determinado recurso mientras el proceso está bloqueado en esa llamada, deberá indicarlo en algún campo asociado al recurso. Este campo será consultado por las llamadas afectadas, que se bloquearán si el recurso está reservado por una llamada bloqueada. Por ejemplo, considere una llamada de escritura de un fichero que debe asegurar que no haya escrituras simultáneas sobre el mismo fichero. Para ello, puede usar un campo del *nodo-i* del fichero que indique esta circunstancia, como se muestra en el siguiente fragmento:

```
write(...) {
    ...........
    Si (nodo-i.en_uso)
        Bloquear(lista_nodo-i);

    nodo-i.en_uso = true;
    ...........
    Posibles bloqueos mientras escribe los bloques afectados
    ...........
    nodo-i.en_uso = false;
    Si (procesos bloqueados en lista_nodo-i)
    Desbloquear(lista_nodo-i);
    ...........
}
```

Nótese que no habría condiciones de carrera en el uso del campo `en_uso` puesto que las llamadas al sistema son atómicas en este tipo de sistemas.

Dadas las características del modelo no expulsable, la implementación de mecanismos de sincronización, tales como los semáforos, sería directa. Así, si se pretende ofrecer un mecanismo de tipo semáforo a las aplicaciones, las

6 Realmente, con frecuencia, las rutinas de interrupción no manipulan la cola de listos, ya que las operaciones de desbloqueo las aplazan para realizarlas dentro del tratamiento de una interrupción software. Por tanto, en este caso, bastaría con elevar el nivel para inhibir este tipo de interrupciones. Precisamente, sí haría falta elevar el nivel al máximo a la hora de manipular la lista de operaciones pendientes vinculadas con la interrupción software, ya que todas las rutinas de interrupción acceden directamente a esta estructura de datos.

llamadas al sistema que implementan las primitivas del semáforo, no necesitarían ninguna técnica para asegurar su comportamiento atómico, puesto que toda llamada al sistema es atómica.

Por último, como resumen, es interesante resaltar de qué tipo es el análisis de concurrencia que debe llevar a cabo el diseñador del sistema operativo en un sistema no expulsable:

- Hay que estudiar cada parte del código de una llamada al sistema para ver si puede verse afectada por la ejecución de una rutina de interrupción. Si una determinada parte del código de la llamada se ve afectado por la rutina de interrupción de nivel N, durante ese fragmento se elevará el nivel de interrupción del procesador al valor N.
- Se debe hacer un proceso similar con cada rutina de interrupción: se estudia su código y se comprueba si en alguna parte puede verse afectado por una interrupción de nivel superior. En caso afirmativo, se eleva el nivel de interrupción para evitar el problema de condición de carrera.

Evidentemente, cada vez que se incluye nuevo código en una rutina de interrupción, hay que repetir el análisis de conflictos para todas las rutinas de menor prioridad, que podrían verse afectadas por este nuevo código.

Sincronización en un núcleo expulsable

Este tipo de sistema presenta más dificultades a la hora de lograr una correcta sincronización dado que implica una mayor concurrencia.

En cualquier caso, la solución frente al problema que surge cuando una rutina del sistema operativo es interrumpida por una rutina de interrupción de mayor prioridad, es la misma que para un núcleo no expulsable: elevar el nivel de interrupción del procesador durante el fragmento correspondiente para evitar la activación de la rutina de interrupción conflictiva.

El problema principal de este tipo de núcleo reside en la sincronización entre llamadas concurrentes. Dado que la ejecución de llamadas al sistema por parte de varios procesos puede verse entremezclada de forma impredecible, hay que asegurarse de que no se produzcan problemas de condiciones de carrera estableciendo las secciones críticas que se requieran. Para ello, se puede elevar el nivel de interrupción de manera que se inhiban las interrupciones *software* asociadas al cambio de contexto involuntario durante el fragmento conflictivo, lo que asegura que mientras se ejecuta dicho fragmento no se intercala la ejecución de otra llamada. Aplicándolo al ejemplo planteado previamente, quedaría un esquema como el siguiente:

```
crear_proceso(...) {
    ...........
    nivel_anterior = fijar_nivel_int(NIVEL_1);
    pos=BuscarBCPLibre();
    tabla_procesos[pos].libre = false;
    fijar_nivel_int(nivel_anterior);
    ...........
}
```

La solución planteada resuelve el problema, pero puede ser inadecuada si la sección crítica es larga (en el ejemplo, es posible que `BuscarBCPLibre` consuma un tiempo apreciable). En ese caso, se empeoraría el tiempo de respuesta de los procesos. Si un proceso urgente se desbloquea mientras otro poco prioritario está en la sección crítica, no comenzará a ejecutar hasta que este segundo proceso concluya la sección crítica. Nótese que, si hay secciones críticas largas, el núcleo tiende a ser no expulsable. Asimismo, hay que resaltar que esta estrategia afecta a todos los procesos, con independencia de si el proceso ejecuta código que entre en conflicto con el que mantiene la sección crítica. Además, esta solución no sería válida si la sección crítica puede incluir el bloqueo del proceso.

La solución habitual es implementar un semáforo o algún mecanismo de sincronización equivalente. Para lograr la atomicidad en las operaciones del semáforo se recurre al mecanismo anterior: elevar el nivel de interrupción de manera que se inhiban las interrupciones *software*. Sin embargo, en este caso, esta prohibición de las interrupciones *software* sólo afectaría a las operaciones sobre el semáforo, que son muy breves, pudiendo ejecutarse la sección crítica con todas las interrupciones habilitadas. Con esta técnica, el ejemplo anterior quedaría de la siguiente forma:

```
crear_proceso(...) {
    ...........
    bajar(semaforo_tabla_procesos);

    /* Sección crítica ejecuta con inter. habilitadas */
    pos=BuscarBCPLibre();
    tabla_procesos[pos].libre = false;

    subir(semaforo_tabla_procesos);
    ...........
}
```

Usando esta estrategia, sólo se ejecuta con las interrupciones *software* inhibidas durante muy poco tiempo (el correspondiente a las operaciones del semáforo). Asimismo, la sección crítica propiamente dicha sólo involucra a los procesos afectados por la misma, pudiendo, además, incluir el bloqueo del proceso.

De todas formas, hay que tener en cuenta que si la sección crítica es muy corta, será más adecuado utilizar directamente la estrategia de inhabilitar las interrupciones *software*, puesto que la solución basada en semáforos inclu-

ye una cierta sobrecarga debida a las operaciones vinculadas con los mismos. Además, conlleva cambios de contexto por los bloqueos que se producen al competir por el semáforo, que no se generan en la solución que no los utiliza.

El diseñador del sistema operativo tiene que determinar qué semáforos usar para proteger las distintas estructuras de datos del sistema operativo. Para ello, debe establecer un compromiso con respecto a la granularidad del semáforo, es decir, en cuanto a la cantidad de información que protege un determinado semáforo: cuanta más información sea protegida por un semáforo, menos concurrencia habrá en el sistema, pero menos sobrecarga causada por la gestión de los semáforos. A continuación, se plantea un ejemplo hipotético para ilustrar este aspecto.

Supóngase un sistema UNIX en el que se requiere controlar el acceso a dos estructuras de datos en memoria vinculadas con el sistema de ficheros: por ejemplo, la tabla intermedia de ficheros, donde se almacenan los punteros de posición de todos los ficheros abiertos en el sistema, y la tabla de *nodos-i*, que guarda en memoria los *nodos-i* de los ficheros que están siendo utilizados en el sistema. Considere en este punto las dos alternativas posibles.

Si se establecen sendos semáforos para el control de acceso a cada estructura, podrán ejecutar concurrentemente dos llamadas si cada una de ellas sólo involucra a una de las estructuras. Sin embargo, si una llamada determinada requiere el uso de ambas estructuras, como, por ejemplo, la apertura de un fichero, deberá cerrar y abrir ambos semáforos, con la consiguiente sobrecarga.

En caso de que se defina un único semáforo para el control de acceso a ambas estructuras, se limita la concurrencia, puesto que dos llamadas, tal que cada una de ellas sólo usa una de las estructuras de datos, no podrán ejecutarse concurrentemente. Sin embargo, una llamada que usa ambos recursos tendrá menos sobrecarga, ya que será suficiente con cerrar y abrir un único semáforo.

Del análisis previo se concluye que sería conveniente usar dos semáforos si hay un número significativo de llamadas que usan de forma independiente cada recurso, mientras que sería adecuado usar un único semáforo si la mayoría de las llamadas usa ambos recursos conjuntamente. Nótese que llevándolo a un extremo, se puede establecer un semáforo para todo el sistema operativo, lo que haría que el sistema se comportara como un núcleo no expulsable.

Se debe incidir en que los semáforos no son aplicables a los problemas de sincronización entre una rutina del sistema operativo y una rutina de interrupción de mayor prioridad puesto que una rutina de interrupción no se puede bloquear, como se ha comentado reiteradamente a lo largo de esta presentación.

Usando la misma estrategia que se emplea para implementar los semáforos, se puede construir cualquier mecanismo de sincronización similar, tanto para uso interno como para exportar a las aplicaciones. En Windows, aplicando los conceptos presentados en esta sección, el sistema operativo implementa una primitiva de sincronización, denominada *dispatcher object*, en la que se basan los distintos mecanismos de sincronización exportados a las aplicaciones.

Como última consideración, y para comparar con la sincronización en sistemas no expulsables, se debe resaltar que en los sistemas expulsables el análisis de los problemas de sincronización que debe realizar el diseñador del sistema operativo se complica apreciablemente. A los problemas ya existentes en los sistemas no expulsables, hay que añadir los que conlleva la ejecución concurrente de llamadas, que, en principio, requeriría analizar las posibles interferencias entre todas las llamadas al sistema.

Sincronización en multiprocesadores

Los problemas de sincronización dentro del sistema operativo se complican considerablemente en un sistema multiprocesador. Hay que tener en cuenta que en este tipo de sistemas la concurrencia, que en un monoprocesador implica la ejecución alternativa de distintos procesos, se corresponde con la ejecución en paralelo de los procesos. Evidentemente, esto dificulta la correcta sincronización y hace que algunas estrategias utilizadas en monoprocesadores no sean válidas para sistemas multiprocesador.

Para empezar, además de las dos situaciones conflictivas planteadas en sistemas monoprocesador (una rutina de un evento de mayor prioridad que interrumpe la ejecución de otra rutina y la ejecución concurrente de varias llamadas al sistema), se producen otras adicionales, puesto que, mientras se está ejecutando una rutina de interrupción de una determinada prioridad en un procesador, se pueden estar ejecutando rutinas de interrupción menos prioritarias y llamadas al sistema en otros procesadores.

Por otro lado, las técnicas de sincronización basadas en elevar el nivel de interrupción para inhibir el tratamiento de determinadas interrupciones no son directamente aplicables ya que, aunque se impida que se ejecute dicha rutina de interrupción en el procesador donde se elevó el nivel, esta rutina podrá ejecutarse en cualquier otro procesador cuyo nivel de interrupción actual lo permita. Asimismo, la estrategia de impedir los cambios de contexto involuntarios durante una llamada al sistema no evita que se ejecuten concurrentemente llamadas al sistema en un multiprocesador. Las principales técnicas de sincronización utilizadas en multiprocesadores se basan en el mecanismo de *spin-lock* descrito en la sección 6.9.2 “Espera activa”.

En este esquema básico, con diversas variedades[7], se basa la sincronización en multiprocesadores. Nótese que, sin embargo, este mecanismo de sincronización carece de sentido en un monoprocesador: un proceso haciendo espera activa sobre un cerrojo consumirá su turno de ejecución sin poder obtenerlo, puesto que para ello debe ejecutar el

7 Además del modelo básico descrito, existen otras variedades. En Linux existen los *spin-locks* de lectura/escritura, que permiten que varios procesos puedan entrar simultáneamente en una sección crítica si sólo van a leer la estructura de datos protegida por el cerrojo, pero que sólo permiten entrar a un proceso en el caso de que vaya a modificarla. En Windows se proporcionan los *queued spin-locks*, que implementan el cerrojo optimizando las operaciones de coherencia de las memorias cache de los procesadores implicados y que permiten controlar cuál va a ser el próximo proceso/procesador que obtendrá el cerrojo.

proceso que lo posee. En muchos sistemas operativos, tanto en su versión para sistemas monoprocesador como en la destinada a multiprocesadores se mantienen los *spin-locks*. Sin embargo, en la versión para sistemas monoprocesador, las primitivas de gestión del cerrojo se definen como operaciones nulas[8]. Esto sólo afecta a la sincronización dentro del núcleo, por supuesto que para sincronizar acciones de distintos procesos no se puede recurrir a un *spin-lock* nulo; en estos casos es mejor recurrir a implementación basadas en *mutex* o semáforos que bloquean al proceso y no hacen espera activa.

Hay que resaltar que, mientras un proceso está ejecutando un bucle de espera activa sobre un cerrojo, el procesador involucrado no puede dedicarse a ejecutar otros procesos. Por tanto, no debería realizarse un cambio de contexto, ni voluntario ni involuntario, en el procesador que ejecuta el proceso que mantiene posesión del cerrojo, pues alargaría la espera activa de los procesadores en los que se están ejecutando procesos que intentan tomar posesión del mismo.

Sincronización con rutinas de interrupción

Antes de analizar cómo resolver este tipo de problema de sincronización, hay que recordar que en un sistema multiprocesador la ejecución de una rutina de interrupción puede convivir con la de cualquier otra rutina del sistema operativo, con independencia de qué nivel de prioridad esté asociado a cada una de ellas. Por tanto, el análisis de los problemas de sincronización se hace más complejo en este tipo de sistema. Asimismo, es conveniente reseñar que en un multiprocesador es necesario incluir mecanismos de sincronización en toda rutina afectada, a diferencia de lo que ocurre en un sistema monoprocesador donde, como se analizó previamente, sólo se incluye en la de menor prioridad.

La estrategia para resolver este tipo de problemas se basa en los *spin-locks*. Las rutinas afectadas usarán un cerrojo de este tipo para asegurarse de que no se ejecutan concurrentemente. Además, la rutina de menor prioridad, ya sea una rutina de interrupción o una llamada al sistema, debe elevar el nivel de interrupción en el procesador donde ejecuta para asegurarse de que no se producen interbloqueos: una rutina en posesión de un cerrojo es interrumpida por una rutina que también lo requiere, quedándose esta última indefinidamente en un bucle de espera que congela la ejecución del procesador (a partir de ese momento, lo único que podría ejecutar son rutinas de interrupción de mayor prioridad). Resumiendo, la técnica de sincronización consiste en lo siguiente: si hay N rutinas que deben sincronizar entre sí alguna parte de su ejecución, deberá definirse un *spin-lock* para ello y se establecerá que el nivel de interrupción que debe fijarse mientras se está usando el cerrojo corresponderá con el de la rutina de mayor prioridad. A continuación, se muestra una implementación hipotética de esta técnica.

```
/* establece el cerrojo */
int spin_lock_interrupcion(int cerrojo, int nivel) {
nivel_anterior = fijar_nivel_int(nivel);
/* espera hasta que el cerrojo valga 0 y escribe un 1 */
        while (TestAndSet(&cerrojo) == 1);
        return nivel_anterior;
}
/* libera el cerrojo */
spin_unlock_interrupcion(int &cerrojo, int nivel) {
        cerrojo = 0;
        fijar_nivel_int(nivel);
}
```

Aplicando este mecanismo al ejemplo de sincronización en el acceso a la cola de procesos listos para ejecutar planteado previamente, que requeriría elevar el nivel de interrupción del procesador al máximo ya que generalmente se accede desde todas las rutinas de interrupción, resultaría un esquema similar al siguiente:

```
int cerrojo_listos = 0;

unlock(mutex) {
    int nivel_previo;
    .....
    Si (procesos esperando) {
        Seleccionar un proceso P;
        nivel_previo = spin_lock_interrupcion(cerrojo_listos, NIVEL_MAXIMO);
        insertar_ultimo(lista_listos, P);
        spin_unlock_interrupcion(cerrojo_listos, nivel_previo);
    }
    .....
}

int_terminal {
```

8 Habitualmente, esto se consigue usando una compilación condicional dependiendo de si el sistema al que va destinado el sistema operativo es multiprocesador o no. En Windows se usa esta técnica con la mayor parte del sistema operativo, con excepción de los módulos *Ntdll.dll* y *Kernel32.dll* en los que se parchea directamente el código ejecutable para lograr que estas operaciones sean nulas en un sistema monoprocesador.

```
        int nivel_previo;
        .....
        Si (procesos esperando) {
            Seleccionar un proceso Q;
            nivel_previo = spin_lock_interrupcion(cerrojo_listos, NIVEL_MAXIMO);
            insertar_ultimo(lista_listos, Q);
            spin_unlock_interrupcion(cerrojo_listos, nivel_previo);
        }
        .....
    }
```

El esquema explicado se corresponde con el utilizado en Windows, donde cada cerrojo tiene asociado un nivel de interrupción. Sin embargo, en Linux, dado que no se implementa un esquema de prioridades, para resolver este tipo de esquemas de sincronización, se prohíben las interrupciones mientras se mantiene el cerrojo usando para ello las primitivas *spin_lock_irq* y *spin_unlock_irq*.

Obsérvese que en un sistema monoprocesador, dado que las operaciones del *spin-lock* se redefinen como nulas, queda como resultado la misma solución que se explicó para un sistema monoprocesador: inhibir la interrupción que pueda causar un conflicto.

Sincronización entre llamadas al sistema concurrentes

Nuevamente, en este caso la solución está basada en *spin-locks*, aunque, como se hizo en el estudio de este problema para un sistema monoprocesador, se va a realizar un análisis distinguiendo si se trata de un sistema con un núcleo expulsable o no.

En el caso de un núcleo no expulsable bastaría con que las llamadas al sistema que necesitan sincronizar ciertas partes de su ejecución usaran *spin-locks* para hacerlo. No sería necesario en este caso modificar el nivel de interrupción, por lo que se usarían directamente las primitivas de gestión de este tipo de cerrojos (en Linux *spin_lock* y *spin_unlock*). Dado que un sistema monoprocesador estas primitivas se convierten en operaciones nulas, no tendrán ningún efecto, lo cual no es sorprendente ya que, como se analizó anteriormente, para un núcleo no expulsable en un sistema monoprocesador no existe este problema al no permitirse la ejecución concurrente de llamadas.

Si se trata de un núcleo expulsable, además de usar *spin-locks*, es preciso asegurarse de que no se produce un cambio de contexto involuntario mientras se está en posesión de un cerrojo ya que, como se explicó previamente, podría producirse un interbloqueo si el proceso que entra a ejecutar intenta obtener ese mismo cerrojo. Por tanto, se usarán las primitivas de *spin-locks* que alteran el nivel o estado de las interrupciones (a las que en el ejemplo se las denominó *spin_lock_interrupcion* y *spin_unlock_interrupcion*), asociándoles como nivel de interrupción aquel que impide que se produzcan cambios de contexto involuntarios, es decir, el que inhibe las interrupciones *software*. Nuevamente, si se redefinen como nulas las operaciones directas sobre el cerrojo, el resultado es el mismo que un sistema monoprocesador, es decir, el problema se soluciona inhibiendo la interrupción *software* en los fragmentos conflictivos.

Como ocurría con los semáforos para los sistemas monoprocesador con núcleos expulsables, hay que establecer la granularidad del *spin-lock*. Se trata de la misma deliberación: intentar maximizar el paralelismo haciendo que la estructura de datos protegida por un *spin-lock* sea pequeña, pero suficientemente grande para que la sobrecarga por el uso de los distintos *spin-locks* implicados sea tolerable. Nótese que llevado a un extremo, se podría usar un único cerrojo para todo el sistema operativo. Esta era la solución usada en la versión 2.0 de Linux, que utilizaba un cerrojo global, denominado *kernel_flag*, que aseguraba que en cada momento sólo un único procesador ejecutara en modo sistema el código del sistema operativo. Se trataba de una solución de compromiso que impedía cualquier tipo de paralelismo en la ejecución del sistema operativo y que, evidentemente, se ha ido mejorando en las sucesivas versiones de Linux, "rompiendo" ese único cerrojo en múltiples cerrojos que controlan el acceso a las distintas estructuras del sistema operativo.

Sea cual sea el modelo de núcleo, surge en este punto la misma cuestión que apareció en el análisis de la sincronización entre llamadas para un sistema monoprocesador con un núcleo expulsable: ¿qué ocurre si la sección crítica es muy larga o requiere que el proceso pueda bloquearse durante la misma? Y la respuesta es la misma que en el análisis previo: se deberían implementar semáforos (o un mecanismo equivalente) para resolver estas deficiencias. En este caso, una sección crítica muy larga, más que afectar directamente al tiempo de respuesta de los procesos, causa una disminución del paralelismo del sistema, ya que puede haber uno o más procesadores haciendo espera activa mientras dura la sección crítica ejecutada en otro procesador.

Dadas las similitudes con el caso de un núcleo expulsable para un sistema monoprocesador, se pueden aplicar algunas de las consideraciones expuestas en el análisis realizado para ese caso:

- En cuanto a la implementación del semáforo, sus operaciones conseguirán atomicidad gracias al uso de los *spin-locks*, con las interrupciones *software* inhibidas en el caso de un núcleo expulsable.
- Con el semáforo la sección crítica se ejecuta con todas las interrupciones habilitadas y sin mantener la posesión de ningún cerrojo, proporcionando un tiempo de respuesta y un nivel de paralelismo adecuados, y permitiendo que haya bloqueos durante la sección crítica. En cualquier caso, si la sección crítica es muy breve, puede ser más adecuado usar directamente un *spin-lock* en vez de un semáforo, para evitar la sobrecarga asociada a las operaciones del semáforo.
- Nuevamente, hay que lograr un compromiso a la hora de fijar la granularidad del semáforo: la cantidad de información protegida por un semáforo debe de ser suficientemente pequeña para asegurar un nivel de paralelismo adecuado en la ejecución de llamadas al sistema, pero intentando limitar el número de semáfo-

ros que debe obtener el proceso durante una llamada al sistema para, de esta forma, acotar la sobrecarga debida a la sincronización.

- Usando la misma estrategia que se emplea para implementar los semáforos, se puede construir cualquier mecanismo de sincronización similar, tanto para uso interno como para exportar a las aplicaciones.

Para terminar, se presenta una última consideración sobre la sincronización en multiprocesadores. Del análisis realizado en esta sección, se puede apreciar que los sistemas operativos para monoprocesadores con un modelo de núcleo expulsable están mucho mejor preparados para afrontar el reto de adaptarse a sistemas multiprocesadores, dado que muchos de los desafíos que plantea la existencia de un paralelismo real, como, por ejemplo, la determinación de qué semáforos utilizar para proteger las distintas estructuras de datos del sistema operativo, ya se abordan en los sistemas expulsables.

6.9.5. Comunicación dentro del sistema operativo

Como se vio en el capítulo "2 Introducción a los sistemas operativos", un sistema operativo es un programa extenso y complejo que está compuesto, por una serie de componentes con funciones bien definidas. Un aspecto importante en el diseño de un sistema operativo es el mecanismo de comunicación empleado para comunicar los componentes que conforman el sistema operativo. Estos mecanismos dependen de la estructura del sistema operativo.

Comunicación en sistemas operativos monolíticos

En un sistema operativo monolítico todos sus componentes se encuentran integrados en un único programa (el sistema operativo) que se ejecuta en un único espacio de direcciones. Además, todas las funciones que ofrece se ejecutan en modo privilegiado. En este tipo de sistemas la comunicación se realiza mediante el uso de estructura de datos compartidas y llamadas a procedimientos. Un sistema operativo de este tipo no tiene por qué ser secuencial. Así por ejemplo, Linux es un sistema operativo con estructura monolítica pero incluye una serie de *threads* que se denominan *threads* de núcleo o *threads* de kernel. En este caso la comunicación entre los distintos *threads* debe asegurar una correcta sincronización, que se puede conseguir según lo expuesto en la sección anterior.

Comunicación en sistemas operativos con estructura cliente-servidor

En este tipo de modelo el enfoque consiste en implementar la mayor parte de los servicios y funciones del sistema operativo en procesos de usuario, dejando sólo una pequeña parte del sistema operativo ejecutando en modo núcleo. A esta parte se le denomina micronúcleo y a los procesos que ejecutan el resto de funciones se les denomina servidores. La comunicación entre los componentes de este tipo de sistema operativo se realiza mediante el paso de mensajes. Cada elemento del sistema operativo realiza operaciones directamente sobre datos locales y utiliza el paso de mensajes para la comunicación con otros elementos del sistema. Para este tipo de sistemas, la sincronización se simplifica debido a que el propio paso de mensajes introduce de forma implícita un cierto grado de sincronización.

Windows también sigue esta filosofía de diseño. Windows utiliza para la comunicación entre los componentes del sistema operativo un mecanismo conocido como **llamadas a procedimientos locales** (LPC, *local procedure call*). El objetivo de este mecanismo es disponer de un esquema de paso de mensajes de alta velocidad. Este mecanismo no se encuentra disponible a través de los servicios Windows, es un mecanismo exclusivo del sistema operativo.

Una LPC es un mecanismo que se utiliza entre un proceso servidor y uno o más procesos clientes. Las LPC permiten tres métodos de intercambio de mensajes:

- Un mensaje de menos de 256 bytes se puede enviar invocando a una LPC con un *buffer* que contiene el mensaje. El mensaje es copiado desde el espacio de direcciones del proceso que envía el mensaje al espacio de direcciones del sistema, y de aquí al espacio de direcciones de receptor del mensaje.
- Si un cliente y un servidor desean intercambiar más de 256 bytes de datos, pueden utilizar una sección de memoria compartida. El emisor almacena el mensaje en esta sección y envía un pequeño mensaje al receptor indicándole dónde encontrar los datos en la sección compartida.
- Cuando un cliente y un servidor desean comunicarse grandes cantidades de datos que no caben en la sección compartida, los datos pueden ser leídos o escritos directamente en el espacio de direcciones del cliente.

El mecanismo de LPC utilizado en Windows utiliza *objetos de tipo puerto* para la comunicación. Hay varios tipos de puertos posibles:

- *Puerto de conexión con un servidor*. Puerto con nombre utilizado por los clientes para realizar una conexión con un servidor.
- *Puerto de comunicación con un servidor*. Puerto sin nombre que utiliza un servidor para la comunicación con un cliente. El servidor mantiene un puerto activo por cliente.
- *Puerto de comunicación de cliente*. Puerto sin nombre que utiliza un cliente para la comunicación con un servidor.
- *Puerto de comunicación sin nombre*. Puerto creado por dos *threads* del mismo proceso.

Las LPC se utilizan normalmente de la siguiente forma: un servidor crea un puerto de conexión con nombre. A continuación un cliente pide una conexión a este puerto. Si el cliente obtiene acceso se crean dos puertos sin nom-

bre: uno para el cliente y otro para el servidor. El cliente y el servidor obtienen manejadores para sus puertos respectivos y finalmente el cliente y el servidor se comunican a través de estos puertos.

6.10. SERVICIOS UNIX

En esta sección se presentan algunos de los servicios que ofrece UNIX para los distintos mecanismos de comunicación y sincronización de procesos que se han ido presentando a lo largo del capítulo. Los servicios de sincronización pura incluyen los semáforos y los *mutex* y variables condicionales. Para comunicar procesos se pueden emplear tuberías y colas de mensajes.

6.10.1. Tuberías UNIX

En UNIX existen tuberías sin nombre o simplemente *pipes* y tuberías con nombre o FIFOS. Un *pipe* no tiene nombre y, por lo tanto, sólo puede ser utilizado entre los procesos que lo hereden a través de la llamada `fork`. La figura 6.34 muestra la jerarquía de procesos que pueden utilizar una misma tubería.

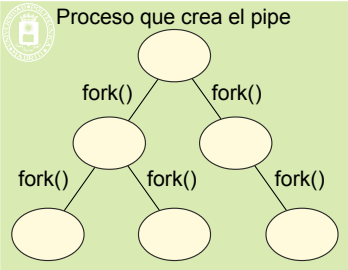


Figura 6.34 *Jerarquía de procesos que pueden compartir un mismo pipe en UNIX.*

Para leer y escribir de una tubería en UNIX, se utilizan descriptores de fichero. Las tuberías sin nombre tienen asociados dos descriptores de fichero. Uno de ellos se emplea para leer y el otro para escribir. Un FIFO puede ser abierto para escritura o para lectura por más de un proceso. A continuación se describen los servicios UNIX relacionados con las tuberías.

- int **pipe** (int fildes[2]);

Este servicio permite crear una tubería. Esta llamada devuelve dos descriptores de ficheros (véase la figura 6.35) que se utilizan como identificadores:

- `fildes[0]`, descriptor de fichero que se emplea para leer del *pipe*.
- `fildes[1]`, descriptor de fichero que se utiliza para escribir en el *pipe*.

La llamada `pipe` devuelve 0 si fue bien y -1 en caso de error.

- int **mkfifo** (char *fifo, mode_t mode);

Esta llamada permite crear una tubería con nombre, que en UNIX se conocen como FIFO. Los FIFO tienen un nombre local que lo identifican dentro de una misma máquina. El nombre que se utiliza corresponde con el de un fichero. Esta característica permite que los FIFO puedan utilizarse para comunicar y sincronizar procesos de la misma máquina, sin necesidad de que lo hereden por medio de la llamada `fork`.

El primer argumento representa el nombre del FIFO. El segundo argumento representa los permisos asociados al FIFO. La llamada devuelve 0 si se ejecutó con éxito o -1 en caso de error.

- int **open** (char *fifo, int flag);

Este servicio permite abrir una tubería con nombre. Este servicio también se emplea para abrir ficheros. El primer argumento identifica el nombre del FIFO que se quiere abrir y el segundo la forma en la que se va a acceder al FIFO. Los posibles valores de este segundo argumento son:

- `O_RDONLY`, se abre el FIFO para realizar sólo operaciones de lectura.
- `O_WRONLY`, se abre FIFO para realizar sólo operaciones de escritura.
- `O_NONBLOCK`.

El servicio `open` devuelve un descriptor de fichero que se puede utilizar para leer y escribir del FIFO, según el caso. En caso de error devuelve -1.

En el comportamiento por defecto la llamada `open` bloquea al proceso que la ejecuta hasta que haya algún otro proceso que abra el otro extremo del FIFO. Sin embargo, si se utiliza `O_NONBLOCK`, el proceso que abre para lectura (`O_NONBLOCK | O_RDONLY`) lo hace sin quedar bloqueado, mientras que el que lo abra para escritura fracasa con ENXIO si ningún proceso lo tiene abierto para lectura.

- int **close** (int fd);

Este servicio cierra un descriptor de fichero asociado a una tubería con o sin nombre. El argumento de `close` indica el descriptor de fichero que se desea cerrar. La llamada devuelve 0 si se ejecutó con éxito. En caso de error devuelve -1.

- int **unlink** (char *fifo);

Permite borrar un FIFO. Esta llamada también se emplea para borrar ficheros. Esta llamada pospone la destrucción del FIFO hasta que todos los procesos que lo estén utilizando lo hayan cerrado con la función `close`. En el caso de una tubería sin nombre, ésta se destruye cuando se cierra el último descriptor que tiene asociado.

- int **read** (int fd, char *buffer, int n);

Este servicio se utiliza para leer datos de un *pipe* o de un FIFO (véase la aclaración 6.3). El primer argumento indica el descriptor de lectura del *pipe*. El segundo argumento especifica el *buffer* de usuario donde se van a situar los datos leídos del *pipe*. El último argumento indica el número de bytes que se desean leer del *pipe*. La llamada devuelve el número de bytes leídos. En caso de error, la llamada devuelve -1.

Aclaración 6.3. La llamada `read` se emplea en UNIX también para leer datos de un fichero. De igual forma, en las escrituras, la llamada `write` se utiliza en UNIX para escribir datos en ficheros.

La semántica de una operación de lectura cuando se aplica sobre un *pipe* en UNIX es la que se indicó en la sección 6.6.4 “Tubería o pipe”. En UNIX si no hay escritores y el *pipe* está vacío, la llamada devuelve cero, indicando fin de fichero (en este caso la llamada no bloquea al proceso).

La lectura sobre un *pipe* en UNIX es atómica cuando el número de datos que se desean leer es menor que el tamaño del *pipe*.

- int **write** (int fd, char *buffer, int n);

Este servicio se utiliza para escribir datos en una tubería (véase la aclaración 6.3). El primer argumento representa el descriptor de fichero que se emplea para escribir en un *pipe*. El segundo argumento especifica el *buffer* de usuario donde se encuentran los datos que se van a escribir al *pipe*. El último argumento indica el número de bytes a escribir. Los datos se escriben en el *pipe* en orden FIFO.

La semántica del servicio `write` aplicado a un *pipe* es la que se vio en la sección 6.6.4 “Tubería o pipe”. En UNIX, cuando no hay lectores y se intenta escribir en una tubería, el sistema operativo envía la señal `SIGPIPE` al proceso.

Al igual que las lecturas, las escrituras sobre un *pipe* son atómicas. En general, esta atomicidad se asegura siempre que el número de datos involucrados en la operación sea menor que el tamaño del *pipe*.

6.10.2. Ejemplos con tuberías UNIX

A continuación se van a emplear las tuberías UNIX para resolver alguno de los modelos descritos en la sección 6.5 “Modelos de comunicación y sincronización”.

Secuencia de conexión con tuberías

La figura 6.35 muestra los pasos necesarios para establecer la conexión de dos procesos mediante *pipes*. Es de destacar que los procesos cierran los descriptores que no utilizan.

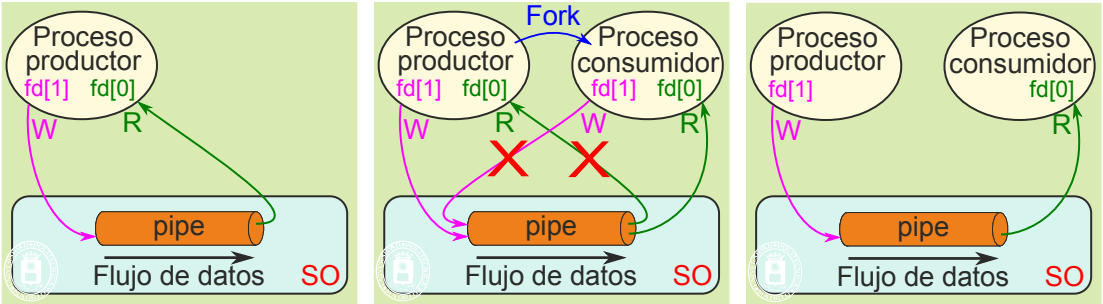


Figura 6.35 *Pasos para establecer una tubería UNIX entre dos procesos.*

Productor-consumidor con procesos y *pipe*

El programa 6.3 muestra un ejemplo de fragmento de código, que se puede utilizar para resolver problemas de tipo productor-consumidor mediante las tuberías que ofrece UNIX. En este ejemplo se crea un proceso hijo por medio de la llamada `fork`. A continuación el proceso hijo hará las veces de productor y el proceso padre de consumidor.

Programa 6.3 Productor-consumidor con procesos y *pipe* UNIX.

```
#include <stdio.h>
#include <unistd.h>
```

```
int main(void){
   int fd[2];              /* descriptores del pipe */
   int dato_p[4];          /* datos a producir */
   int dato_c;             /* dato a consumir */
   pipe(fd);
   if (fork() == 0) {      /* productor (proceso hijo)*/
      close (fd[0]);
      for(;;){
         /* producir dato_p */
         ......
         write(fd[1], dato_p, 4*sizeof(int));
      }
   } else {                /* consumidor (proceso padre)*/
      close (fd[1]);
      for(;;) {
         read(fd[0], &dato_c, sizeof(int));
         /* consumir dato */
      ......
      }
   }
return 0;
}
```

Productor-consumidor con procesos y FIFO

El ejemplo 6.4 incluye tres programas que realizan las funciones siguientes:

- El primero crea el FIFO.
- El segundo es el consumidor, que abre el FIFO exclusivamente para lectura. Seguidamente entra en un bucle que lee carácter a carácter del FIFO y lo escribe en la salida estándar.
- El tercero es el productor que abre el FIFO exclusivamente para escritura. Seguidamente entra en un bucle que lee carácter a carácter de la entrada estándar y lo manda al FIFO.

Programa 6.4 Productor-consumidor con procesos y FIFO UNIX.

Creación del FIFO

```
#define MYNAME "mk_FIFO"

#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
   if (mkfifo(argv[1], 0666) == 0)          /* Crea el FIFO */
      return 0;
   perror(MYNAME": mkfifo()");
   return 1;
}
```

Proceso consumidor

```
#define MYNAME "C_PPs_FIFO"

#include <ctype.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
   int fd;
   char ch;
   fd = open(argv[1], O_RDONLY);            /* Abre el FIFO para lectura*/
   if (fd == -1) {
```

```
        perror(MYNAME": open()");
        exit(1);
    }
    while(read(fd, &ch, 1) == 1) {
        ch = toupper(ch);
        write(1, &ch, 1);
    }
    return 0;
}
```

Proceso productor

```
#define MYNAME "P_PPs_FIFO"

#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
    int fd;
    char ch;
    fd = creat(argv[1], 0666);          /* Abre el FIFO para escritura*/
    if (fd == -1) {
        perror(MYNAME": creat()");
        return 1;
    }
    while(read(0, &ch, 1) == 1)
        write(fd, &ch, 1);
    return 0;
}
```

Sección crítica con tuberías

En esta sección se describe la forma de resolver el problema de la sección crítica utilizando tuberías de UNIX.

Uno de los procesos debe encargarse de crear la tubería e introducir el testigo en la misma. Como testigo se utilizará un simple carácter. El fragmento de código que se debe utilizar es el siguiente:

```
int fildes[2];                  /* tubería utilizada para sincronizar */
char testigo;                   /* se declara un carácter como testigo */

pipe(fildes);                   /* se crea la tubería */
write(fildes[1], &testigo, 1);  /* se inserta el testigo en la tubería */
```

Una vez creada la tubería e insertado el testigo en ella, los procesos deben proteger el código correspondiente a la sección crítica de la siguiente forma:

```
read(fildes[0], &testigo, 1);
< código correspondiente a la sección crítica >
write(fildes[1], &testigo, 1);
```

La operación `read` eliminará el testigo de la tubería y la operación `write` lo insertará de nuevo en ella.

Ejecución de secuencias de mandatos

Aunque en las secciones anteriores se han presentado dos posibles utilizaciones de las tuberías, su uso más extendido se encuentra en la ejecución de secuencias de mandatos. A continuación se presenta un programa que permite ejecutar el mandato `ls | wc`. La ejecución de este mandato supone la ejecución de los programas `ls` y `wc` de UNIX y su conexión mediante una tubería. El código que permite la ejecución de este mandato es el que se muestra en el programa 6.5.

Programa 6.5 Programa que ejecuta `ls | wc`.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main(void) {
    int fd[2];
```

```
    pid_t pid;

    /* se crea la tubería */
    if (pipe(fd) < 0) {
        perror("Error al crear la tubería");
        return 1;
    }

    pid = fork();
    switch (pid) {
        case -1:            /* error */
            perror("Error en el fork");
            return 1;
        case 0:             /* proceso hijo ejecuta ls */
            close(fd[0]);
            close(STDOUT_FILENO);
            dup(fd[1]);
            close(fd[1]);
            execlp("ls", "ls", NULL);
            perror("Error en el exec");
            return 1;
            break;
        default:        /* proceeso padre ejecuta wc */
            close(fd[1]);
            close(STDIN_FILENO);
            dup(fd[0]);
            close(fd[0]);
            execlp("wc", "wc", NULL);
            perror("Error en el exec");
            return 1;
    }
            return 0;
}
```

El proceso hijo (véase la figura 6.36) redirige su salida estándar a la tubería (véase la aclaración 6.4). Por su parte el proceso padre redirecciona su entrada estándar a la tubería. Con esto se consigue que el proceso que ejecuta el programa `ls` escriba sus datos de salida en la tubería, y el proceso que ejecuta el programa `wc` lea sus datos de la tubería.

Aclaración 6.4 En UNIX, cada proceso dispone al comenzar su ejecución de tres descriptores de fichero en uso: la entrada, la salida y la salida de errores. La entrada suele estar asociada al teclado y la salida y salida de errores estándar se encuentra asociado normalmente a la pantalla. Estos descriptores de fichero se corresponden en UNIX con los descriptores con número 0, 1 y 2 respectivamente, que tienen asociados como constantes simbólicas los valores `STDIN_FILENO`, `STDOUT_FILENO` y `STDERR_FILENO`.

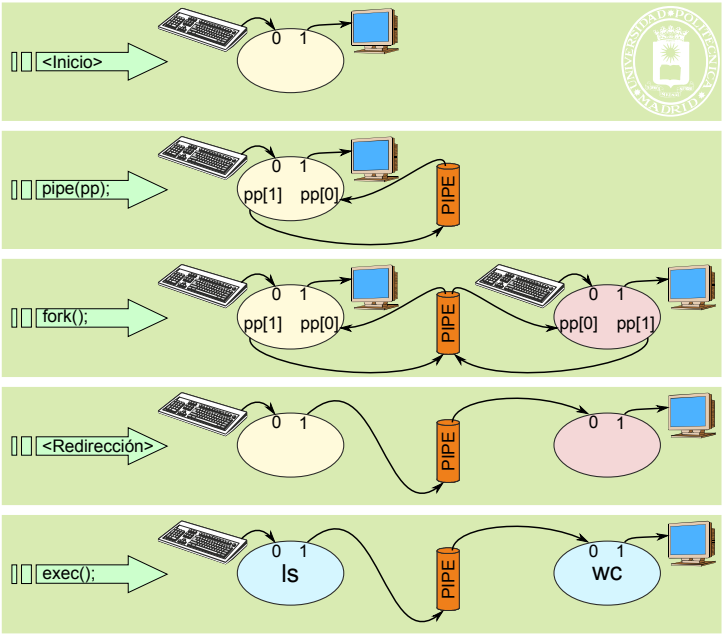


Figura 6.36 *Ejecución de mandatos con tuberías en UNIX.*

Los pasos que realiza el proceso hijo para redirigir su salida estándar a la tubería son los siguientes:

- Cierra el descriptor de lectura de la tubería, `fd[0]`, ya que no lo utiliza `(close(fd[0])`.
- Cierra la salida estándar, que inicialmente en un proceso referencia el terminal (`close(STDOUT_FILENO)`). Esta operación libera el descriptor de fichero 1, es decir, el descriptor `STDOUT_FILENO`.
- Duplica el descriptor de escritura de la tubería mediante la sentencia `dup(fd[1])` (véase la aclaración 6.5). Esta llamada devuelve un descriptor, que será el más bajo disponible, en este caso el descriptor 1, que es la salida estándar. Con esta operación se consigue que el descriptor de fichero 1 y el descriptor almacenado en `fd[1]` sirvan para escribir datos en la tubería. De esta forma se ha conseguido redirigir el descriptor de salida estándar en el proceso hijo a la tubería.
- Se cierra el descriptor `fd[1]`, ya que el proceso hijo no lo va a utilizar en adelante. Recuérdese que el descriptor 1 sigue siendo válido para escribir datos en la tubería.
- Cuando el proceso hijo invoca el servicio `exec` para ejecutar un nuevo programa, se conserva en el BCP la tabla de descriptores de ficheros abiertos, y en este caso el descriptor de salida estándar 1 está referenciando a la tubería. Cuando el proceso comienza a ejecutar el código del programa `ls`, todas las escrituras que se hagan sobre el descriptor de salida estándar se harán realmente sobre la tubería.

Aclaración 6.5. El servicio `dup` de UNIX duplica un descriptor de fichero abierto. Su prototipo es:

```
int dup(int fd);
```

El servicio `dup` duplica el descriptor de fichero `fd`. La llamada devuelve el nuevo descriptor de fichero. Este descriptor de fichero referencia al mismo fichero al que referencia `fd`.

6.10.3. Sockets

En esta sección se incluyen los servicios para *sockets*. Como se ha indicado anteriormente, los *sockets* de Windows siguen la misma especificación, por lo que no se incluye una sección para ellos.

int socket (int dominio, int tipo, int protocolo);

Este servicio crea un *socket* del dominio, tipo y protocolo dados, pero sin dirección. Esta dirección se asocia con el servicio `bind`.

Los argumentos se han descrito en la sección: “6.6.5 Sockets. Comunicación remota”, página 318.

El servicio **devuelve** un descriptor de fichero con el que se accede al *socket*. Si fracasa devuelve -1.

int bind (int sd, struct sockaddr *dir, int tam);

Este servicio asocia una dirección local al *socket*.

Argumentos:

- `sd`: descriptor de fichero del *socket*.
- `*dir`: Dirección local. La dirección debe suministrarse en la estructura `sockaddr` particular para el dominio del *socket*, según se ha descrito en la sección: “6.6.5 Sockets. Comunicación remota”, página 318. Consideraciones importantes:
  - La estructura `sockaddr` debe ponerse a cero antes de usarse, lo que se puede hacer con la función `bzero()`.
  - Para el dominio AF_INET, si se rellena el campo sin_port a cero, se deja libertad al sistema operativo para que asigne un puerto libre. Esto suele hacerse en el cliente. Sin embargo, el servidor usa generalmente un puerto bien conocido, por lo que hay que especificar su valor.

Devuelve 0 en caso de éxito y -1 si se produce un error.

int connect (int sd, struct sockaddr *dir, int tam);

Este servicio lo ejecuta el proceso cliente con *socket* basado en conexión para solicitar una conexión. Se asocia una dirección remota (la del servidor) al *socket* del cliente.

Argumentos:

- `sd`: descriptor de fichero del *socket* basado en conexión.
- `*dir`: dirección remota del servidor que se quiere acceder. Aplican los comentarios del servicio anterior.
- `tam`: Tamaño de la estructura `sockaddr`.

Devuelve 0 en caso de éxito y -1 si se produce un error.

int listen (int sd, int baklog);

Este servicio lo utiliza el proceso servidor con *socket* basado en conexión para recibir peticiones de conexiones por parte de los clientes. Permite indicar el número máximo de peticiones encoladas, mediante `baklog`.

Argumentos:

- `sd`: descriptor de fichero del *socket* basado en conexión.
- `baklog`: Número de peticiones pendientes de ser atendidas que puede encolar el *socket*.

- **Devuelve** 0 en caso de éxito y -1 si se produce un error.

◙ int accept (int sd, struct sockaddr *dir, int *tam);

Este servicio lo utiliza el proceso servidor con *socket* basado en conexión para aceptar de una petición de conexión. El servicio toma una solicitud de la cola del `listen` y **crea un nuevo *socket*** conectado. El servicio bloquea al proceso en caso de que no exista ninguna petición de conexión.
Argumentos:

- `sd`: descriptor de fichero del *socket*.
- `*dir`: Contiene la dirección del *socket* cliente.
- `*tam`: Argumento de entrada y salida con el tamaño de la estructura `sockaddr` del *socket* del cliente. Devuelve el tamaño real de la estructura `sockaddr` del *socket* del cliente.

El servicio **devuelve** un descriptor de fichero con el que se accede al nuevo *socket* creado. Si fracasa devuelve -1.

◙ int send (int sd, char *mem, int tam, int flags);
◙ int recv (int sd, char *mem, int tam, int flags);

Estos dos servicios permiten la transmisión de mensajes con *sockets* conectados son equivalentes al read y write que se han visto en el capítulo "5 E/S y Sistema de ficheros", con las siguientes diferencias:

- En lectura, si no hay datos el proceso queda bloqueado a la espera de que lleguen. Sin embargo, se puede cambiar el comportamiento del *socket* para que funcione de forma no bloqueante.
- En escritura, si no se puede mandar el mensaje de forma atómica, no se manda, devolviendo el error EMSGSIZE.
- El argumento `flags` permite establecer determinadas opciones de comunicación. Si se pone a cero los servicios son equivalentes al read y write.

Estos servicios **devuelven** el número de bytes enviados o recibidos, o -1 en caso de fracaso.

◙ int sendto (int sd, char *mem, int tam, int flags, struct sockaddr *dir, int len);
◙ int recvfrom (int sd, char *mem, int tam, int flags, struct sockaddr *dir, int *len);

Estos dos servicios sirven para transmisión mensajes con *sockets* NO conectados. El argumento `*dir` del servicio `sendto` tiene la dirección del *socket* destino y `len` el tamaño de la estructura `sockaddr` usada. En el servicio `recvfrom`, el argumento `*dir` tiene la dirección del remitente del mensaje.

6.10.4. Ejemplos con *sockets*

Ejemplo de comunicación con *sockets* de tipo *datagrama*

En este ejemplo se plantea un servidor de eco que utiliza el puerto UDP número 7 (puerto normal del servidor de eco).

Como suele ser normal con los servidores, éste está arrancado de forma permanente. En su arranque ejecuta los tres servicios *socket*, `bind` y `recvfrom`, creando un *socket* de tipo *datagrama* y con puerto número 7. Seguidamente, se queda bloqueado en el `recvfrom`, hasta que un cliente mande un mensaje.

El cliente debe crear un *socket* de tipo *datagrama* al que se le asigna automáticamente un puerto local libre.

El diálogo entre cliente y servidor consiste en la ejecución de una serie de parejas de servicios `sendto` y `recvfrom`, para transmitir alternadamente mensajes del cliente al servidor y del servidor al cliente.

El cliente cerrará su *socket* cuanto termine de utilizar el servicio, mientras que el servidor solamente cerrará su *socket* si se cierra el servicio.

La tabla 6.2 muestra las secuencias de servicios involucrados, mientras que la figura 6.37 muestra también la relación entre cliente y servidor.

Tabla 6.2 *Servicios utilizados para establecer una comunicación con sockets datagrama* y su correspondencia con el servicio postal.

Cliente **(Inicia la conversación)**	**Servidor SIN estado** **(Recibe mensajes y los atiende)**
socket ■ Adquirir buzón UDP **bind** ■ Asignarle una dirección libre **sendto** ■ Enviar carta con remite: Quiero X **recvfrom** ■ Recoger respuesta **close** ■ Eliminar buzón	**socket** ■ Adquirir buzón UDP **bind** ■ Asignarle dirección bien conocida **recvfrom** ■ Recoger carta ■ Tomar dirección del remitente **sendto** ■ Enviar al remitente: Toma X **close** ■ Eventualmente, eliminar buzó

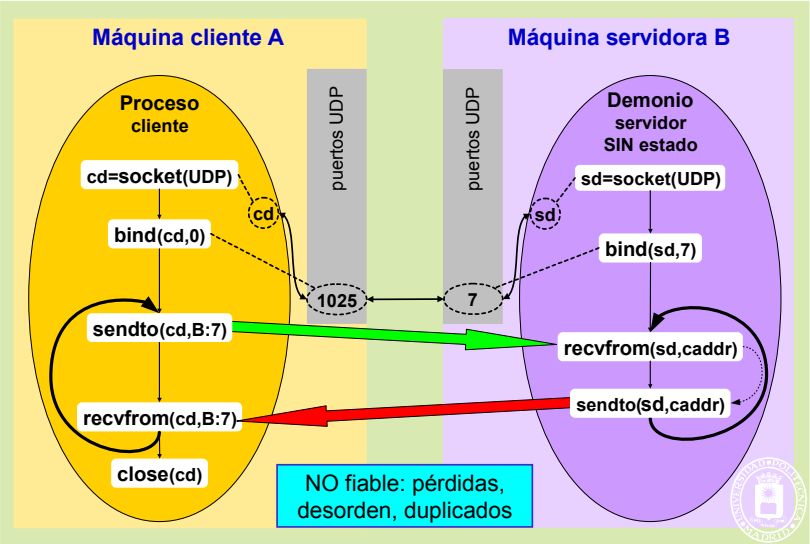


Figura 6.37 *Servicios utilizados para establecer una comunicación con sockets datagrama.*

El programa 6.6 presenta un ejemplo de cliente servidor basado en *sockets* AF_INET *datagrama*. En el código del cliente es de destacar que el `bind` no es necesario, se haría implícitamente al ejecutar el `sendto`. También es de destacar que si se pierde un carácter, tanto en el envío al servidor como en su respuesta el cliente se queda "colgado", bloqueado indefinidamente, esperando una respuesta que no nunca llegará.

Programa 6.6 Código de un cliente y un servidor de eco basado en *sockets* AF_INET *datagrama*.

```
/* Código del cliente de eco sin conexión */

   #define MYNAME "UDP_c"

   #include <arpa/inet.h>          /* inet  * */
   #include <netinet/in.h>         /* IP*, sockaddr  in, ntoh?, hton?, etc. */
   #include <netinet/tcp.h>        /* TCPOPT_*, etc. */
   #include <sys/socket.h>         /* socket */
   #include <sys/types.h>          /* socket */
   #include <sys/un.h>             /* sockaddr_un */
   #include <netdb.h>              /* gethostbyname */

   #include <stdio.h>
   #include <unistd.h>

   int main(int argc, char * argv[])
   {
      int cd, size ;
      struct hostent * hp;
      struct sockaddr_in s_ain, c_ain;
```

```
    unsigned char byte;

    hp = gethostbyname(argv[1]);      /* por ejemplo, otilio.fi.upm.es */
    bzero((char *)&s_ain, sizeof(s_ain));
    s_ain.sin_family = AF_INET;
    memcpy (&(s_ain.sin_addr), hp->h_addr, hp->h_length);
    s_ain.sin_port = htons(7);        /* echo port */

    cd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
    bzero((char *)&c_ain, sizeof(c_ain));
    c_ain.sin_family = AF_INET;
    bind(cd, (struct sockaddr *)&c_ain, sizeof(s_ain));
    size = sizeof(c_ain);
    while(read( 0, &byte, 1) == 1) {
        sendto(cd, &byte, 1, 0, (struct sockaddr *)&s_ain, size);
        recvfrom(cd, &byte, 1, 0, (struct sockaddr *)&s_ain, &size);
        write(1, &byte, 1);
    }
    close(cd);
    return 0;
}
```

O se hace el `bind` explícito o el `sendto` lo hará implícitamente

Si se pierde un byte se "cuelga" el cliente

```
/* Código del servidor de eco sin conexión */

#define MYNAME "UDP_s"

#include <arpa/inet.h>          /* inet_* */
#include <netinet/in.h>         /* IP*, sockaddr_in, ntoh?, hton?, etc. */
#include <netinet/tcp.h>        /* TCPOPT_*, etc. */
#include <sys/socket.h>         /* socket */
#include <sys/types.h>          /* socket */
#include <sys/un.h>             /* sockaddr_un */
#include <netdb.h>              /* gethostbyname */

#include <stdio.h>
#include <unistd.h>

int main(void)
{
    int sd, size;
    unsigned char byte;
    struct sockaddr_in s_ain, c_ain;

    sd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

    bzero((char *)&s_ain, sizeof(s_ain));
    s_ain.sin_family = AF_INET;
    s_ain.sin_addr.s_addr = INADDR_ANY;     /*Cualquier origen*/
    s_ain.sin_port = htons(7);              /* echo server */
    bind(sd, (struct sockaddr *)&s_ain, sizeof(s_ain));
    size = sizeof(c_ain);

    while (1) {
        recvfrom(sd, &byte, 1, 0, (struct sockaddr *)&c_ain, &size);
        sendto(sd, &byte, 1, 0, (struct sockaddr *)&c_ain, size);
    }
}
```

Ejemplo de servidor concurrente con *sockets* de tipo *stream*

La solución que se presenta en este ejemplo es la solución genérica para la que están diseñados los sockets de tipo *stream*. Se observará que, en esta solución el servidor, una vez aceptada una conexión, genera un proceso hijo que se dedica a atender de forma exclusiva a ese cliente. De esta forma, el servidor queda rápidamente libre y puede atender otra solicitud de conexión. Por tanto, cada cliente tendrá su propio proceso servidor particular para atenderle. Todos los servidores dedicados trabajan de forma concurrente. Observe que el esquema es similar al del distribuidor de trabajo planteado para *threads* en la figura 3.26, página 92.

La tabla 6.3 muestra las secuencias de servicios involucrados, mientras que la figura 6.38 muestra también la relación entre cliente y servidor.

Tabla 6.3 *Servicios utilizados para establecer una comunicación con sockets stream* y su correspondencia con una llamada telefónica.

Cliente **(Inicia la conversación)**	**Servidor (centralita)** **(Recibe llamada y redirige)**
socket ■ Adquirir teléfono **bind** ■ Contratar línea (nº de teléfono) **connect** ■ Descolgar ■ Marcar ■ Esperar establecimiento llamada	**socket** ■ Adquirir centralita **bind** ■ Contratar línea (nº de teléfono) **listen** ■ Dimensionar centralita **accept** ■ Esperar establecimiento llamada
send & **recv** ■ Saludo: ¿Hola? **send** & **recv** ■ Diálogo: Quisiera... ¡Gracias! **close** ■ Colgar	■ Redirigir a teléfono de servicio **Servidor dedicado** **(Atiende la conversación)** **recv** & **send** ■ Saludo: ¿Qué desea? **recv** & send ■ Diálogo: ¡Cómo no! Aquí tiene **close** ■ Colgar

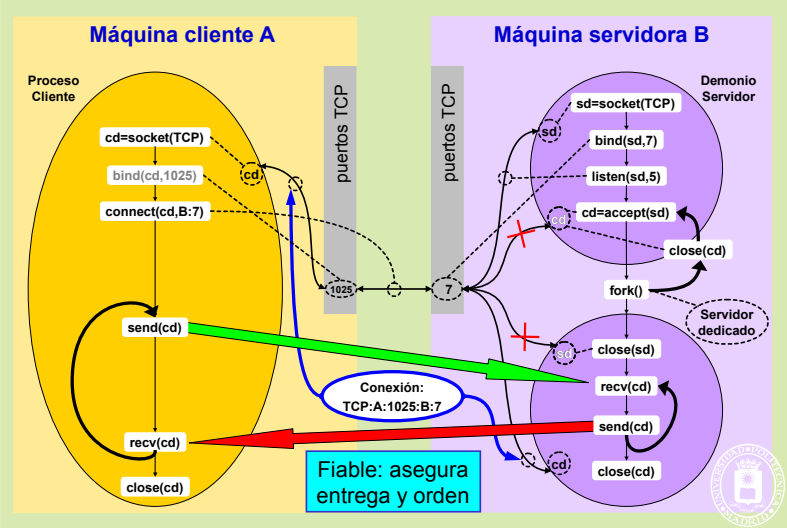


Figura 6.38 *Utilización de sockets con conexión para establecer un servidor hijo, dedicado al cliente que solicita la conexión.*

El programa 6.7 presenta un ejemplo de cliente servidor de eco basado en *sockets* AF_INET *stream*. Es de destacar la llamada al servicio `fork` para producir el servidor dedicado para cliente que solicita la conexión. En este caso no, la conexión establecida es fiable, por lo que se pierden mensajes, por lo que el cliente no puede quedar "colgado".

Programa 6.7 Código de un servidor concurrente un cliente de eco basado en *sockets* AF_INET *stream*.

```
/* Código del cliente de eco con conexión */

int main(int argc, char * argv[])
{
    int cd;
    struct hostent * hp;
    struct sockaddr  in s_ain;
    unsigned char byte;
```

```
    hp = gethostbyname(argv[1]);
    bzero((char *)&s_ain, sizeof(s_ain));
    s_ain.sin_family = AF_INET;
    memcpy (&(s_ain.sin_addr), hp->h_addr, hp->h_length); /* IP */
    s_ain.sin_port = htons(7);              /* echo port */

    cd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    connect(cd, (struct sockaddr *)&s_ain, sizeof(s_ain));

    while(read( 0, &byte, 1) == 1) {
            send(cd, &byte, 1, 0);          /* Bloqueante */
            recv(cd, &byte, 1, 0);          /* Bloqueante */
            write(1, &byte, 1);
    }
    close(cd);
    return 0;
}

/* Código del servidor de eco concurrente con conexión*/

int main(void)
{
    int sd, cd, size;
    unsigned char byte;
    struct sockaddr_in s_ain, c_ain;

    sd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

    bzero((char *)&s_ain, sizeof(s_ain));
    s_ain.sin_family = AF_INET;
    s_ain.sin_addr.s_addr = INADDR_ANY; /*Cualquier origen*/
    s_ain.sin_port = htons(7);                  /* echo port */

    bind(sd, (struct sockaddr *)&s_ain, sizeof(s_ain));

    listen(sd, 5);                              /* 5 = tamaño cola */
    while(1) {
        size = sizeof(c_ain);
        cd = accept(sd, (struct sockaddr *)&c_ain, &size);
        switch(fork()) {
        case -1:
            perror("echo server");
            return 1;
        case 0:
            close(sd);
            while(recv(cd, &byte, 1, 0) == 1)   /*Bloqueante*/
                send(cd, &byte, 1, 0);          /*Bloqueante*/
            close(cd);
            return 0;
        default:
            close(cd);
        } /* switch */
    } /* while */
} /* main */
```

Ejemplo de servidor secuencial con *sockets* de tipo *stream*

Aunque por lo general los *servidores* conviene construirlos con arquitectura concurrente, en algunos casos puede no interesar. La figura 6.39 muestra la secuencia de servicios involucrados en este caso.

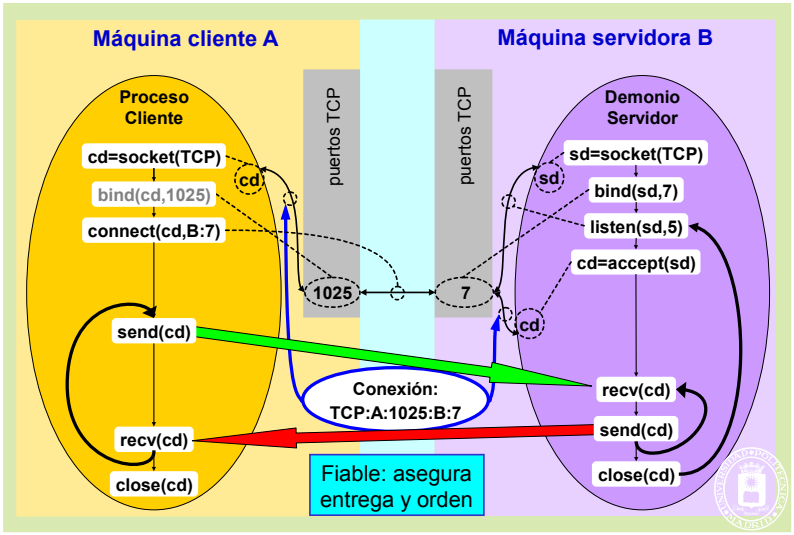


Figura 6.39 *Servicios utilizados en el caso de construir un servidor secuencial.*

6.10.5. Semáforos UNIX

En UNIX un semáforo se declara mediante una variable del tipo `sem_t`. El estándar UNIX define dos tipos de semáforos:

- **Semáforos sin nombre**. Permiten sincronizar los *threads* que se ejecutan dentro de un mismo proceso, o los procesos que lo heredan a través de la llamada `fork`.
- **Semáforos con nombre**. En este caso el semáforo lleva asociado un nombre que sigue la convención de nombrado que se emplea para ficheros. Con este tipo de semáforos se pueden sincronizar procesos sin necesidad de que tengan que heredar el semáforo utilizando la llamada `fork`.

La diferencia que existe entre los semáforos con nombre y sin nombre es similar a la que existe entre las tuberías sin nombre y los FIFOS.

Los servicios UNIX para manejar semáforos son los siguientes:

◙ int **sem_init** (sem_t *sem, int shared, int val);

Todos los semáforos en UNIX deben iniciarse antes de su uso. La función `sem_init` permite iniciar un semáforo sin nombre. Con este servicio se crea y se asigna un valor inicial a un semáforo sin nombre. El primer argumento identifica la variable de tipo semáforo que se quiere utilizar. El segundo argumento indica si el semáforo se puede utilizar para sincronizar *threads* o cualquier otro tipo de proceso. Si `shared` es 0, el semáforo sólo puede utilizarse entre los *threads* creados dentro del proceso que inicia el semáforo. Si `shared` es distinto de 0, entonces se puede utilizar para sincronizar procesos que lo hereden por medio de la llamada `fork`. El tercer argumento representa el valor que se asigna inicialmente al semáforo.

◙ int **sem_destroy** (sem_t *sem)

Con este servicio se destruye un semáforo sin nombre previamente creado con la llamada `sem_init`.

◙ sem_t ***sem_open** (char *name, int flag, mode_t mode, int val);
sem_t ***sem_open**(char *name, int flag);

El servicio `sem_open` permite crear o abrir un semáforo con nombre. La función que se utiliza para invocar este servicio admite dos modalidades según se utilice para crear el semáforo o simplemente abrir uno existente.

Un semáforo con nombre posee un nombre, un dueño y derechos de acceso similares a los de un fichero. El nombre de un semáforo es una cadena de caracteres que sigue la convención de nombrado de un fichero. La función `sem_open` establece una conexión entre un semáforo con nombre y una variable de tipo semáforo.

El valor del segundo argumento determina si la función `sem_open` accede a un semáforo previamente creado o si crea un nuevo. Un valor 0 en `flag` indica que se quiere utilizar un semáforo que ya ha sido creado, en este caso no es necesario los dos últimos parámetros de la función `sem_open`. Si `flag` tiene un valor `O_CREAT`, requiere los dos últimos argumentos de la función. El tercer parámetro especifica los permisos del semáforo que se va a crear, de la misma forma que ocurre en la llamada `open` para ficheros. El cuarto parámetro especifica el valor inicial del semáforo.

UNIX no requiere que los semáforos con nombre se correspondan con entradas de directorio en el sistema de ficheros, aunque sí pueden aparecer.

◙ int **sem_close** (sem_t *sem);

Cierra un semáforo con nombre, rompiendo la asociación que tenía un proceso con un semáforo.

◙ int **sem_unlink** (char *name);

Elimina del sistema un semáforo con nombre. Esta llamada pospone la destrucción del semáforo hasta que todos los procesos que lo estén utilizando lo hayan cerrado con la función `sem_close`.

- int **sem_wait** (sem_t *sem);

Permite hacer un `wait` sobre un semáfor.

- int **sem_post** (sem_t *sem);

Este servicio se corresponde con la operación `signal` sobre un semáforo. Todas las funciones que se han descrito devuelven un valor 0 si la función se ha ejecutado con éxito o -1 en caso de error. En este caso se almacena en la variable `errno` el código que identifica el error.

6.10.6. Ejemplos con semáforos

Estructura de productor-consumidor con semáforos y *threads*

Una posible forma de solucionar el problema del productor-consumidor es utilizar un almacén o *buffer* circular compartido por ambos procesos. El proceso productor fabrica un determinado dato y lo inserta en un *buffer* (véase la figura 6.40). El proceso consumidor retira del *buffer* los elementos insertados por el productor.

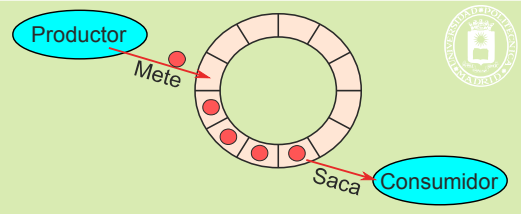


Figura 6.40 *Produtor-consumidor con buffer circular.*

A continuación se presenta una solución al problema del productor-consumidor utilizando *threads* y semáforos. El *buffer* utilizado para esta solución se trata como una cola circular. Cuando el *buffer* tiene un tamaño limitado, como suele ser lo habitual, se dice que el problema es de tipo productor-consumidor con *buffer* circular y acotado.

En este problema existen dos tipos de recursos: los elementos situados en el *buffer* y los huecos donde situar nuevos elementos. Cada uno de estos recursos se representa mediante un semáforo. Cuando un *thread* necesita un recurso de un cierto tipo, resta uno al valor del semáforo correspondiente mediante una operación `wait`. Cuando el proceso libera el recurso, se incrementa el valor del semáforo adecuado mediante la operación `post`.

La **información de control** que se utiliza son un contador de huecos disponibles `n_huecos`, un contador de datos disponibles `n_datos`, el número del último dato generado `ip` y el número del último de dato consumido `ic`.

Los semáforos utilizados para representar estos dos recursos se denominarán `n_huecos` y `n_datos`. Los valores iniciales de estos dos semáforos coincidirán con el número de recursos que estén disponibles inicialmente, que son tamaño del BUFF_SIZE y 0 respectivamente.

El programa 6.8 presenta la estructura que deberían tener los *threads* que hagan los papeles de productor y consumidor. Es de destacar que **solamente se protege el acceso al *buffer***, no la producción ni el consumo del dato. Es de destacar que no es necesario proteger ninguna variable de control, puesto que `ip` e `ic` no se comparten y los contadores `n_huecos` y `n_datos` las protegen sus correspondientes semáforos.

Programa 6.8 Estructura de los procesos productor y consumidor utilizando semáforos y *threads*. Se supone en este ejemplo que cada dato es un simple byte. Se marcan en verde las secciones protegidas que, en este caso, son críticas necesarias.

```
#define BUFF_SIZE          1024
#define TOTAL_DATOS        100000
sem_t n_datos;                          /* datos en buffer compartido */
sem_t n_huecos;                         /* huecos en buffer compartido */

int buffer[BUFF_SIZE];                  /* buffer circular acotado compartido */

int main(void)
{
   pthread_t th1, th2;
   sem_init(&n_datos, 0, 0);            /* Situación inicial */
   sem_init(&n_huecos, 0, BUFF_SIZE);
   pthread_create(&th1, NULL, Productor, NULL);  /*Arranque threads*/
   pthread_create(&th2, NULL, Consumidor, NULL);
   pthread_join(th1, NULL);             /* Esperar terminación */
   pthread_join(th2, NULL);
   sem_destroy(&n_datos);               /* Destruir semáforos */
```

No compartido: son threads

```
    sem_destroy(&n_huecos);
    return 0;                          /* return en el main equivale a exit */
}

void Productor(void)
{
    int ip, dato;
    for(ip=0; ip < TOTAL_DATOS; ip++) {
        <<Producir el dato>>
        sem_wait(&n_huecos);           /* Un hueco menos */
        buffer[ip % BUFF_SIZE] = dato; /* METER */
        sem_post(&n_datos);            /* Un dato más */
    }
}

void Consumidor(void)
{
    int ic, dato;
    for(ic=0; ic < TOTAL_DATOS; ic++) {
        sem_wait(&n_datos);            /* Un dato menos */
        dato = buffer[ic % BUFF_SIZE]; /* SACAR */
        sem_post(&n_huecos);           /* Un hueco más */
        <<Consumir el dato>>
    }
}
```

El semáforo `n_huecos` representa el número de ranuras libres que hay en el *buffer*, y el semáforo `n_datos` el número de datos introducidos en el *buffer* por el productor que aún no han sido retirados por el consumidor. Cuando el productor desea introducir un nuevo elemento en el *buffer*, resta 1 al valor del semáforo `n_huecos` (operación `wait`). Si el valor se hace negativo, el proceso se bloquea, ya que no hay nuevos huecos donde insertar n_datos. Cuando el productor ha insertado un nuevo dato en el *buffer*, incrementa el valor del semáforo `n_datos` (operación `post`).

Por su parte, el proceso consumidor, antes de eliminar del *buffer* un elemento, resta 1 al valor del semáforo `n_datos` (operación `wait`). Si el valor se hace negativo el proceso se bloquea. Cuando elimina del *buffer* un elemento incrementa el valor del semáforo `n_huecos` (operación `post`).

La correcta sincronización entre los dos procesos queda asegurada ya que cuando el proceso consumidor se bloquea en la operación `wait(n_datos)` se despertará cuando el proceso consumidor inserte un nuevo elemento en el *buffer* e incremente el valor de dicho semáforo con la operación `post(n_datos)`. De igual manera, el proceso productor se bloquea cuando el *buffer* está vacío y se desbloquea cuando el proceso consumidor extrae un elemento e incrementa el valor del semáforo `n_huecos`.

Esquema de lectores-escritores con semáforos y *threads*

En esta sección se presenta una posible solución al problema de los lectores escritores empleando semáforos. La estructura de los procesos lectores y escritores se muestra en el programa 6.9.

Programa 6.9 Estructura de los procesos lectores y escritores utilizando semáforos. Se han marcado en verde las secciones protegidas del programa y en rojo las secciones críticas auxiliares para proteger la variable de control `n_lectores`.

```
sem_t en_exclusiva;                    /* Acceso a dato */
sem_t de_lectura;                      /* Acceso a n_lectores */
int n_lectores = 0;                    /* Número de lectores */
int main(void)
{
    pthread_t th1, th2, th3, th4;
    sem_init(&en_exclusiva, 0, 1);             /* Situación inicial */
    sem_init(&de_lectura, 0, 1);
    pthread_create(&th1, NULL, Lector, NULL);  /* Arranque */
    pthread_create(&th2, NULL, Escritor, NULL);
    pthread_create(&th3, NULL, Lector, NULL);
    pthread_create(&th4, NULL, Escritor, NULL);
    pthread_join(th1, NULL);                   /* Esperar terminación */
    pthread_join(th2, NULL);
    pthread_join(th3, NULL);
```

```
    pthread_join(th4, NULL);
    sem_destroy(&de_lectura);                    /* Destruir */
    sem_destroy(&en_exclusiva);
    return 0;
}

void Lector(void)
{
    sem_wait(&de_lectura);
    n_lectores = n_lectores + 1;
    if (n_lectores == 1)
        sem_wait(&en_exclusiva);
    sem_post(&de_lectura);
    <<Lecturas simultáneas del recurso compartido>>
    sem_wait(&de_lectura);
    n_lectores = n_lectores - 1;
    if (n_lectores == 0)
        sem_post(&en_exclusiva);
    sem_post(&de_lectura);

    pthread_exit(0);
}

void Escritor(void)
{
    sem_wait(&en_exclusiva);
    <<Acceso en exclusiva al recurso compartido>>
    sem_post(&en_exclusiva);

    pthread_exit(0);
}
```

En esta solución el semáforo `en_exclusiva` se utiliza para asegurar la exclusión mutua en el acceso al dato a compartir. Su valor inicial debe ser 1, de esta manera, en cuanto un escritor consigue restar 1 a su valor puede modificar el dato y evitar que ningún otro proceso, sea lector o escritor, acceda al recurso compartido.

La variable `n_lectores` se utiliza para representar el número de procesos lectores que se encuentran accediendo de forma simultánea al recurso compartido. A esta variable acceden los procesos lectores en exclusión mutua utilizando el semáforo `de_lectura`. El valor de este semáforo, como el del cualquier otro que se quiera emplear para acceder en exclusión mutua a un fragmento de código, debe ser 1. De esta forma se consigue que sólo un proceso lector modifique el valor de la variable `n_lectores`.

El primer proceso lector será el encargado de solicitar el acceso al recurso compartido restando 1 al valor del semáforo `en_exclusiva` mediante la operación `wait`. El resto de procesos lectores que quieran acceder mientras esté el primero podrán hacerlo sin necesidad de solicitar el acceso al recurso compartido. Cuando el último proceso lector abandona la sección de código que permite acceder al recurso compartido, `n_lectores` se hace 0. En este caso deberá incrementar el valor del semáforo `en_exclusiva` para permitir que cualquier proceso escritor pueda acceder para modificar el recurso compartido.

Esta solución, tal y como se ha descrito, permite resolver el problema de los lectores-escritores pero concede prioridad a los procesos lectores. Siempre que haya un proceso lector consultado el valor del recurso, cualquier proceso lector podrá acceder sin necesidad de solicitar el acceso. Sin embargo, los procesos escritores deberán esperar hasta que haya abandonado la consulta el último lector. Existen también soluciones que permiten dar prioridad a los escritores.

El modelo de recursos limitados con semáforos

Una clásica solución al problema de los filósofos comensales consiste en representar cada uno de los palillos con un semáforo. Para que un filósofo pueda retirar un palillo de la mesa debe realizar una operación `wait` sobre él. Cuando, posteriormente, desea dejar los palillos sobre la mesa ejecuta una operación `signal` sobre los dos palillos. Cada uno de estos semáforos debe estar inicializado a 1. El programa 6.10 muestra una primera solución. Como se puede ver, en este programa se recurre a un vector de semáforos. El elemento `palillo[k]` representa el semáforo utilizado para el palillo *k*.

Programa 6.10 Estructura de un proceso filósofo *k* utilizando semáforos.

```
#define NUM_FILOSOFOS    6
Filosofo(k){
    for(;;){
```

```
        sem_wait(palillo[k]);
        sem_wait(palillo[k+1] % NUM_FILOSOFOS);

        come();

        sem_post(palillo[k]);
        sem_post(palillo[k+1] % NUM_FILOSOFOS);

        piensa();;
    }
}
```

La solución mostrada en el programa anterior, sin embargo, puede conducir a una situación de interbloqueo como se muestra en la figura 6.41.

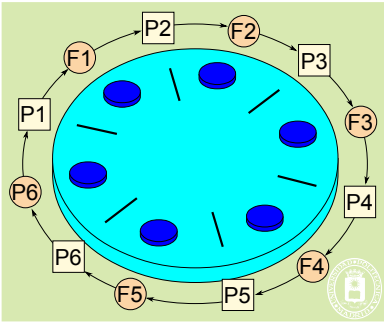


Figura 6.41 *Situación de interbloqueo del programa 6.5.*

Una solución posible para evitar el riesgo de interbloqueo consiste en permitir que sólo NUM_FILOSOFOS -1 filósofos puedan estar sentados a la mesa simultáneamente. En este caso, nunca aparecerá un bucle en el grafo de asignación de recursos (véase la figura 6.41). Para asegurar esto, es necesario modificar la estructura del programa 6.5, añadiendo un nuevo semáforo que representa a la mesa. Este semáforo tiene que estar inicializado a 5 para impedir que los seis filósofos estén sentados a la mesa al mismo tiempo. La nueva estructura del programa se muestra en el programa 6.11.

Programa 6.11 Estructura de un proceso filósofo *k* utilizando semáforos sin interbloqueos.

```
#define NUM_FILOSOFOS    6
Filosofo(k){
    for(;;){
        sem_wait(mesa);
        sem_wait(palillo[k]);
        sem_wait(palillo[k+1] % NUM_FILOSOFOS);

        come();

        sem_post(palillo[k]);
        sem_post(palillo[k+1] % NUM_FILOSOFOS);
        sem_post(mesa);

        piensa();

    }
}
```

Productor-consumidor con semáforos, procesos y memoria compartida

A continuación se presenta una solución a un problema de tipo productor-consumidor utilizando semáforos UNIX y objetos de memoria compartida. El proceso productor genera números enteros. El consumidor consume estos números imprimiendo su valor por la salida estándar.

En este ejemplo se emplean procesos convencionales creados con la llamada `fork`. Dado que este tipo de procesos no comparten memoria de forma natural, es necesario crear y utilizar un segmento de memoria compartida. En este caso se va a utilizar un *buffer* que reside en un segmento de memoria compartida y semáforos con nombre.

En la solución propuesta el **productor** se va a encargar de:

- Crear los semáforos mediante el servicio `sem_open`.
- Crear la zona de memoria compartida mediante la llamada `shm_open`.

- Asignar espacio al segmento creado. Para ello se emplea el servicio `ftruncate`, que permite asignar espacio a un fichero o a un segmento de memoria compartida.
- Proyectar el segmento de memoria compartida sobre su espacio de direcciones utilizando la llamada `mmap`.
- Acceder a la región de memoria compartida para insertar los elementos que produce.
- Desproyectar la zona cuando ha finalizado su trabajo mediante el servicio `munmap`.
- Por último este proceso se encarga de cerrar, mediante la llamada `close`, y destruir el objeto de memoria compartida previamente creado utilizando el servicio `shm_unlink`.

Los pasos que deberá realizar el proceso **consumidor** en la solución propuesta son los siguientes:

- Abrir el segmento de memoria compartida creada por el productor (`shm_open`). Esta operación debe realizarse una vez creada la región. En caso contrario la función `shm_open` devolvería un error.
- Abrir los semáforos que se van a utilizar.
- Proyectar la zona de memoria compartida en su espacio de direcciones (`mmap`).
- Acceder a la región de memoria compartida para eliminar los elementos de *buffer*.
- Desproyectar el segmento de memoria de su espacio de direcciones (`munmap`).
- Cerrar el objeto de memoria compartida (`close`).

El código a ejecutar por el proceso **productor** es el que se presenta en el programa 6.12.

Programa 6.12 Código del proceso productor utilizando objetos de memoria compartida y semáforos UNIX.

```
#include <sys/mmap.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

#define MAX_buffer            1024          /* tamaño del buffer */
#define DATOS_A_PRODUCIR      100000        /* datos a producir */

sem_t   *huecos;
sem_t   *elementos;
int *buffer;                                /* puntero al buffer de números enteros */

int main(void){
   int shd;
   /* se crean e inician semáforos */
   huecos = sem_open("HUECOS", O_CREAT, 0700, MAX_buffer);
   elementos = sem_open("ELEMENTOS", O_CREAT, 0700, 0);
   if (huecos == -1 || elementos == -1) { perror("Error en sem_open"); return 1; }
   /* se crea el segmento de memoria compartida utilizado como buffer circular */
   shd = shm_open("buffer", O_CREAT|O_WRONLY, 0700);
   if (shd == -1) { perror("Error en shm_open"); return 1; }
   ftruncate(shd, MAX_buffer*sizeof(int));
   buffer = (int *)mmap(NULL, MAX_buffer*sizeof(int), PROT_WRITE, MAP_SHARED, shd, 0);
   if (buffer == NULL) { perror("Error en mmap"); return 1; }
   /* se ejecuta el código del productor */
   productor();
   /* se desproyecta el buffer */
   munmap(buffer, MAX_buffer*sizeof(int));
   close(shd);
   shm_unlink("buffer");
   /* cierran y se destruyen los semáforos */
   sem_close(huecos);
   sem_close(elementos);
   sem_unlink("HUECOS");
   sem_unlink("ELEMENTOS");;
   return 0;
}
/* código del proceso productor */
void productor(void){
   int dato;                                /* dato a producir */
   int posicion = 0;                        /* posición donde insertar el elemento*/
   int j;

   for (j=0; j<DATOS_A_PRODUCIR; j++) {
```

```
        dato = j;
        sem_wait(huecos);                          /* un hueco menos */
        buffer[posicion]=dato;
        posicion=(posicion+1) % MAX_buffer;        /* nueva posición */
        sem_post(elementos);                       /* un elemento más */
    }
    return;
}
```

El proceso productor se encarga de crear una región de memoria compartida que denomina `buffer`. También crea los semáforos con nombre `HUECOS` y `ELEMENTOS`. Los permisos que asigna a la región de memoria compartida y a los semáforos vienen dados por el valor 0700, lo que significa que sólo los procesos que pertenezcan al mismo usuario del proceso que los creó podrán acceder a ellos para utilizarlos.

El consumidor utiliza los semáforos y región de memoria creados por el proceso productor, por lo que tiene que ejecutar después.

El código que ejecuta el proceso **consumidor** se muestra en el programa 6.13. Para garantizar que el consumidor ejecuta una vez creados los semáforos y la región de memoria se ha utilizado una espera activa, mediante un bucle que se repite hasta que se pueda abrir la región compartida. Esta es una solución **ineficiente** y que puede dejar al proceso ejecutando de forma indefinida el bucle de espera.

Programa 6.13 Código del proceso consumidor utilizando objetos de memoria compartida y semáforos UNIX.

```
#include <sys/mmap.h>
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

#define MAX_buffer      1024                       /* tamaño del buffer */
#define DATOS_A_PRODUCIR      100000               /* datos a producir */

sem_t    *huecos;
sem_t    *elementos;
int *buffer;                                       /* buffer de números enteros */

int main(void){
    int shd;
    /* se abre el segmento de memoria compartida utilizado como buffer circular */
    while((shd = open("buffer", O_RDONLY)) == -1) continue;  /* Espera activa, solución ineficiente */
    buffer = (int *)mmap(NULL, MAX_buffer*sizeof(int), PROT_READ,MAP_SHARED, shd, 0);
    if (buffer == NULL) { perror("Error en mmap"); return 1; }
    /* se abren los semáforos */
    huecos = sem_open("HUECOS", 0);
    elementos = sem_open("ELEMENTOS", 0);
    if (huecos == -1 || elementos == -1) { perror("Error en sem_open"); return 1; }
    /* se ejecuta el código del consumidor */
    consumidor();
    /* se desproyecta el buffer */
    munmap(buffer, MAX_buffer*sizeof(int));
    close(shd);

    /* se cierran semáforos */
    sem_close(huecos);
    sem_close(elementos);
    return 0;
}
/* código del proceso productor */
void consumidor(void){
    int dato;                                      /* dato a consumir */
    int posicion = 0;                              /* posición que indica el elemento a extraer */
    int j;
    for (j=0; j<DATOS_A_PRODUCIR; j++) {
        dato = j;
        sem_wait(elementos);                       /* un elemento menos */
        dato = buffer[posicion];
```

Espera activa, ¡Peligroso! e ¡Ineficiente!

```
        posicion=(posicion+1) % MAX_buffer;          /* nueva posición */
        sem_post(huecos);                            /* un hueco más */
    }
    return;
}
```

6.10.7. *Mutex* y variables condicionales POSIX

En esta sección se describen los servicios UNIX que permiten utilizar *mutex* y variables condicionales.

Para utilizar un *mutex* un programa debe declarar una variable de tipo `pthread_mutex_t` (definido en el fichero de cabecera `pthread.h`) e iniciarla antes de utilizarla.

- int **pthread_mutex_init** (pthread_mutex_t *mutex, pthread_mutexattr_t *attr);

Este servicio permite iniciar una variable de tipo *mutex*. El segundo argumento especifica los atributos con los que se crea el *mutex* inicialmente. En caso de que este segundo argumento sea `NULL`, se tomarán los atributos por defecto.

- int **pthread_mutex_destroy** (pthread_mutex_t *mutex);

Permite destruir un objeto de tipo *mutex*.

- int **pthread_mutex_lock** (pthread_mutex_t *mutex);

Este servicio se corresponde con la operación `mutex_lock` descrita en la sección 6.7.3 "Mutex y variables condicionales". Esta función intenta obtener el *mutex*. Si el *mutex* ya se encuentra adquirido por otro *thread*, el *thread* que ejecuta la llamada espera.

- int **pthread_mutex_unlock** (pthread_mutex_t *mutex);

Este servicio se corresponde con la operación `mutex_unlock` y permite al *thread* que la ejecuta liberar el *mutex*.

- int **pthread_cond_init** (pthread_cond_t *cond,pthread_condattr_t *attr);

Para emplear en un programa una variable condicional es necesario declarar una variable de tipo `pthread_cond_t` e iniciarla antes de usarla mediante el servicio `pthread_cond_init`. El segundo argumento especifica los atributos con los que se crea inicialmente la variable condicional. Si el segundo argumento es `NULL`, la variable condicional toma los atributos por defecto.

- int **pthread_cond_destroy** (pthread_cond_t *cond);

Permite destruir una variable de tipo condicional.

- int **pthread_cond_wait** (pthread_cond_t *cond, pthread_mutex_t_ *mutex);

Este servicio se corresponde con la operación `c_wait` sobre una variable condicional. El servicio suspende al *thread* hasta que otro *thread* ejecute una operación `c_signal` sobre la variable condicional pasada como primer argumento. De forma atómica se libera el *mutex* pasado como segundo argumento. Cuando el *thread* se despierte volverá a competir por el *mutex*.

- int **pthread_cond_signal** (pthread_cond_t *cond);

Este servicio se corresponde con la operación `c_signal` sobre una variable condicional. Se desbloquea a un *thread* suspendido en la variable condicional pasada como argumento a esta función. No tiene efecto si no hay ningún *thread* esperando sobre la variable condicional.

- int **pthread_cond_broadcast** (pthread_cond_t *cond);

Permite desbloquear a todos los *threads* suspendidos en una variable condicional.

6.10.8. Ejemplos con *mutex* y variables condicionales

Productor-consumidor con *mutex* y variables condicionales

En esta sección se presenta una posible solución al problema del productor-consumidor con *buffer* acotado presentado en secciones pasadas, utilizando *mutex* y variables condicionales. En este problema el recurso compartido es el *buffer* y los *threads* deben acceder a él en exclusión mutua. Para ello se utiliza un *mutex* sobre el que los *threads* ejecutarán operaciones `mutex_lock` y `mutex_unlock`.

Hay dos situaciones en las que el *thread* productor y el consumidor no pueden continuar su ejecución, una vez que han comenzado la ejecución del código correspondiente a la sección crítica:

- El productor no puede continuar cuando el *buffer* está lleno. Para que este *thread* pueda bloquearse es necesario que ejecute una operación `c_wait` sobre una variable condicional que se denomina `lleno`.

- El *thread* consumidor debe bloquearse cuando el *buffer* se encuentra vacío. En este caso se utilizará una variable condicional que se denomina `vacío`.

Para que ambos *threads* puedan sincronizarse correctamente, es necesario que ambos conozcan el número de elementos que hay en el *buffer*. Cuando el número de elementos es 0, el *thread* consumidor deberá bloquearse. Por su parte, cuando el número de elementos coincide con el tamaño del *buffer*, el *thread* productor deberá bloquearse. La variable `n_elementos` se utiliza para conocer el número de elementos insertados en el *buffer*.

El programa 6.14 presenta la solución a este problema empleando *mutex* y variables condicionales.

Programa 6.14 Productor-consumidor utilizando *mutex* y variables condicionales.

```
#include <pthread.h>
#include <stdio.h>

#define MAX_buffer            1024              /* tamaño del buffer */
#define DATOS_A_PRODUCIR      100000            /* datos a producir */

pthread_mutex_t mutex;        /* mutex para controlar el acceso al buffer compartido */
pthread_cond_t lleno;         /* esperar si lleno */
pthread_cond_t vacio;         /* espera si vacío */
int n_elementos=0             /* número de elementos en el buffer */
int buffer[MAX_buffer];       /* buffer común */

int main(int argc, char *argv[]){

   pthread_t th1, th2;

   pthread_mutex_init(&mutex, NULL);
   pthread_cond_init(&lleno, NULL);
   pthread_cond_init(&vacio, NULL);

   pthread_create(&th1, NULL, Productor, NULL);
   pthread_create(&th2, NULL, Consumidor, NULL);

   pthread_join(th1, NULL);
   pthread_join(th2, NULL);

   pthread_mutex_destroy(&mutex);
   pthread_cond_destroy(&lleno);
   pthread_cond_destroy(&vacio);
   return 0;
}

/* código del productor */
void Productor(void) {
   int dato, i, pos = 0;

   for(i=0; i<DATOS_A_PRODUCIR; i++ ) {
      dato = i;                                  /* producir dato */
      pthread_mutex_lock(&mutex);                /* acceder al buffer */
      while (n_elementos == MAX_buffer)          /* si buffer lleno */
         pthread_cond_wait(&lleno, &mutex);      /* se bloquea */
      buffer[pos] = i;
      pos = (pos + 1) % MAX_buffer;
      n_elementos = n_elementos + 1;
      pthread_cond_signal(&vacio);          /* buffer no vacío */
      pthread_mutex_unlock(&mutex);
   }
   pthread_exit(0);
}
/* código del consumidor */
void Consumidor(void) {
   int dato, i ,pos = 0;

   for(i=0; i<DATOS_A_PRODUCIR; i++ ) {
      pthread_mutex_lock(&mutex);                /* acceder al buffer */
      while (n_elementos == 0)                   /* si buffer vacío */
```

```
        pthread_cond_wait(&vacio, &mutex);          /* se bloquea */
      dato = buffer[pos];
      pos = (pos + 1) % MAX_buffer;
      n_elementos = n_elementos - 1 ;

      pthread_cond_signal(&lleno);                /* buffer no lleno */
      pthread_mutex_unlock(&mutex);
      printf("Consume %d \n", dato);              /* consume dato */
    }
    pthread_exit(0);
  }
```

El *thread* productor evalúa la condición `n_elementos == MAX_buffer` para determinar si el *buffer* está lleno. En caso de que sea así se bloquea ejecutando la función `pthread_cond_wait` sobre la variable condicional lleno.

Por su parte, el *thread* consumidor evalúa la condición `n_elementos == 0` para determinar si el *buffer* está vacío. En dicho caso se bloquea en la variable condicional `vacio`.

Cuando el productor inserta un elemento en el *buffer*, el consumidor podrá continuar en caso de que estuviera bloqueado en la variable `vacio`. Para despertar al *thread* consumidor el productor ejecuta el siguiente fragmento de código:

```
pthread_cond_signal(&vacio);   /* buffer no vacío */
```

Cuando el *thread* consumidor elimina un elemento del *buffer* y éste deja de estar lleno, despierta al *thread* productor en caso de que estuviera bloqueado en la variable condicional `lleno`. Para ello el *thread* consumidor ejecuta:

```
pos = (pos + 1) % MAX_buffer;
n_elementos --;
pthread_cond_signal(&lleno);   /* buffer no lleno */
```

Recuérdese que cuando un *thread* se despierta de la operación `pthread_cond_wait` vuelve de nuevo a competir por el *mutex*, por tanto mientras el *thread* que le ha despertado no abandone la sección crítica y libere el *mutex* no podrá acceder a ella.

Lectores-escritores con *mutex* y condiciones

El programa 6.15 presenta un ejemplo de código de escritores-lectores resuelto con *mutex* y condiciones.

Programa 6.15 Lectores-escritores utilizando *mutex* y variables condicionales.

```
pthread_mutex_t mutex;                          /* Control de acceso */
pthread_cond_t a_leer, a_escribir;              /* Condiciones de espera */
int leyendo, escribiendo;                       /* Variables de control: Estado del acceso */
int main(void)
{
   pthread_t th1, th2, th3, th4;
   pthread_mutex_init(&mutex, NULL);            /* Situación inicial */
   pthread_cond_init(&a_leer, NULL);
   pthread_cond_init(&a_escribir, NULL);
   pthread_create(&th1, NULL, Lector, NULL);    /* Arranque */
   pthread_create(&th2, NULL, Escritor, NULL);
   pthread_create(&th3, NULL, Lector, NULL);
   pthread_create(&th4, NULL, Escritor, NULL);
   pthread_join(th1, NULL);                     /* Esperar terminación */
   pthread_join(th2, NULL);
   pthread_join(th3, NULL);
   pthread_join(th4, NULL);
   pthread_mutex_destroy(&mutex);               /* Destruir */
   pthread_cond_destroy(&a_leer);
   pthread_cond_destroy(&a_escribir);
   return 0;
}

void Lector(void)
{
   pthread_mutex_lock(&mutex);
   while(escribiendo != 0)                      /*condición espera */
```

```
        pthread_cond_wait(&a_leer, &mutex);
    leyendo++;
    pthread_mutex_unlock(&mutex);

      <<Lecturas simultáneas del recurso compartido>>

    pthread_mutex_lock(&mutex);
    leyendo--;
    if (leyendo == 0)
      pthread_cond_signal(&a_escribir);
    pthread_mutex_unlock(&mutex);
    pthread_exit(0);
}

void Escritor(void)
{
    pthread_mutex_lock(&mutex);
    while(leyendo !=0 || escribiendo !=0)            /*condición espera */
        pthread_cond_wait(&a_escribir, &mutex);
    escribiendo++;
    pthread_mutex_unlock(&mutex);

    <<Acceso en exclusiva al recurso compartido>>

    pthread_mutex_lock(&mutex);
    escribiendo--;
    pthread_cond_signal(&a_escribir);
    pthread_cond_broadcast(&a_leer);
    pthread_mutex_unlock(&mutex);
    pthread_exit(0);
}
```

6.10.9. Colas de mensajes en UNIX

Las colas de mensajes UNIX son un mecanismo de comunicación y sincronización que pueden utilizar los procesos que ejecutan en la misma máquina, bien sea un multiprocesador o un multicomputador.

Una cola de mensajes en UNIX lleva asociado un nombre local que sigue la convención de nombrado de los ficheros. Cualquier proceso que conozca el nombre de una cola y tenga derechos de acceso sobre ella podrá enviar o recibir mensajes de dicha cola. Una cola se identifica en UNIX mediante una variable de tipo `mqd_t` (incluida en el fichero de cabecera `mqueue.h`). A continuación se presentan los servicios que ofrece UNIX para la utilización de colas de mensajes.

◙ mqd_t **mq_open** (char *name, int flag, mode_t mode, struct mq_attr *attr);
mqd_t **mq_open**(char *name, int flag);

Permite crear o abrir una cola de mensajes. Existen dos modalidades con las que se puede invocar a esta función según se quiera crear o simplemente abrir una cola ya existente. El primer argumento especifica el nombre de la cola que se quiere abrir o crear. Como se dijo anteriormente este nombre sigue la convención de nombrado de los ficheros. El valor del segundo argumento determina si la cola se quiere crear o abrir. Si este argumento incluye el valor `O_CREAT` se indica que la cola se quiere crear, en caso contrario se desea abrir una cola ya existente. Además este segundo argumento debe indicar el modo en el que se quiere acceder a la cola de mensajes:

- `O_RDONLY`, indica que se podrán recibir mensajes de la cola pero no enviar.
- `O_WRONLY`, permite enviar mensajes a la cola pero no recibirlos.
- `O_RDWR`, permite enviar y recibir mensajes de la cola.
- `O_NONBLOCK`, permite especificar si las operaciones `send` y `receive` sobre una cola serán bloqueantes o no.

En caso de que se desee crear la cola son necesarios los dos últimos argumentos de esta función. El tercer argumento indica los permisos con los que se crea la cola. El último argumento indica los atributos con los que se crea la cola. Los atributos asociados a una cola se establecen con una variable de tipo `mq_attr`, que es una estructura con los siguientes campos:

- `mq_msgsize`, especifica el tamaño máximo de un mensaje almacenado en la cola.
- `mq_maxmsg`, indica el número de mensajes que se pueden almacenar en la cola.

La función `mq_open` devuelve un descriptor de cola de mensaje de tipo `mqd_t` que se puede emplear en las siguientes funciones o -1 en caso de error.

- int **mq_close**(mqd_t mqdes);

Cierra una cola de mensajes previamente abierta.

- int **mq_unlink**(char *name);

Borra una cola de mensajes.

- int **mq_send**(mqd_t mqdes, char *msg, size_t len, int prio);

Envía el mensaje apuntado por el segundo argumento a la cola de mensajes que se especifica como primer argumento de la función. El tercer argumento indica la longitud de mensaje que se envía. Si la cola se creó sin el valor `O_NONBLOCK`, una llamada a esta función bloqueará al proceso que la ejecuta si la cola de mensajes se encuentra llena. Si la cola se creó con el valor `O_NONBLOCK`, el proceso no se bloqueará cuando la cola se encuentre llena, devolviendo un error (un valor -1).

El cuarto argumento especifica la prioridad con la que se envía el mensaje. Los mensajes se insertan en la cola según su prioridad, el primer mensaje será el de mayor prioridad.

- int **mq_receive**(mqd_t mqdes, char *msg, size_t len, int *prio);

Permite recibir un mensaje de una cola de mensajes. El mensaje se almacena en la dirección pasada como segundo argumento. El tercer argumento indica el tamaño del mensaje que se desea recibir. La función `mq_receive` siempre extrae de la cola el mensaje de mayor prioridad, y esta prioridad se almacena en el cuarto argumento si éste es distinto de `NULL`. La función devuelve el tamaño del mensaje que se extrae de la cola.

Si la cola se creó sin el valor `O_NONBLOCK`, la llamada bloqueará al proceso si la cola está vacía. En caso contrario la función no bloquea al proceso y la llamada devuelve un error.

La función `mq_receive` devuelve el tamaño del mensaje recibido o -1 en caso de error.

- int **mq_setattr**(mqd_t mqdes, struct mq_attr *qstat, struct mq_attr *oldmqstat);

Permite cambiar los atributos asociados a una cola de mensajes. Los nuevos atributos se pasan en el segundo argumento. Si el tercer argumento es distinto de `NULL`, entonces se almacenan en él los antiguos atributos asociados a la cola.

- int **mq_getattr**(mqd_t mqdes, struct mq_attr *qstat);

Devuelve los atributos de una cola de mensajes. Esta llamada almacena en el segundo argumento los atributos asociados a una cola de mensajes.

6.10.10. Ejemplos de colas de mensajes en UNIX

Secciones críticas con colas de mensajes

La operación `mq_receive` sobre una cola de mensajes permite bloquear el proceso que la ejecuta cuando la cola de mensajes se encuentra vacía. Esta característica permite emplear las colas UNIX para resolver el problema de la sección crítica, de forma análoga a como se indicó en la sección 6.7.5 “Paso de mensajes”.

En este ejemplo se crea una cola de mensajes con capacidad para almacenar un único mensaje que hará las funciones de testigo. Cuando un proceso quiere acceder al código de la sección crítica ejecutará la función `mq_receive` para extraer el mensaje que hace de testigo de la cola. Si la cola está vacía, el proceso se bloquea ya que en este caso el testigo lo posee otro proceso.

Cuando el proceso finaliza la ejecución del código de la sección crítica inserta de nuevo el mensaje en la cola mediante la operación `mq_send`.

Inicialmente alguno de los procesos que van a ejecutar el código de la sección crítica deberá crear la cola e insertar el testigo inicial ejecutando el siguiente fragmento de código:

```
mqd_t cola_mutex;          /* se declara la cola donde insertar el testigo */
struct mq_attr attr;       /* atributos asociados a la cola */
char testigo;              /* mensaje que hace de testigo */

attr.mq_maxmsg = 1;        /* número máximo de mensajes */
attr.mq_msgsize = 1;       /* tamaño del mensaje */

cola_mutex = mq_open(“mutex”, O_CREAT|O_RDWR, 0700, &attr);

/* se inserta el primer mensaje que hace las veces de testigo */
mq_send(cola_mutex, &testigo, 1, 0);
```

Este proceso crea una cola denominada `mutex`. El resto de procesos deberán abrir esta cola para su utilización ejecutando el siguiente fragmento de código:

```
mqd_t cola_mutex;          /* se declara la cola */

cola_mutex = mq_open(“mutex”, O_RDWR);
```

Una vez que todos los procesos tienen acceso a la cola, sincronizan su acceso a la sección crítica ejecutando el siguiente fragmento de código:

```
mq_receive(cola_mutex, &testigo, 1, 0);
< código de la sección crítica >
mq_send(cola_mutex, &testigo, 1, 0);
```

De esta forma el primer proceso que ejecuta la operación `mq_receive` extrae el mensaje de la cola y la vacía, de tal manera que el resto de procesos se bloqueará hasta que de nuevo vuelva a haber un mensaje disponible en la cola. Este mensaje lo inserta el proceso que lo extrajo mediante la operación `mq_send`. De esta forma, alguno de los procesos bloqueados se despertará y volverá a extraer el mensaje de la cola vaciándola.

Productor-consumidor con colas de mensajes

A continuación se presenta una posible solución al problema del productor-consumidor utilizando colas de mensajes. Esta solución vuelve a ser similar a la que se ha venido presentando en secciones anteriores.

En este ejemplo los datos que se producen y consumen son de tipo entero y se crea una cola de mensajes con capacidad para almacenar 1024 mensajes cada uno de ellos del tamaño necesario para representar un entero. Cuando el proceso productor produce un elemento, lo inserta en la cola de mensajes mediante la operación `mq_send`. Si la cola se encuentra llena esta operación bloquea al proceso productor.

Cuando el proceso consumidor desea procesar un nuevo elemento extrae de la cola de mensajes un nuevo dato mediante la operación `mq_receive`. Si la cola se encuentra vacía el proceso se bloquea hasta que el productor cree algún nuevo elemento. En el ejemplo que se presenta se deja al proceso productor la responsabilidad de crear la cola de mensajes.

El código del proceso productor es que el se muestra en el programa 6.16.

Programa 6.16 Proceso productor utilizando colas de mensajes UNIX.

```
#include <mqueue.h>
#include <stdio.h>
#define MAX_buffer          1024          /* tamaño del buffer */
#define DATOS_A_PRODUCIR    100000        /* datos a producir */

mqd_t almacen;        /* cola de mensaje donde dejar los datos producidos y recoger los datos a consumir */

int main(void) {
    struct mq_attr attr;

    attr.mq_maxmsg = MAX_buffer;
    attr.mq_msgsize = sizeof(int);

    almacen = mq_open("ALMACEN", O_CREAT|O_WRONLY, 0700, &attr);
    if (almacen == -1){
        perror("mq_open");
        return 1;
    }
    Productor();
    mq_close(almacen);
    return 0;
}

/* código del productor */
void Productor(void){
    int dato;
    int i;

    for(i=0; i<DATOS_A_PRODUCIR; i++ )
    {
        /* producir dato */
        dato = i;
        if (mq_send(almacen, &dato, sizeof(int), 0) == -1) {
            perror("Error en mq_send");
            mq_close(almacen);
            exit(1);
        }
    }
    return;
}
```

El proceso consumidor debe ejecutar el código del programa 6.17.

Programa 6.17 Proceso consumidor utilizando colas de mensajes UNIX.

```
#include <mqueue.h>
#include <stdio.h>
#define MAX_buffer          1024              /* tamaño del buffer */
#define DATOS_A_PRODUCIR    100000            /* datos a producir */

mqd_t almacen;        /* cola de mensaje donde dejar los datos producidos recoger los datos a consumir */
int main(void) {
   struct mq_attr attr;

   almacen = mq_open("ALMACEN", O_RDONLY);
   if (almacen == -1){
      perror("mq_open");
      return 1;
   }
   Consumidor();
   mq_close(almacen);
   return 0;
}

/* código del consumidor */
void Consumidor(void) {
   int dato;
   int i;

   for(i=0; i<DATOS_A_PRODUCIR; i++ )
   {
      /* recibir dato */
      if (mq_receive(almacen, &dato, sizeof(int), 0) == -1) {
         perror("Error en mq_receive");
         mq_close(almacen);
         exit(1);
      }
      /* consumir el dato */
      printf("Dato consumido %d\n", dato);
   }
      return;
}
```

Servidor concurrente con colas de mensajes

En esta sección se presenta un ejemplo muy sencillo de aplicación cliente-servidor que emplea un servidor concurrente mediante *threads* para servir las peticiones de los procesos clientes. El modelo que sigue este servidor es el de un *thread* distribuidor. Los clientes solicitan dos tipos de operaciones a este servidor sumar y multiplicar dos números de tipo entero.

El empleo de un servidor mediante *threads* permite ofrecer servicio concurrente a los clientes. Para ello cada petición al *thread* distribuidor supone la creación de un *thread* que se encarga de satisfacer dicha petición. El *thread* se encontrará ligado al cliente durante todo el tiempo que dure la operación correspondiente, y se encargará de responder al *thread* cuando la operación haya concluido. Una vez ejecutada la petición el *thread* se destruye.

Con esta estructura mientras un *thread* está atendiendo a un cliente, el proceso servidor puede ejecutar de forma concurrente y esperar la recepción de nuevas peticiones por parte de otros clientes, lo que permite que dentro del servidor se puedan servir de forma concurrente peticiones de diferentes clientes. En la figura 6.42 se representa de forma gráfica el funcionamiento de este tipo de servidores.

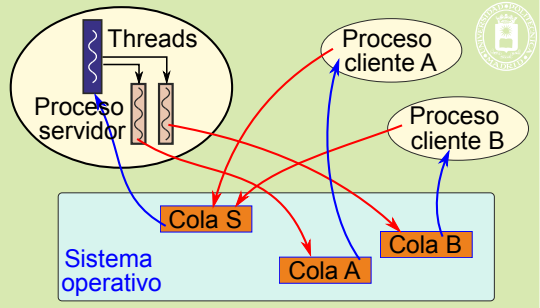


Figura 6.42 *Estructura de un servidor multithread con comunicación por medio de colas de mensajes.*

El mensaje que envían los procesos clientes al servidor en nuestro caso concreto viene determinado por la siguiente estructura:

```
struct peticion {
    int operacion;
    int operando_A;
    int operando_B;
    char cliente[256];
};
```

Donde `operación` determina si se va a realizar la suma o la multiplicación de los números representados por `operando_A` y `operando_B`. El último campo de esta estructura representa el nombre de la cola del cliente a la que el servidor debe enviar el resultado. Las dos operaciones posibles vienen determinadas por las siguientes constantes:

```
#define SUMA        0
#define PRODUCTO  1
```

Tanto la estructura anterior como las dos constantes anteriores, se pueden incluir en un fichero de cabecera, que se denomina `mensaje.h`.

Una vez calculado el resultado el *thread* encargado de satisfacer la petición de un cliente envía a la cola de mensajes asociado al cliente el resultado de la operación.

El programa 6.18 presenta el código del proceso servidor.

Programa 6.18 Código del proceso servidor.

```
#include "mensaje.h"
#include <mqueue.h>
#include <pthread.h>
#include <stdio.h>

/* mutex y variables condicionales para proteger la copia del mensaje*/
pthread_mutex_t mutex_mensaje;
int mensaje_no_copiado = TRUE;          /* TRUE con valor a 1 */
pthread_cond_t cond_mensaje;

int main(void)
{
    mqd_t q_servidor;           /* cola del servidor */
    struct peticion mess;       /* mensaje a recibir */
    struct mq_attr q_attr;      /* atributos de la cola */
    pthread_attr_t t_attr;      /* atributos de los threads */

    attr.mq_maxmsg = 20;
    attr.mq_msgsize = sizeof(struct peticion));

    q_servidor = mq_open("SERVIDOR", O_CREAT|O_RDONLY, 0700, &attr);
    if (q_servidor == -1) {
        perror("No se puede crear la cola de servidor");
        return 1;
    }

    pthread_mutex_init(&mutex_mensaje, NULL);
    pthread_cond_init(&cond_mensaje, NULL);
    pthread_attr_init(&attr);

    /* atributos de los threads */
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

    while (TRUE) {
        mq_receive(q_servidor, &mess, sizeof(struct mensaje), 0);

        pthread_create(&thid, &attr, tratar_mensaje, &mess);

        /* se espera a que el thread copie el mensaje */
        pthread_mutex_lock(&mutex_mensaje);
        while (mensaje_no_copiado)
            pthread_cond_wait(&cond_mensaje, &mutex_mensaje);
```

```
        mensaje_no_copiado = TRUE;
        pthread_mutex_unlock(&mutex_mensaje);
    }
}

void tratar_mensaje(struct mensaje *mes){
    struct peticion mensaje;        /* mensaje local */
    struct mqd_t q_cliente;         /* cola del cliente */
    int resultado;                  /* resultado de la operación */

    /* el thread copia el mensaje a un mensaje local */
    pthread_mutex_lock(&mutex_mensaje);
    memcpy((char *) &mensaje, (char *)&mes, sizeof(struct peticion));

    /* ya se puede despertar al servidor*/
    mensaje_no_copiado = FALSE;          /* FALSE con valor 0 */
    pthread_cond_signal(&cond_mensaje);
    pthread_mutex_unlock(&mutex_mensaje);

    /* ejecutar la petición del cliente y preparar respuesta */
    if (mensaje.operacion == SUMA)
        resultado = mensaje_local.operando_A + mensaje_local.operando_B;
    else
        resultado = mensaje_local.operando_A * mensaje_local.operando_B;

    /* Se devuelve el resultado al cliente */
    /* Para ello se envía el resultado a su cola */
    q_cliente = mq_open(mensaje_local.nombre, O_WRONLY);

    if (q_cliente == -1)
        perror("No se puede abrir la cola del cliente */
    else {
        mqsend(q_cliente, (char *) &resultado, sizeof(int), 0);
        mq_close(q_cliente);
    }
    pthread_exit(0);
}
```

El servidor crea la cola donde los clientes van a enviar las peticiones y a continuación entra en un bucle infinito esperando peticiones de los clientes. Cada vez que llega una petición, se crea un *thread* al que se le pasa como parámetro la dirección de memoria donde reside el mensaje recibido, el cual incluye la petición que ha elaborado el cliente.

Llegados a este punto el *thread* principal no puede esperar la recepción de otro mensaje hasta que el *thread* que se ha creado copie el mensaje en un mensaje local. Si esto no se hiciera, el *thread* principal podría sobrescribir el mensaje `mess` modificando de igual forma el mensaje que se ha pasado al *thread* encargado de satisfacer la petición, lo que provocaría resultados erróneos. Para conseguir esto, el *thread* principal debe bloquearse hasta que el *thread* encargado de ejecutar la petición del cliente haya copiado el mensaje pasado como argumento. Una vez que este *thread* ha copiado el mensaje despierta al *thread* distribuidor para que continúe la recepción de nuevas peticiones.

Para conseguir esta correcta sincronización se utiliza la variable de tipo *mutex* `mutex_mensaje`, la variable condicional `cond_mensaje`, y la variable `no_copiado` utilizada como predicado lógico sobre la cual realizar la espera en el *thread* principal. En la figura 6.43 se representa el proceso que se ha descrito.

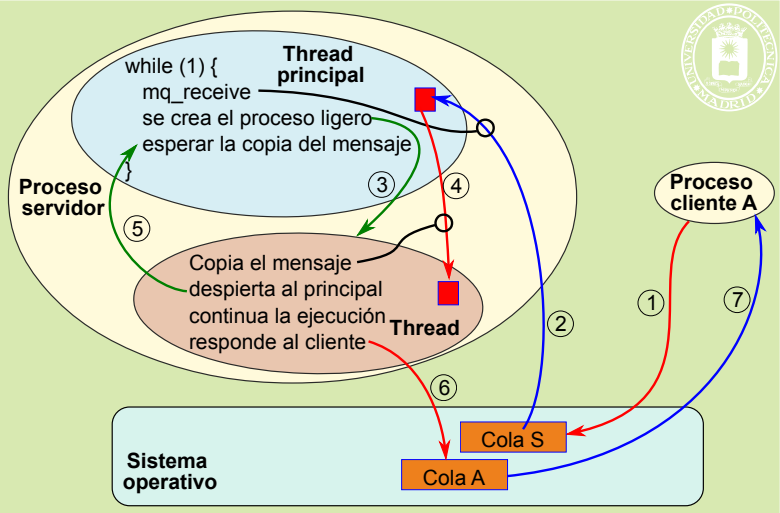


Figura 6.43 *Estructura del proceso servidor.*

Una vez que el *thread* creado ha copiado el mensaje en uno local, se encarga de satisfacer la petición y devolver los resultados a la cola del cliente, cuyo nombre envía el cliente que ha efectuado la petición en el mensaje.

El código del proceso cliente que quiere realizar una suma se muestra en el programa 6.19.

Programa 6.19 Código del proceso cliente.

```
#include <mensaje.h>
#include <pthread.h>
#include <mqueue.h>
#include <stdio.h>

int main(void)
{
    mqd_t q_servidor;          /* cola del servidor */
    struct peticion mess;      /* mensaje a enviar */
    mqd_t l_cliente;           /* cola del cliente */
    struct mq_attr q_attr;     /* atributos de la cola del */
    char cliente[256];         /* nombre de la cola cliente */
    int resultado;             /* resultado de la operación */

    attr.mq_maxmsg = 1;
    attr.mq_msgsize = sizeof(struct peticion));

    /* se asigna un nombre a la cola cliente en función de
       su identificador de proceso */
    sprintf(cliente, "CLIENTE%d", getpid());
    q_cliente = mq_open(cliente, O_CREAT|O_RDONLY, 0700, &attr);
    if (q_cliente == -1) {
       perror("No se puede crear la cola del cliente");
       return 1;
    }

    /* se abre la cola del servidor */
    q_servidor = mq_open("SERVIDOR", O_WRONLY, 0700, &attr);
    if (q_servidor == -1) {
       perror("No se puede abrir la cola del cliente");
       return 1;
    }

    /* se rellena el mensaje de petición */
    mess.operacion = SUMA;
    mess.operando_A = 1000;
    mess.operando_B = 4000;
    strcpy(mess.cliente, cliente);

    mq_send(q_servidor, (char *)&mess, sizeof(struct peticicion), 0);
    /* se espera la respuesta del servidor */
    mq_receive(q_cliente, (char *)&resultado, sizeof(int), 0);
```

```
    printf("El resultado es %d\n", resultado);

    mq_close(q_servidor);
    mq_close(q_cliente);
    mq_unlink(cliente);
    return 0;
}
```

El código del proceso cliente es mucho más sencillo, simplemente se encarga de crear su cola de mensajes, a la que asigna un nombre que incluye la cadena de caracteres `CLIENTE` seguido del identificador de proceso del proceso cliente. A continuación se abre la cola del proceso servidor y se envía a dicha cola la petición. Una vez enviada la petición, el proceso cliente se bloquea en la operación `mq_receive` hasta que llega la respuesta del servidor.

La tabla 6.4 recoge las principales características de los mecanismos de comunicación de UNIX y la tabla 6.5 las principales características de los mecanismos de sincronización de UNIX.

Tabla 6.4 *Características de los mecanismos de comunicación de UNIX.*

Mecanismo	Nombrado	Identificador	Almacenamiento	Flujo de datos
Tubería	Sin nombre Con nombre	Descriptor de fichero	Sí	Unidireccional
Cola de mensajes	Con nombre	Descriptor propio	Sí	Bidireccional

Tabla 6.5 *Características de los mecanismos de sincronización de UNIX.*

Mecanismo	Nombrado	Identificador
Tuberías	Sin nombre Con nombre	Descriptor de fichero
Semáforos	Sin nombre Con nombre	Variable de tipo semáforo
mutex y variables condicionales	Sin nombre	Variable de tipo *mutex* y condicional
Cola de mensajes	Con nombre	Descriptor propio

6.10.11. Cerrojos en UNIX

En UNIX los cerrojos se establecen con el servicio `fcntl` que se analiza seguidamente.

◙ int **fcntl** (int fd, int cmd, struct flock *flockptr);

En el capítulo "5.16.1 Servicios UNIX para ficheros" se indicó que este servicio sirve para establecer cerrojos sobre un fichero previamente abierto. Según la versión del sistema operativo pueden establecerse distintas implementaciones de cerrojos consultivos y obligatorios. Aquí nos centraremos en la **implementación clásica de cerrojos consultivos** definida en POSIX.

Los **argumentos** son los siguientes:

- fd: descriptor de fichero sobre el que se hace la operación.
- cmd: Operación a realizar, que puede ser:
  - F_GETLK. Comprueba si existe un cerrojo.
  - F_SETLK. Establece cerrojo. El comportamiento del servicio es no bloqueante o asíncrono. Si no se puede establecer el cerrojo, el proceso no se bloquea y devuelve -1 y `errno` = EACCES o EAGAIN.
  - F_SETLKW. Establece cerrojo. El comportamiento del servicio es bloqueante o síncrono. Si no se puede establecer el cerrojo, el proceso se bloquea hasta que se pueda establecer dicho cerrojo. En caso de que llegue una señal al proceso el servicio termina y devuelve -1.
- *flockptr: El cerrojo se establece sobre la región y con la modalidad indicada en esta estructura, cuya declaración es:

```
struct flock{
    short l_type;       /* F_RDLCK (compartido), F_WRLCK (exclusivo), F_UNLCK (elimina)   */
    short l_whence;     /* SEEK_SET, SEEK_CUR, SEEK_END */
    off_t l_start;      /* Desplazamiento relativo a l_whence  */
    off_t l_len;        /* Tamaño. 0 == hasta EOF       */
    pid_t l_pid;        /* Sólo para F_GETLK, primer pid con lock*/
}
```

Descripción.

Los cerrojos se asocian al proceso y para establecer un cerrojo compartido se tiene que haber abierto el fichero con permisos de lectura, mientras que para el cerrojo exclusivo el fichero tiene que estar abierto para escritura. Si se quieren cerrojos de los dos tipos hay que abrir el fichero para lectura y escritura.

Si el proceso termina, se eliminan todos los cerrojos.

Si el proceso cierra un descriptor cualquiera del fichero (que puede tener más de uno) se pierden los cerrojos.

Los cerrojos no se heredan.

No funcionan en sistemas de ficheros en red, como NFS.

Para determinar el comienzo del cerrojo se especifica un origen y un desplazamiento. El origen puede ser, como con el servicio lseek, SEEK_SET, SEEK_CUR o SEEL_END. El desplazamiento puede negativo. La figura 6.44 muestra algunos ejemplos de cerrojos establecidos sobre un ficheros.

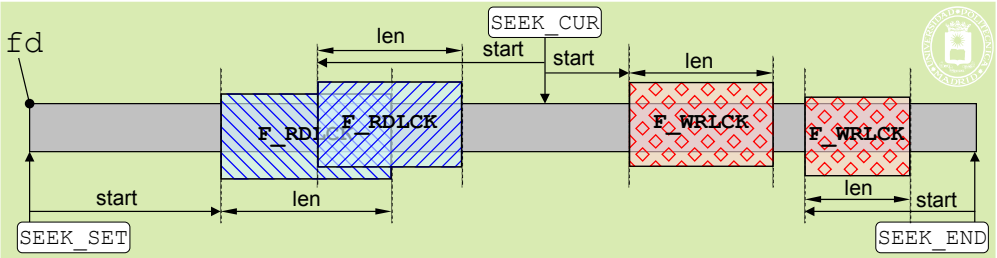


Figura 6.44 *Distintos cerrojos establecidos en un fichero.*

Cerrojos de fichero abierto

Se trata de una versión de cerrojos específica de Linux. Son muy parecidos a los clásicos descritos antes, pero se asocian al descriptor y no al proceso, por lo que se heredan. Además, solamente se pierden cuando se cierra el último descriptor del fichero.

Cerrojos obligatorios

Linux permite cerrojos obligatorios, pero no son muy fiables. Su uso exige las siguientes condidiones:

- Al montar el sistema de ficheros hay que habilitarlos con la opción: `mount -o mand`
- En el fichero hay que deshabilitar los permisos de grupo y habilitar los permisos: `setgroup-ID` (SGID)

◙ int **flock** (int fildes, int operation);

UNIX incluye la llamada `flock` para poder bloquear y desbloquear un fichero completo. Donde `operation` indica las posibles operaciones de bloqueo que se pueden establecer sobre el fichero. `LOCK_SH` permite poner un bloqueo compartido sobre al fichero, de forma que varios procesos pueden tener un bloqueo sobre el fichero al mismo tiempo (acceso múltiple controlado). `LOCK_EX` permite poner un bloqueo exclusivo sobre al fichero, de forma que sólo un proceso puede tener un bloqueo sobre el fichero en un determinado instante (acceso único). `LOCK_UN`, libera un bloqueo. `LOCK_NB`, indica que el proceso no se quiere quedar esperando si la adquisición de un cerrojo no tiene éxito. Se especifica como `LOCK_SH | LOCK_NB`. El mismo fichero no puede tener al mismo tiempo ficheros exclusivos y compartidos.

6.10.12. Ejemplos con cerrojos UNIX

Lectores-escritores con cerrojos fcntl

Se presenta un ejemplo de lectores-escritores resuelto con cerrojos. Se utiliza un fichero para comunicar los procesos, en el que solamente se escribe un entero. Cada escritor repite 10 veces la siguiente secuencia: lee el entero, lo incrementa y lo vuelve a escribir, sustituyendo al valor anterior. Cada lector repite 10 veces la lectura del entero almacenado en el fichero e imprime su valor.

Dado que el fichero solamente contendrá un entero, el cerrojo se hace sobre todo el fichero.

En caso de que el fichero esté vacío inicialmente, las lecturas no actualizarán la variable `val`, tanto en el lector como en el escritor. Para evitar la indefinición que ello supone, se ha inicializado dicha variable a 0.

Programa 6.20 Ejemplo de lectores y escritores con cerrojos fcntl. Se han marcado en verde las secciones de acceso protegido, que, en el caso del lector, es de acceso múltiple, pero en el caso del escritor es con exclusión mutua.

```
/* LECTOR */

int main(void)
{
    int fd, cnt;
    int val = 0;
    struct flock fl;
    fl.l_whence = SEEK_SET;
```

```
    fl.l_start = 0;
    fl.l_len = 0;                    /* Cerrojo sobre todo el fichero */
    fl.l_pid = getpid();
    fd = open("BD", O_RDONLY);
    for (cnt = 0; cnt < 10; cnt++)
    {
      fl.l_type = F_RDLCK;           /* Cerrojo compartido */
      fcntl(fd, F_SETLKW, &fl);      /* Servicio bloqueante */
      lseek(fd, 0, SEEK_SET);
      read(fd, &val, sizeof(int));
      printf("%d\n", val);
      /*Lecturas simultáneas del recurso compartido*/
      fl.l_type = F_UNLCK;
      fcntl(fd, F_SETLK, &fl);       /* Elimina el cerrojo */
    }
    return 0;
  }

/* ESCRITOR */

  int main(void)
  {
    int fd, cnt;
    int val = 0;
    struct flock fl;
    fl.l_whence = SEEK_SET;
    fl.l_start = 0;
    fl.l_len = 0;                    /* Cerrojo sobre todo el fichero */
    fl.l_pid = getpid();
    fd = open("BD", O_RDWR);
    for (cnt = 0; cnt < 10; cnt++)
    {
      fl.l_type = F_WRLCK;           /* Cerrojo exclusivo */
      fcntl(fd, F_SETLKW, &fl);      /* Servicio bloqueante */
      lseek(fd, 0, SEEK_SET);
      read(fd, &val, sizeof(int));
      val++; /*Acceso exclusivo*/
      lseek(fd, 0, SEEK_SET);
      write(fd, &val, sizeof(int));
      fl.l_type = F_UNLCK;
      fcntl(fd, F_SETLK, &fl);       /* Elimina el cerrojo */
    }
    return 0;
  }
```

6.11. SERVICIOS WINDOWS

En esta sección se presentan los servicios que ofrece Windows para la sincronización y comunicación de procesos. Windows dispone de dos mecanismos de comunicación, que también se pueden utilizar para sincronizar. Estos mecanismos son las tuberías y los *mailslots*. Como mecanismos de sincronización puros, Windows dispone de secciones críticas, semáforos, *mutex* y eventos.

Los objetos de sincronización en Windows tienen asociados un manejador que se puede utilizar en funciones de espera (*wait*), que permiten a un *thread* bloquear su ejecución. Las dos principales funciones de tipo *wait* son `WaitForSingleObject` y `WaitForMultipleObject` (aclaración 6.6).

Aclaración 6.6. Estos servicios también se utilizan en Windows para esperar la terminación de uno o más *threads*, como se vio en el capítulo "3 Procesos".

Los prototipos de estas funciones son:

- DWORD **WaitForSingleObject**(HANDLE hHandle, DWORD dwMilliseconds);

Esta función bloquea el *thread* hasta que el objeto con manejador `hHandle` haya sido notificado mediante la función correspondiente. El segundo argumento especifica el tiempo de bloqueo. Si `dwMilliseconds` es `INFINITE`, la función bloquea indefinidamente al proceso hasta que el evento sea notificado.

◙ DWORD **WaitForMultipleObjects**(DWORD nCount, CONST HANDLE *lpHandles,
BOOL fWaitFail, DWORD dwMilliseconds);

Esta función bloquea el *thread* hasta que uno o todos los objetos especificados en el segundo argumento hayan sido notificados. El primer argumento indica el número de objetos por los que se espera. El segundo parámetro es un vector de manejadores. Cada manejador referencia a un objeto por el que se espera. Si `fWaitAll` es `TRUE`, la llamada bloqueará al *thread* hasta que todos los objetos hayan sido notificados, en caso contrario el *thread* se despertará en cuanto un objeto haya sido notificado. El último argumento tiene el mismo significado que en la función anterior.

En las siguientes secciones se describen los mecanismos de comunicación y sincronización de Windows, así como el comportamiento y uso concreto de las funciones anteriores en los mecanismos de sincronización.

6.11.1. Tuberías en Windows

Windows ofrece, al igual que UNIX, dos tipos de tuberías: sin nombre y con nombre. Las primeras presentan las mismas características que los *pipes* en UNIX, sólo permiten transferir datos entre procesos que hereden el *pipe*. Las tuberías con nombre, sin embargo, tienen las siguientes propiedades:

- Las tuberías con nombre están orientadas a mensajes, de tal manera que un proceso puede leer mensajes de longitud variable, escritos por otro proceso.
- Son bidireccionales, así que dos procesos pueden intercambiar mensajes utilizando una misma tubería.
- Puede haber múltiples instancias independientes de la misma tubería. Todas las instancias comparten el mismo nombre, pero cada una de ellas tiene sus propios *buffer*s y manejadores. El uso de instancias permite que múltiples clientes puedan utilizar la misma tubería con nombre de forma simultánea.
- Se pueden utilizar en sistemas conectados a una red, por lo que los hace especialmente útiles en aplicaciones cliente-servidor. En este tipo de aplicaciones un proceso (el servidor) crea una tubería con nombre y los procesos clientes se conectan a una instancia de esa tubería.

Las tuberías sin nombre tienen asociados dos manejadores: uno para lectura y otro para escritura. Las tuberías con nombre únicamente tienen asociado un manejador que se puede utilizar para operaciones de lectura, de escritura o ambas.

A continuación se presentan los principales servicios que ofrece Windows para trabajar con tuberías sin nombre.

◙ BOOL **Createpipe**(PHANDLE phRead, PHANDLE phWrite,
LPSECURITY_ATTRIBUTES lpsa, DWORD cbPipe);

Este servicio permite crear una tubería sin nombre. Los dos primeros argumentos representan los manejadores de lectura y escritura respectivamente. El argumento `lpsa`, indica los atributos de seguridad asociados a la tubería. El parámetro `cbPipe`, indica el tamaño de la tubería. Si su valor es cero se toma el valor por defecto. La llamada devuelve `TRUE` en caso de éxito y `FALSE` en caso contrario.

El atributo de seguridad viene dado por una estructura (`LPSECURITY_ATTRIBUTES`) con tres campos:

- `nLength`: especifica el tamaño de la estructura en bytes.
- `lpSecurityDescriptor`: puntero a un descriptor de seguridad que controla el acceso al objeto. Si es `NULL` se utiliza el descriptor de seguridad asociado al proceso que realiza la llamada.
- `bInheritHandle`: indica si el objeto puede ser heredado por los procesos que se creen. Si es `TRUE`, los procesos creados heredan el manejador (véase el consejo de programación 6.1).

Consejo de programación 6.1. La forma de utilizar los atributos de seguridad en la mayoría de los casos es la siguiente:

```
LPSECURITY_ATTRIBUTES lpsa; /* variable de tipo LPSECURITY_ATTRIBUTES */
/* se rellena la estructura */
lpsa.nLength = sizeof(LPSECURITY_ATTRIBUTES);
lpsa.lpSecurityDescriptor = NULL;        /* descriptor por defecto */
lpsa.bInheritHandle = TRUE;              /* si se quiere herencia */
lpsa.bInheritHandle = FALSE;             /* si no se quiere herencia */
```

◙ HANDLE **CreateNamedpipe** (LPCTSTR lpszPipeName, DWORD fdwOpenMode,
DWORD fdwPipeMode, DWORD nMaxInstances, DWORD cbOutBuf,
DWORD cbInBuf, DWORD dwTimeOut, LPSECURITY_ATTRIBUTES lpsa);

Este servicio permite crear una tubería con nombre. El parámetro `lpszPipeName` indica el nombre de la tubería. El nombre de la tubería debe tomar la siguiente forma:

\\.\pipe\[camino]nombre_tuberia

El argumento `fdwOpenMode` puede especificar alguno de los siguientes valores:

- `PIPE_ACCESS_DUPLEX`, permite que el manejador asociado a la tubería con nombre se pueda utilizar en operaciones de lectura y escritura.
- `PIPE_ACCESS_INBOUND`, sólo permite realizar operaciones de lectura de la tubería.

- PIPE_ACCESS_OUTiBOUND, sólo permite realizar operaciones de escritura sobre la tubería.

El argumento fdwPipeMode tiene tres pares de valores mutuamente exclusivos. Estos valores indican si la escritura está orientada a mensajes o a bytes, si la lectura se realiza en mensajes o en bloques, y si las operaciones de lectura bloquean. Estos valores son:

- PIPE_TYPE_BYTE y PIPE_TYPE_MESSAGE, indican si los datos se escriben como un flujo de bytes o como mensajes.
- PIPE_READMODE_BYTE y PIPE_READMODE_MESSAGE, indican si los datos se leen como un flujo de bytes o como mensajes. PIPE_READMODE_MESSAGE requiere PIPE_TYPE_MESSAGE.
- PIPE_WAIT y PIPE_NOWAIT, indican si las operaciones de lectura sobre la tubería bloquearán el proceso lector en caso de que se encuentre vacía o no.

El parámetro nMaxInstances determina el número máximo de instancias de la tubería, es decir, el número máximo de clientes simultáneos. cbOutBuf y cbInBuf indican los tamaños en bytes de los *buffer*s utilizados para la tubería. Un valor de cero representa los valores por defecto. dwTimeOut indica el tiempo de bloqueo, en milisegundos, de la función WaitNamedPipe, explicada más adelante. El último argumento representa los atributos de seguridad asociados a la tubería. La función devuelve el manejador asociado a la tubería (recuerde que en Windows las tuberías con nombre sólo disponen de un manejador).

◙ HANDLE **CreateFile**(LPCSTR lpFileName, DWORD dwDesiredAccess,DWORD dwShareMode,
LPVOID lpSecurityAttributes, DWORD CreationDisposition,
DWORD dwFlagsAndAttributes, HANDLE hTemplateFile);

La apertura de una tubería con nombre es una operación que típicamente realizan los clientes en aplicaciones de tipo cliente/servidor. Para abrir una tubería con nombre se recurre al servicio CreateFile, que se utiliza en Windows para crear ficheros. Su efecto es la creación, o apertura, de un fichero con nombre lpFileName. El resto de parámetros se describe con mayor detalle en la sección "5.16.3 Servicios Windows para ficheros". Este servicio se utiliza para abrir una tubería existente de la siguiente forma:

HandlePipe = **CreateFile**(PipeName, GENERIC_READ | GENERIC_WRITE, 0, NULL,
OPEN_EXISTING , FILE_ATTRIBUTE_NORMAL, NULL);

Con la llamada anterior se abrirá, para lectura y escritura, una tubería de nombre PipeName. La llamada fallará si la tubería con nombre todavía no se ha creado o todas sus instancias están ocupadas.

◙ BOOL **WaitNamedpipe**(PipeName, DWORD dwTimeOut);

Relacionado con la apertura de una tubería con nombre, se encuentra también el servicio WaitNamedPipe. Esta llamada bloquea a un proceso si todas las instancias de la tubería están ocupadas (normalmente por otros clientes). Si dwTimeOut es NULL, se utilizará el tiempo de expiración especificado en CreateNamedPipe. Los posibles valores para este parámetro son:

- NMPWAIT_NOWAIT, devuelve inmediatamente si el servidor no está listo para recibir mensajes.
- NMPWAIT_WAIT_FOREVER, bloquea el proceso cliente hasta que el servidor esté listo para recibir mensajes.
- NMPWAIT_USE_DEFAULT_WAIT, utiliza el tiempo por defecto especificado en CreateNamedPipe.

◙ BOOL **CloseHandle** (HANDLE hfile);

Este servicio cierra el manejador asociado a la tubería con o sin nombre. La llamada resta uno al contador de instancias asociado al objeto. Si el contador se hace 0 la tubería se destruye. Devuelve TRUE en caso de éxito o FALSE en caso de error.

◙ BOOL **ReadFile** (HANDLE hFile, LPVOID lpbuffer, DWORD nBytes,
LPDWORD lpnBytes, LPOVERLAPPED lpOverlapped);

Este servicio se utiliza para leer datos de una tubería. El servicio es el mismo que ofrece Windows para leer de ficheros El parámetro hFile es el manejador asociado a la tubería que se utiliza para leer. lpbuffer especifica el *buffer* de usuario donde se van a situar los datos leídos de la tubería. El argumento nBytes indica el número de bytes que se desean leer. En lpnBytes se almacena el número de datos realmente leídos. El último argumento es una estructura que se utiliza para indicar la posición de la cual se quiere leer. En el caso de tuberías se utiliza NULL. La función devuelve TRUE si se ejecutó con éxito o FALSE en caso contrario.

◙ BOOL **WriteFile** (HANDLE hFile, LPVOID lpbuffer, DWORD nBytes,
LPDWORD lpnBytes, LPOVERLAPPED lpOverlapped);

Este servicio permite escribir datos en una tubería. Al igual que con las lecturas, Windows utiliza el mismo servicio que el empleado para escribir en ficheros.El parámetro hFile es el manejador asociado a la tubería que se utiliza para escribir. lpbuffer especifica el *buffer* de usuario donde se encuentran los datos que se quieren escribir en la tubería. El argumento nBytes indica el número de bytes que se desean escribir. En lpnBytes se almacena el número de datos realmente escritos. El último argumento es una estructura que se utiliza para indicar la posición de la cual se quiere escribir. En el caso de tuberías se utiliza NULL. La función devuelve TRUE si se ejecutó con éxito o FALSE en caso contrario.

Otros servicios

Windows especifica además una serie de servicios que facilitan el desarrollo de aplicaciones cliente/servidor utilizando tuberías con nombre. Entre ellos, se pueden citar: `TransactNamedPipe`, utilizado por los clientes para enviar una petición a un servidor y esperar una respuesta de él, y. `ConnectNamedPipe`, que bloquea al servidor hasta que un cliente establezca una conexión con él. El servicio `DisconnectNamedPipe` permite al servidor desconectarse de un cliente.

Ejecución de mandatos con tuberías

En esta sección se va a implementar el mandato `dir | sort`, similar al mandato `ls | sort` que se desarrolló en la sección 6.9.1 "Soporte hardware para la sincronización". Los pasos realizados por este programa son muy similares a los que se llevaron a cabo en dicho programa. Estos pasos son:

- El proceso crea una tubería sin nombre mediante `CreatePipe`.
- Se crea un proceso hijo, que ejecutará el mandato `dir`, con su salida estándar redirigida a la salida de la tubería.
- Se crea un proceso hijo, que ejecutará el mandato `sort`, con su entrada estándar redirigida a la salida de la tubería.
- Se espera a la terminación de los dos procesos.

El programa 6.21 muestra el código del programa que ejecuta `dir|more`.

Programa 6.21 Ejecuta el mandato `dir | more`.

```
#include <windows.h>
#include <stdarg.h>
#include <stdio.h>

int main (int argc, LPTSTR argv [])
{
   HANDLE hRead, hWrite;
   SECURITY_ATTRIBUTES Apipe =
      {sizeof(SECURITY_ATTRIBUTES), NULL, TRUE);
   STARTUPINFO info;
   PROCESS_INFORMATION pi1, pid2;

   /* se crea el pipe */
   CreatePipe(&hRead, &hWrite, &Apipe, 0);
   /* Se prepara la redirección para el primer proceso */
   info.hStdInput = GetStdHandle(STD_INPUT_HANDLE);
   info.hStdOutput = hWrite;
   info.dwFlags = STARTF_USESSTDHANDLES;
   CreateProcess(NULL, "dir", NULL, NULL,
                 TRUE /* hereda manejadores */, 0, NULL, NULL,
                 &info, &pid1);
   CloseHandle(pid1.hThread);

   /* cierra el pipe para escritura */
   CloseHandle(hWrite);

   /* redirige la entrada estándar para el segundo proceso */
   info.hStdInput = hRead;
   info.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);
   info.dwFlags = STARTF_USESSTDHANDLES;
      CreateProcess(NULL, "sort", NULL, NULL,
                 TRUE /* hereda manejadores */, 0, NULL, NULL,
                 &info, &pid2);
   CloseHandle(pid2.hThread);
   CloseHandle(hRead);
   /* se espera la terminación de los dos procesos */
   WaitForSingleObject(pid1.hProcess, INFINITE);
   CloseHandle(pid1.hProcess);

   WaitForSingleObject(pid2.hProcess, INFINITE);
   CloseHandle(pid2.hProcess);

   return 0;
```

```
}
```

Aplicación cliente-servidor con tuberías con nombre

En esta sección se desarrolla la aplicación cliente-servidor de la sección 6.9.4 "Sincronización dentro del sistema operativo" utilizando tuberías con nombre. El proceso servidor crea una tubería con nombre y a continuación se bloquea (llamada `ConnectNamedPipe`) esperando a que un cliente establezca una conexión con él. El programa 6.22 muestra el código del proceso servidor multithread con tuberías con nombre. La estructura `struct peticion` utilizada en la sección 6.9.4 "Sincronización dentro del sistema operativo" no necesita en este caso el nombre de la cola de cliente.

Programa 6.22 Proceso servidor multitrhread con tuberías con nombre.

```
#include "mensaje.h"
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

int main (int argc, LPTSTR argv [])
{
   DWORD IdThread;
   HANDLE hPipe, hThread;
   int bufsize;

   /* el tamaño del buffer asociado a la tubería es el de una
      petición */
   bufsize = sizeof(struct peticion);

   /* En cada vuelta del bucle se crea una instancia de la tubería se espera la conexión de un cliente y se lanza un
      thread para atenderlo */
   for(;;) {
      hPipe = CreateNamedPipe(
         \\.\pipe\mituberia,                                  /* nombre */
         PIPE_ACCESS_DUPLEX,                                  /* lectura y escritura */
         PIPE_TYPE_BYTE | PIPE_READMODE_BYTE | PIPE_WAIT,
         PIPE_UNLIMITED_INSTANCES,                            /* max. instancias */
         bufsize, bufsize,                                    /* buffer de escritura y lectura */
         INFINITE,                                            /* espera en el cliente */
         NULL);                                               /* sin atributos de seguridad */

      if (hPipe == NULL) {
         printf ("Error al crear la tuberia. Error: %x\n",      GetLastError ());
         return 1;
      }
      if (ConnectNamedPipe(hPipe, NULL))
      {
         /* se crea el thread para atender la petición */
         hThread = CreateThread (
                     NULL,                                    /* sin atributos de seguridad */
                     0,                                       /* pila por defecto */
                     (LPTHREAD_START_ROUTINE) tratar_mensaje,
                     (LPVOID) hPipe,
                     0,                                       /* no suspendido */
                     &IdThread);

         if (hThread == NULL) {
            printf ("Error al crear el thread. Error: %x\n",    GetLastError ());
            return 1;
         }
      } else    /* fallo en la conexión */
         printf ("Error en la conexión. Error: %x\n",   GetLastError ());
         return 1;
      }

void tratar_mensaje (LPVOID lpvParam) {
   struct peticion mess;
```

```
    HANDLE hPipe;
    int exito, nrw;
    int resultado;

    hPipe = (HANDLE) lpvParam;
    exito = ReadFile (hPipe, (char *)&mess, sizeof(struct peticion), &nrw, NULL);
    if (exito == FALSE || nrw != sizeof(struct peticion)) {
        printf ("Error al leer. Error: %x\n",   GetLastError ());
        return;
    }

    /* ejecutar la peticion y preparar la respuesta */
    if (mess.operacion == SUMA)
        resultado = mess.operando_A + mess.operando_B;
    else
        resultado = mess.operando_A * mess.operando_B;

    exito = WriteFile (hPipe, (char *)&resultado, sizeof(int), &nrw, NULL);
    if (exito == FALSE || nrw != sizeof(struct peticion)) {
        printf ("Error al escribir. Error: %x\n",      GetLastError ());
        return;
    }

    /* desconectar la tubería y cerrarla */
    DisconnectNamedPipe(hPipe);
    CloseHandle(hPipe);
}
```

El programa 6.23 muestra el código del proceso cliente. En este caso el cliente intenta abrir la tubería del servidor y espera a que haya instancias libres de la misma.

Programa 6.23 Proceso cliente utilizando tuberías con nombre.

```
#include <stdio.h>
#include <windows.h>
#include “mensaje.h”

int main (int argc, LPTSTR argv [])
{
    HANDLE hPipe;
    struct peticion mess;
    int resultado, exito, nrw;

    /* intenta abrir la tubería del servidor */
    while (1)   {
        hPipe = CreateFile (“\\.\pipe\mituberia”, GENERIC_READ|GENERIC_WRITE, 0, NULL,
                        OPEN_EXISTING, 0, NULL);

        /* salir del bucle si se ha conectado */
        if (hPipe != INVALID_HANDLE_VALUE)
            break;

        /* Falla cuando el error es distinto de ERROR_PIPE_BUSY */
        if (GetLastError () != ERROR_PIPE_BUSY)
            printf ("Error al abrir la tuberia. Error: %x\n",       GetLastError ());
            return 1;
        }

        /* si todas las instancias de la tubería están ocupadas
            esperar 1 segundo */
        WaitNamedPipe("\\.\pipe\mituberia", 1000);
    }
    /* se ha establecido la conexión */
    mess.operacion = SUMA;
    mess.operando_A = 1000;
```

```
    mess.operando_B = 4000;

    exito = WriteFile(hPipe, (char *)&mess, sizeof(struct peticion), &nrw, NULL);

    if (exito == FALSE || nrw != sizeof(struct peticion)) {
        printf ("Error al escribir. Error: %x\n",GetLastError ());
        return 1;
    }

    /* espera la respuesta*/
    exito = ReadFile(hPipe, (char *)&resultado, sizeof(int),&nrw, NULL);

    if (exito == FALSE || nrw != sizeof(struct peticion)) {
        printf ("Error al leer. Error: %x\n", GetLastError ());
        return 1;
    }
    printf("El resultado es %d\n!, resultado);
    CloseHandle(hPipe);
    return 0;
}
```

6.11.2. Secciones críticas en Windows

Las secciones críticas son un mecanismo de sincronización especialmente concebido para resolver el acceso a secciones de código que deben ejecutarse en exclusión mutua.

Las secciones críticas son objetos que se crean y se borran pero no tienen manejadores asociados. Sólo se pueden utilizar por los *threads* creados dentro de un proceso. Los servicios utilizados para tratar con secciones críticas son los siguientes:

- VOID **InitializeCriticalSection**(LPCCRITICAL_SECTION lpcsCriticalSection);
  VOID **DeleteCriticalSection**(LPCCRITICAL_SECTION lpcsCriticalSection);
  VOID **EnterCriticalSection**(LPCCRITICAL_SECTION lpcsCriticalSection);
  VOID **LeaveCriticalSection**(LPCCRITICAL_SECTION lpcsCriticalSection);

Las dos primeras funciones se utilizan para crear y borrar secciones críticas. Las dos siguientes sirven para entrar y salir de la sección crítica.

El acceso en exclusión mutua a una sección crítica se resuelve de forma muy sencilla utilizando este mecanismo de Windows. En primer lugar se deberá crear una sección crítica e iniciarla:

```
    LPCCRITICAL_SECTION SC;
InitializeCriticalSection(&SC);
```

Siempre que se desee acceder a una sección crítica deberá utilizarse el siguiente fragmento de código:

```
    EnterCriticalSection(&SC);
< Código de la sección crítica >
LeaveCriticalSection (&SC);
```

Cuando deje de accederse a la sección crítica se deberá destruir utilizando:

```
    DeleteCriticalSection(&SC);
```

6.11.3. Semáforos en Windows

En Windows los semáforos tienen asociado un nombre. A continuación se indican los servicios de Windows para trabajar con semáforos.

- HANDLE **CreateSemaphore**(LPSECURITY_ATTRIBUTES lpsa,
  LONG cSemInitial, LONG cSemMax, LPCTSTR lpszSemName);

Permite crear un semáforo. Los semáforos en Windows se manipulan, como el resto de los objetos, con manejadores. El primer parámetro especifica los atributos de seguridad asociados al semáforo. El argumento `cSemInitial` indica el valor inicial del semáforo y `cSemMax` el valor máximo que puede tomar. El nombre del semáforo viene dado por el parámetro `lpszSemName`. La llamada devuelve un manejador de semáforo válido o NULL en caso de error.

- HANDLE **OpenSemaphore**(LONG dwDesiredAccess, LONG BinheritHandle,
  lpszName SemName);

El servicio se utiliza para abrir un semáforo, una vez creado. El parámetro `dwDesiredAccess` puede tomar los siguientes valores:

- `SEMAPHORE_ALL_ACCESS`, total acceso al semáforo.
- `SEMAPHORE_MODIFY_STATE`, permite la ejecución de la función `ReleaseSemaphore`.
- `SYNCHRONIZE`, permite el uso del semáforo para sincronización, es decir, en las funciones de espera (*wait*).

El segundo argumento indica si se puede heredar el semáforo a los procesos hijos. Un valor de `TRUE` permite heredarlo. `SenName` indica el nombre del semáforo que se desea abrir. La función devuelve un manejador de semáforo válido en caso de éxito o `NULL` en caso de error.

◙ BOOL **CloseHandle**(HANDLE hObject);

Este servicio se utiliza para cerrar un semáforo. Si la llamada termina correctamente, se cierra el semáforo. Si el contador del manejador es cero, se liberan los recursos ocupados por el semáforo. Devuelve `TRUE` en caso de éxito o `FALSE` en caso de error.

Operación wait

Se utiliza el siguiente servicio de Windows.

◙ DWORD **WaitForSingleObject**(HANDLE hSem, DWORD dwTimeOut);

Este servicio se corresponde con la operación `wait` sobre un semáforo. Esta es el servicio general de sincronización que ofrece Windows. Cuando se aplica a un semáforo implementa la función `wait`. El parámetro `dwTimeOut` debe tomar en este caso valor `INFINITE`.

◙ BOOL **ReleaseSemaphore**(HANDLE hSemaphore, LONG cReleaseCount,
LPLONG lpPreviousCount);

Este servicio es el que se utiliza para implementar la operación `signal` sobre un semáforo. Incrementa el semáforo en `cReleaseCount`. Para que tenga el comportamiento de la función `signal`, debe llamarse de la siguiente forma:

ReleaseSemaphore(hSemaphore, 1, NULL);

Productor-consumidor con memoria compartida

En esta sección se desarrolla una solución al problema del productor-consumidor similar al descrito en la sección 6.9.2 "Espera activa", utilizando semáforos y memoria compartida entre procesos. Para que los procesos puedan compartir memoria, se va a utilizar ficheros proyectados en memoria que siguen el mismo comportamiento que los objetos de memoria compartida de UNIX.

En la solución que se presenta a continuación, similar a la presentada en la sección 6.11.2 "Secciones críticas en Windows", el proceso productor se encarga de:

- Crear los semáforos (`CreateSemaphore`).
- Crear un fichero (`CreateFile`). Este fichero se utilizará como segmento de memoria compartida entre los procesos.
- Asignar espacio al fichero creado (`SetFileSize`).
- Proyectar el fichero en memoria (`CreateFileMapping` y `MapViewOfFile`).
- Acceder al segmento de memoria compartida descrito por el fichero para insertar los elementos que produce.
- Desproyectar el fichero cuando se ha finalizado el trabajo (`UnmapViewOfFile`).
- Cerrar en último lugar el fichero (`CloseHandle`) y borrarlo (`DeleteFile`).

Los pasos que debe realizar el proceso consumidor son los siguientes:

- Abrir los semáforos (`OpenSemaphore`).
- Abrir el fichero a utilizar como segmento de memoria compartida (`CreateFile`).
- Proyectar el fichero en su espacio de direcciones (`CreateFileMapping` y `MapViewOfFile`).
- Acceder al segmento de memoria compartida descrito por el fichero para eliminar los elementos que produce el productor.
- Desproyectar el fichero del espacio de direcciones (`UnmapViewOfFile`).
- Cerrar el fichero (`CloseHandle`).

El programa 6.24 muestra el código que ejecuta el proceso productor.

Programa 6.24 Código del proceso productor utilizando semáforos Windows y Memoria compartida.

```
#include <windows.h>
#include <stdio.h>

#define MAX_buffer          1024          /* tamaño del buffer */
```

```
#define DATOS_A_PRODUCIR      100000      /* datos a producir */

int main (int argc, LPTSTR argv [])
{
   HANDLE huecos, elementos;
   HANDLE hIN, hInMap;
   int *buffer;

   huecos = CreateSemaphore(NULL, MAX_buffer, MAX_buffer, “HUECOS”);
   elementos = CreateSemaphore(NULL, 0, MAX_buffer, “ELEMENTOS”);
   if (huecos == NULL || elementos == NULL) {
      printf ("Error al crear los semáforos. Error: %x\n",     GetLastError ());
      return 1;
   }

   /* Crear el fichero que se utilizará como segmento de memoria compartida y asignar espacio */
   hIn = CreateFile(“ALMACEN”, GENERIC_WRITE, 0, NULL,
                    CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
   SetFileSize(hIN, MAX_buffer * sizeof(int));

   /* proyectar el fichero en memoria */
   hInMap = CreateFileMapping(hIN, NULL, PAGE_WRITEONLY, 0, 0, NULL);
   buffer = (int *) MapViewOfFile(hInMap, FILE_MAP_WRITE, 0, 0, 0);

   if (buffer = NULL) {
      printf ("Error al proyectar el fichero. Error: %x\n",GetLastError ());
      return 1;
   }

   productor(buffer);

   UnmapViewOfFile(buffer);
   CloseHandle(hInMap);
   CloseHandle(hIn);
   CloseHandle(huecos);
   CloseHandle(elementos);
   DeleteFile(“ALMACEN”);
}

/* función productor */
void productor(int *buffer) {
   int dato;              /* dato a producir*/
   int posicion = 0;      /* posición donde insertar el elemento */
   int j;

   for (j=0; j < DATOS_A_PRODUCIR; j++){
      dato = j;
      WaitForSingleObject(huecos, INFINITE);/* un hueco menos */
      buffer[posicion]= dato;
      posicion = (posicion+1) % MAX_buffer;  /* nueva posición */
      ReleaseSemaphore(elementos, 1, NULL);
   }

   return;
}
```

El programa 6.25 muestra el código del proceso consumidor.

Programa 6.25 Código del proceso consumidor utilizando semáforos y memoria compartida entre procesos en Windows.

```
#include <windows.h>
#include <stdio.h>
#define MAX_buffer            1024            /* tamaño del buffer */
#define DATOS_A_PRODUCIR      100000          /* datos a producir */
```

```
int main (int argc, LPTSTR argv [])
{
    HANDLE huecos, elementos;
    HANDLE hIN, hInMap;
    int *buffer;

    huecos = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, “HUECOS”);
    elementos = OpenSemaphore (SEMAPHORE_ALL_ACCESS, FALSE, “ELEMENTOS”);
    if (huecos == NULL || elementos == NULL) {
        printf ("Error al crear los semáforos. Error: %x\n", GetLastError ());
        return -1;
    }
    /* Abrir el fichero que se utilizará como segmento de memoria compartida y asignar espacio */
    hIn = CreateFile(“ALMACEN”, GENERIC_READ, 0, NULL,
                OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

    /* proyectar el fichero en memoria */
    hInMap = CreateFileMapping(hIN, NULL, PAGE_READEONLY, 0, 0, NULL);
    buffer = (int *) MapViewOfFile(hInMap, FILE_MAP_READ, 0, 0, 0);

    if (buffer = NULL) {
        printf ("Error al proyectar el fichero. Error: %x\n",GetLastError ());
        return -;
    }

    consumidor(buffer);
    UnmapViewOfFile(buffer);
    CloseHandle(hInMap);
    CloseHandle(hIn);
    CloseHandle(huecos);
    CloseHandle(elementos);
}
/* función consumidor */
void consumidor(int *buffer) {
    int dato;           /* dato a producir*/
    int posicion = 0;   /* posición que indica el elemento a extraer */
    int j;

    for (j=0; j < DATOS_A_PRODUCIR; j++){
        WaitForSingleObject(elementos, INFINITE);       /* un elemento menos */
        dato = buffer[posicion];
        posicion = (posicion+1) % MAX_buffer;           /* nueva posición */
        ReleaseSemaphore(huecos, 1, NULL);              /* un hueco máx */
    }
    return;
}
```

6.11.4. *Mutex* y eventos en Windows

Los *mutex*, al igual que los semáforos, son objetos con nombre. Los *mutex* sirven para implementar secciones críticas. La diferencia entre las secciones críticas y los *mutex* de Windows radica en que las secciones críticas no tienen nombre y sólo se pueden utilizar entre *threads* de un mismo proceso. Un *mutex* se manipula utilizando manejadores.

A continuación se indican los servicios utilizados en Windows para trabajar con *mutex*.

- HANDLE **Create**mutex(LPSECURITY_ATTRIBUTES lpsa, BOOL fInitialOwner, LPCTSTR lpszmutexName);

Este servicio se utiliza para crear un *mutex*. El servicio crea un *mutex* con atributos de seguridad `lpsa`. Si `fInitialOwner` es `TRUE`, el propietario del *mutex* será el *thread* que lo crea. El nombre del *mutex* viene dado por el tercer argumento. En caso de éxito la llamada devuelve un manejador de *mutex* válido y en caso de error `NULL`.

- HANDLE **Open**mutex(LONG dwDesiredAccess, LONG BineheritHandle, lpszName SemName);

Este servicio se utiliza para abrir un *mutex*. Los parámetros y su comportamiento son similares a los de la llamada `OpenSemaphore`.

- BOOL **CloseHandle**(HANDLE hObject);

Este servicio se utiliza para cerrar un *mutex*. Si la llamada termina correctamente, se cierra el *mutex*. Si el contador del manejador es cero, se liberan los recursos ocupados por el *mutex*. Devuelve `TRUE` en caso de éxito o `FALSE` en caso de error.

- DWORD **WaitForSingleObject**(HANDLE hmutex, DWORD dwTimeOut);

Este servicio se utiliza para realizar la operación `lock` sobre un *mutex*. Esta es el servicio general de sincronización que ofrece Windows. Cuando se aplica a un *mutex* implementa la función `lock`. El parámetro `dwTimeOut` debe tomar en este caso valor `INFINITE`.

- BOOL **Release**mutex(HANDLE hmutex);

La operación `unlock` sobre un *mutex* se implementa con este servicio. Los eventos en Windows son comparables a las variables condicionales, es decir, se utilizan para notificar que alguna operación se ha completado o que ha ocurrido algún suceso. Sin embargo, las variables condicionales se encuentran asociadas a un *mutex* y los eventos no.

Los **eventos** en Windows se clasifican en manuales y automáticos. Los primeros se pueden utilizar para desbloquear varios *threads* bloqueados en un evento. En este caso el evento permanece en estado de notificación y debe eliminarse de forma manual. Los automáticos se utilizan para desbloquear un único *thread*, es decir, el evento notifica a un único *thread* y a continuación deja de estar en este estado de forma automática. UNIX no dispone de enventos manuales. En esta sección sólo se tratarán los automáticos, ya que éstos son los más similares a las variables condicionales descritas en la sección 6.7.3 "Mutex y variables condicionales". A continuación se indican los principales servicios para tratar con eventos.

- HANDLE **CreateEvent**(LPSECURITY_ATTRIBUTES lpsa, BOOL fManualReset,
  BOOL fInitialState, LPTCSTR lpszEventName);

Este servicio crea un evento de nombre `lpszEventName` con atributos de seguridad `lpsa`. Si `fManualReset` es `TRUE` el evento será manual, en caso contrario será automático. Si `fInitialState` es `TRUE` el evento se creará en estado de notificación, en caso contrario no. La llamada devuelve un manejador de evento válido en caso de éxito o `NULL` si se produjo algún fallo.

Esta función es similar al servicio `pthread_cond_init` de UNIX. Para emular el comportamiento de esta llamada se debería invocar a `CreateEvent` de la siguiente forma:

CreateEvent(NULL, FALSE, FALSE, name);

Siendo `name` el nombre dado al evento.

- BOOL **CloseHandle**(HANDLE hObject);

El servicio permite destruir un evento. Si la llamada termina correctamente, se cierra el evento. Si el contador del manejador es cero, se liberan los recursos ocupados por el evento y se destruye. Devuelve `TRUE` en caso de éxito o `FALSE` en caso de error.

- DWORD **WaitForSingleObject**(HANDLE hEvent, DWORD dwTimeOut);

Este servicio cuando se aplica a un evento implementa la función `c_wait` sobre una variable condicional. El parámetro `dwTimeOut` debe tomar en este caso valor `INFINITE`.

- BOOL **SetEvent**(HANDLE hEvent);
  BOOL **PulseEvent**(HANDLE hEvent);

Estos servicios se utilizan para notificar un evento. El primero notifica un evento y despierta a un único *thread* bloqueado en `WaitForSingleObject`**.** La segunda función notifica un evento y despierta a todos los *threads* bloqueados en él. Cuando se notifica un evento automático, después de la llamada y una vez desbloqueado el *thread* o *threads* correspondientes, el evento pasará a estado de no notificación. Si ningún *thread* está esperando el evento cuando se ejecuta alguna de estas llamadas, el evento permanecerá señalizado hasta que algún *thread* ejecute una llamada de espera.

Prestaciones 6.1. Dentro de un mismo proceso, es más eficiente utilizar secciones críticas que *mutex*. Esto se debe a que las secciones críticas ejecutan enteramente dentro del espacio de direcciones del proceso y no suponen una llamada al sistema operativo, mientras que los *mutex* sí.

Secciones críticas con eventos

El problema de la sección crítica puede resolverse utilizando eventos. Para ello ha de crearse un evento cuyo estado inicial sea notificado:

```
HANDLE hEvent;
hEvent = CreateEvent(NULL,    /* sin atributos de seguridad */
                     FALSE,   /* evento automático */
                     TRUE,    /* evento inicialmente notificado */
                     "evento";) /* nombre del evento */
```

Para acceder a la sección crítica basta con protegerla de la siguiente manera:

```
WaitForSingleObject(hEvent, INFINITE);
<Sección crítica >
SetEvent(hEvent);
```

Variables condicionales utilizando eventos

Como se dijo anteriormente, los eventos de Windows son similares a las variables condicionales, sin embargo, los eventos no se encuentran asociados a ningún *mutex*, como ocurre con las variables condicionales. A continuación se va a describir cómo utilizar los *mutex* y los eventos de Windows para emular el comportamiento de los *mutex* y las variables condicionales.

Para implementar el siguiente fragmento de código (típico cuando se utilizan *mutex* y variables condicionales):

```
lock(m);
/* código de la sección crítica */
while (condición == FALSE)
   c_wait(c, m);
/* resto de la sección crítica */
unlock(m);
```

En Windows debe crearse un *mutex* y un evento en estado inicial de no notificación. Una vez creados, el código anterior se convertirá en el siguiente:

```
WaitForSingleObject(mutex, INFINITE);
/* código de la sección crítica */
while (condicion == FALSE) {
   Releasemutex(mutex);    /* se libera el mutex */
   WaitForMultipleObjects(2, hHandles, TRUE, INFINITE);
}
/* resto de la sección crítica */
Releasemutex(mutex);
```

donde `hHandles` es un vector:

```
HANDLE hHanles[2];
hHanles[0] = mutex;
hHanles[1] = evento;
```

Recuérdese que una operación `c_wait` sobre una variable condicional bloquea el proceso y libera de forma atómica el *mutex* para permitir que otros procesos puedan ejecutar. Utilizando eventos, es necesario en primer lugar liberar la sección crítica (`Releasemutex`) y esperar a continuación por el *mutex* y el evento (`WaitForMultipleObjects`).

Para implementar la siguiente estructura:

```
lock(m);
/* código de la sección crítica */
/* se modifica la condición y ésta se hace TRUE */
condición = TRUE;
c_signal(c);
unlock(m);
```

Se debe utilizar el siguiente fragmento de código:

```
WaitForSingleObject(mutex, INFINITE);
/* código de la sección crítica */
/* se modifica la condición y esta se hace TRUE */
condición = TRUE;
SetEvent(evento);
Releasemutex(mutex);
```

Sólo cuando se ejecute el último `Releasemutex` se desbloqueará un proceso bloqueado en `WaitForMultipleObjects` (sobre el *mutex* y el evento). Cuando el proceso bloqueado en esta función despierte, continuará la ejecución con el *mutex* adquirido y el evento en estado de no notificación.

6.11.5. Mailslots

Los *mailslots* son parecidos a las tuberías con nombre de Windows y a las colas de mensajes de UNIX. Sus principales características son:

- Un mailslot es un pseudofichero unidireccional que reside en memoria. El tamaño máximo de los mensajes que se pueden enviar a un mailslot es de 64 KiB.

- Pueden tener múltiples lectores y múltiples escritores. Sin embargo, sólo los procesos que creen el mailslot o lo hereden pueden leer mensajes de él.
- Pueden utilizarse entre procesos que ejecutan en máquinas de un mismo dominio de red.

A continuación se indican las principales funciones relacionadas con los mailslots.

◙ HANDLE **CreateMailslot**(LPCTSTR lpName, DWORD MaxMessSize, DWORD lReadTimeout, LPSECUTIRY_ATTRIBUTES lpsa);

Este servicio crea un mailslot de nombre `lpName`. `MaxMessSize` especifica el tamaño máximo del mensaje. Un valor de 0 indica que el mensaje puede ser de cualquier tamaño. El parámetro `lReadTimeout` especifica el tiempo en milisegundos que una operación de lectura del mailslot bloqueará al proceso hasta obtener un mensaje. El valor `MAILSLOT_WAIT_FOREVER` bloquea el proceso que lee hasta que obtenga un mensaje.

El nombre del mailslot debe seguir la siguiente convención:

\\.\mailslot\nombre

La función devuelve un manejador de mailslot válido si se ejecutó con éxito o `INVALID_HANDLE_VALUE` en caso contrario.

Para abrir un mailslot se utiliza la función **`CreateFile`** ya descrita anteriormente. El formato del nombre debe ser de la siguiente forma:

- `\\.\mailslot\nombre`, para abrir un mailslot local.
- `\\.\maquina\mailslot\nombre`, para abrir un mailslot creado en una máquina remota.
- `\\.\nombredominio\mailslot\nombre`, para abrir un mailslot creado en un dominio.

Una aplicación debe especificar `FILE_SHARE_READ` para abrir un mailslot existente.

Para cerrar un mailslot se utiliza `CloseHandle`. Cuando el mailslot deja de tener manejadores abiertos, se destruye.

Las operaciones de lectura y escritura en un mailslot se realiza utilizando los servicios **`ReadFile`** y **`WriteFile`** respectivamente.

◙ BOOL **SetMailSlotInfo**(HANDLE hMailslot, DWORD lReadTimeout);

Este servicio permite asignar atributos a un mailslot. Los únicos atributos que se pueden modificar sobre un mailslot ya creado es el tiempo de bloqueo de la operación `ReadFile`.

◙ BOOL **GetMailslotInfo**(HANDLE hMailslot, LPDWORD lpMaxMess, LPDWORD lpNextSize, LPDWORD lpMessCount, LPDWORD lpReadTimeout);

Permite obtener los atributos asoaciados a un mailslot. En `lpMaxMess` se almacenará el tamaño máximo del mensaje, en `lpNextSize` el tamaño del siguiente mensaje. Si no hay ningún mensaje en el mailslot, el parámetro anterior tomará el valor `lpNextSize`. El argumento `lpMessCount` almacenará el número de mensajes pendientes de lectura. En el último argumento se almacenará el tiempo de bloqueo de la función `ReadFile`.

El desarrollo de una aplicación cliente-servidor utilizando mailslots es similar al uso de colas de mensajes UNIX y se propone como ejercicio al lector al final del capítulo.

Como recapitulación, las tablas 6.6 y 6.7 presentan las características de los mecanismos de comunicación y sincronización respectivamente de Windows.

Tabla 6.6 *Características de los mecanismos de comunicación de Windows.*

Mecanismo	Nombrado	Identificador	Almacenamiento	Flujo de datos
Tubería	Sin nombre Con nombre	Manejador	Si	Undireccional Bidireccional
Mailslot	Nombre dentro de un dominio de Windos	Manejador	Si	Bidireccional

Tabla 6.7 *Características de los mecanismos de sincronización de Windows.*

Mecanismo	Nombrado	Identificador
Sección crítica	Sin nombre	Variable de tipo sección crítica
Tubería	Sin nombre Con nombre	Manejador
Semáforo	Con nombre	Manejador
mutex	Con nombre	Manejador
Evento	Con nombre	Manejador
Mailslot	Con nombre	Manejador

6.12. LECTURAS RECOMENDADAS Y BIBLIOGRAFÍA

El tema de la comunicación y sincronización entre procesos se analiza en multitud de libros. Uno de los primeros en tratar los algoritmos para resolver el problema de la exclusión mutua fue Dijkstra [Dijkstra 1965]. En [Ben Ari 1990] se realiza un estudio sobre la concurrencia de procesos y lor mecanismos de comunicación y sincronización. El estudio de la comunicación y sincronización entre procesos puede completarse también en [Stallings 2005] y [Silberschatz 2004]. En [Beck 1998], [Daniel 2005] y [Love 2005] se analiza los mecanismos de comunicación y sincronización en Linux, en [Solomon 2004] y se estudia la implementación en Windows NT y [McKusick 1996], [Bach 1986] [Mauro 2000] y la implementación en diversas versiones de UNIX.

6.13. EJERCICIOS

1. Dos procesos se comunican a través de un fichero, de forma que uno escribe en el fichero y el otro lee del mismo. Para sincronizarse el proceso escritor envía una señal al lector. Proponga un esquema del código de ambos procesos. ¿Qué problema plantea la solución anterior?

2. El siguiente fragmento de código intenta resolver el problema de la sección crítica para dos procesos

```
P0 y P1.
while (turno != i)
;
SECCIÓN CRÍTICA;
turno = j;
```

La variable `turno` tiene valor inicial 0. La variable i vale 0 en el proceso P0 y 1 en el proceso P1. La variable j vale 1 en el proceso P0 y 0 en el proceso P1. ¿Resuelve este código el problema de la sección crítica?

3. ¿Cómo podría implementarse un semáforo utilizando una tubería?

4. Un semáforo binario es un semáforo cuyo valor sólo puede ser 0 ó -1. Indique cómo puede implementarse un semáforo general utilizando semáforos binarios.

5. Suponga un sistema que no dispone de *mutex* y variables condicionales. Muestre cómo pueden implementarse éstos utilizando semáforos.

6. Considérese un sistema en el que existe un proceso agente y tres procesos fumadores. Cada fumador está continuamente liando un cigarrillo y fumándolo. Para fumar un cigarrillo son necesarios tres ingredientes: papel, tabaco y cerillas. Cada fumador tiene reserva de un solo ingrediente distinto de los otros dos. El agente tiene infinita reserva de los tres ingredientes, poniendo cada vez dos de ellos sobre la mesa. De esta forma, el fumador con el tercer ingrediente puede liar el cigarrillo y fumárselo, indicando al agente cuando ha terminado, momento en que el agente pone en la mesa otro par de ingredientes y continúa el ciclo.
Escribir un programa que sincronice al agente y a los tres fumadores utilizando semáforos.

Resolver el mismo problema utilizando mensajes.

7. Implemente la aplicación cliente-servidor desarrollada en la sección 6.9.4 "Sincronización dentro del sistema operativo" utilizando los mailslots de Windows.

8. Modifique el programa 6.4 de manera que sean los escritores los procesos prioritarios.

9. Muestre cómo podrían implementarse los semáforos con nombre de UNIX en un sistema que no dispone de ellos pero sí dispone de colas de mensajes.

10. Resuelva el problema de los lectores-escritores utilizando las colas de mensajes de UNIX.

11. Resuelva el problema de los lectores-escritores usando los mailslot de Windows.

12. Una barbería está compuesta por una sala de espera, con *n* sillas, y la sala del barbero, que dispone de un sillón para el cliente que está siendo atendido. Las condiciones de atención a los clientes son las siguientes:

- Si no hay ningún cliente, el barbero se va a dormir.
- Si entra un cliente en la barbería y todas las sillas están ocupadas, el cliente abandona la barbería.
- Si hay sitio y el barbero está ocupado, se sienta en una silla libre.
- Si el barbero estaba dormido, el cliente le despierta.

Escriba un programa, que coordine al barbero y a los clientes utilizando colas de mensajes UNIX para la comunicación entre procesos.

13. Resuelva el problema 6.8 utilizando en este caso mailslots.

14. Resuelva el problema 6.8 utilizando las tuberías con nombre que ofrece Windows.

15. Un puente es estrecho y sólo permite pasar vehículos en un sentido al mismo tiempo. El puente sólo permite pasar un coche al mismo tiempo. Si pasa un coche en un sentido y hay coches en el mismo sentido que quieren pasar, entonces estos tienen prioridad frente a los del otro sentido (si hubiera alguno esperando). Suponga que los coches son los procesos y el puente el recurso. El aspecto que tendrá un coche al pasar por el lado izquierdo del puente sería:

```
entrar_izquierdo
pasar puente
salir izquierdo
```

y de igual forma con el derecho. Se pide escribir las rutinas `entrar_izquierdo` y `salir_izquierdo` usando como mecanismos

de sincronización los semáforos.

16. Resuelva el problema anterior utilizando *mutex* y variables condicionales.

APÉNDICE 1
RESUMEN DE LLAMADAS AL SISTEMA

En este apéndice se incluye una tabla comparativa de llamadas al sistema UNIX y Windows, agrupadas por temas. Además se incluye un breve comentario de cada llamada.

La tabla no incluye todas las llamadas al sistema, sino sólo aquéllas que se han utilizado en el libro para programar los ejemplos, que constituyen el subconjunto más significativo y de uso más frecuente.

Tema	UNIX	Windows	Comentarios
Señales	pause		Suspende proceso hasta recepción de señal.
Señales	kill		Manda una señal.
Señales	sigemptyset, sigfillset, sigaddset, sigdelset, sigismember		Manipulación de conjuntos de señales.
Señales	sigprocmask		Consulta o modifica la máscara de señales.
Señales	sigpending		Obtiene las señales que están pendientes de entregar.
Señales	sigaction		Gestión detallada de señales.
Señales	sigsetjmp, siglongjmp		Realizan saltos no locales.
Señales	sigsuspend		Especifica máscara y suspende proceso hasta señal.
Memoria	mmap	CreateFileMapping	Proyecta en memoria un fichero.
Memoria	mmap	MapViewOfFile	En Windows requiere utilizar dos funciones (CreateFileMapping y MapViewOfFile).
Memoria	munmap	UnmapViewOfFile	Desproyecta un fichero.
M. Compartida	shmget	CreateFileMapping	Crea o asigna un segmento de memoria compartida.
M. Compartida	shmat	MapViewOfFile	Proyecta un segmento de memoria compartida.
M. Compartida	shmdt	UnmapViewOfFile	Desproyecta un segmento de memoria compartida.
Bibliotecas	dlopen, dlsym, dlclose	LoadLibrary, GetProcAddress, FreeLibrary	Carga y montaje explícito de bibliotecas dinámicas.
Cerrojos de ficheros y E/S	fcntl (cmd = f_setlk, ...)	LockFile, LockFileEx	Establece un cerrojo a un fichero.
Cerrojos de ficheros y E/S	fcntl (cmd = f_setlk, ...)	UnlockFile, UnlockFileEx	Elimina un cerrojo de un fichero.
Procesos	fork () y exec ()	CreateProcess	Crea proceso (CreateProcess equivale a fork + exec).
Procesos	_exit	ExitProcess	Termina el proceso.
Procesos	getpid	GetCurrentProcess, GetCurrentProcessId	Obtiene identificador del proceso.
Procesos	wait, waitpid	GetExitCodeProcess	Obtiene información de proceso ya terminado.
Procesos	execl, execv, execle, execve, execlp,		Ejecuta un programa (no hay equivalente en Windows).

Tema	UNIX	Windows	Comentarios
	execvp		
Procesos	fork		Crea proceso duplicado (no hay equivalente en Windows).
Procesos	getppid		Obtiene id. del padre (en Windows no hay relación padre/hijo).
Procesos	getgid, getegid		Obtiene id. del grupo (en Windows no hay grupos de procesos).
Procesos	kill	TerminateProcess	Finaliza la ejecución de un proceso.
Procesos	waitpid	WaitForMultipleObjects (manejadores de procesos)	Espera la terminación de un proceso (en Windows de múltiples procesos).
Procesos	wait, waitpid	WaitForSingleObject (manejador de proceso)	Espera la terminación de un proceso.
Planificación	sched_setscheduler, nice	SetPriorityClass, SetThreadPriority	Controla aspectos de planificación de procesos y *threads*.
Planificación	pthread_yield	Sleep(0)	Cede el procesador.
Planificación	sched_setaffinity, sched_setaffinity	SetProcessAffinityMask, SetThreadAffinityMask	Establece la asignación de procesos y *threads* a procesadores.
Comunicación	close	CloseHandle (manejador de tubería)	Cierra una tubería.
Comunicación	mq_open	CreateFile (mailslot)	Abre una cola de mensajes en UNIX y un mailslot en Windows.
Comunicación	mq_open	CreateMailslot	Crea una cola de mensajes en UNIX y un mailslot en Windows.
Comunicación	mq_close	CloseHandle (manejador de mailslot)	Cierra una cola de mensajes en UNIX y un mailslot en Windows.
Comunicación	mq_send	WriteFile (manejador de mailslot)	Envía datos a una cola de mensajes en UNIX y a un mailslot en Windows.
Comunicación	mq_receive	ReadFile (manejador de mailslot)	Recibe datos de una cola de mensajes en UNIX y de un mailslot en Windows.
Comunicación	mq_unlink	CloseHandle (manejador de mailslot)	Borra una cola de mensajes en UNIX y un mailslot en Windows cuando deja de estar referenciado.
Comunicación	mq_getattr	GetMailslotInfo	Obtiene atributos de una cola de mensajes en UNIX y un mailslot en Windows.
Comunicación	mq_setattr	SetMailSlotInfo	Fija los atributos de una cola de mensajes en UNIX y un mailslot en Windows.
Comunicación	mkfifo	CreateNamedPipe	Crea una tubería con nombre.
Comunicación	pipe	CreatePipe	Crea una tubería sin nombre.
Comunicación	dup, dup2, fcntl	DuplicateHandle	Duplica un manejador de fichero.
Comunicación	read (tubería)	ReadFile (tubería)	Lee datos de una tubería.
Comunicación	write (tubería)	WriteFile (tubería)	Escribe datos en una tubería.
Comunicación	close	CloseHandle (tubería)	Cierra una tubería.
Threads	pthread_create	CreateThread	Crea un proceso ligero.
Threads	pthread_exit	ExitThread	Finaliza la ejecución de un proceso ligero.
Threads		GetCurrentThread	Devuelve el manejador del proceso ligero que ejecuta.
Threads		GetCurrentThreadId	Devuelve el identificador del proceso ligero que ejecuta.
Threads	pthread_join	GetExitCodeThread	Obtiene el código de finalización de un proceso ligero.
Threads		ResumeThread	Pone en ejecución un proceso ligero suspendido.
Threads		SuspendThread	Suspende la ejecución de un proceso ligero.

Tema	UNIX	Windows	Comentarios
Threads	pthread_join	WaitForSingleObject (manejador de proceso ligero)	Espera la terminación de un proceso ligero.
Threads		WaitForMulipleObject (manejadores de proceso ligero)	Espera la terminación de múltiples procesos ligeros en Windows.
Threads		GetPriorityClass	Devuelve la clase de prioridad de un proceso.
Threads	sched_getparam	GetThreadPriority	Devuelve la prioridad de un proceso ligero.
Threads		SetPriorityClass	Fija la clase de prioridad de un proceso.
Threads	sched_setparam	SetThreadPriority	Fija la prioridad de un proceso ligero.
Sincronización	pthread_cond_destroy	CloseHandle (manejador de evento)	Destruye una variable condicional en UNIX y un evento en Windows cuando deja de estar referenciado.
Sincronización	pthread_cond_init	CreateEvent	Inicia una variable condicional y un evento.
Sincronización	pthread_cond_broadcast	PulseEvent	Despierta los procesos ligeros bloqueados en una variable condicional o un evento.
Sincronización	pthread_cond_signal	SetEvent	Despierta un proceso ligero bloqueado en una variable condicional o evento.
Sincronización	pthread_cond_wait	WaitForSingleObject (manejador de evento)	Bloquea un proceso en una variable condicional o evento.
Semáforos	pthread_mutex_destroy	CloseHandle (manejador de mutex)	Destruye un *mutex*.
Semáforos	pthread_mutex_init	Createmutex	Inicia un *mutex*.
Semáforos	pthread_mutex_unlok	Releasemutex	Operación unlock sobre un *mutex*.
Semáforos	pthread_mutex_lock	WaitForSingleObject (manejador de mutex)	Operación lock sobre un *mutex*.
Semáforos	sem_open	CreateSemaphore	Crea un semáforo con nombre.
Semáforos	sem_init		Inicia un semáforo sin nombre.
Semáforos	sem_open	OpenSemaphore	Abre un semáforo con nombre.
Semáforos	sem_close	CloseHandle (manejador de semáforo)	Cierra un semáforo.
Semáforos	sem_post	ReleaseSemaphore	Operación signal sobre semáforo.
Semáforos	sem_wait	WaitForSingleObject (manejador de semáforo)	Operación wait sobre semáforo.
Manejo de Errores	errno	GetLastError	Almacena información sobre la última llamada al sistema.
Tiempo	time	GetSystemTime	Obtiene el tiempo de calendario.
Tiempo	localtime	GetLocalTime	Obtiene el tiempo de calendario en horario local.
Tiempo	stime	SetSystemTime	Establece la hora y fecha.
Tiempo	alarm	SetTimer	Establece un temporizador.
Tiempo	times	GetProcessTimes	Obtiene los tiempos del proceso.
Ficheros y E/S	tcgetattr, tcsetattr	SetConsoleMode	Establece el modo de operación del terminal.
Ficheros y E/S	read, write	ReadConsole, WriteConsole	Lectura y escritura en el terminal.
Ficheros y E/S	close	CloseHandle	No está limitada a ficheros.
Ficheros y E/S	open, creat	CreateFile	Crea o abre un fichero.
Ficheros y E/S	unlink	DeleteFile	Borra un fichero.
Ficheros y E/S	fsync	FlushFilebuffers	Vuelca la cache del fichero a disco.
Ficheros y E/S	stat, fstat	GetFileAttributes	Obtiene los atributos de un fiche-

Tema	UNIX	Windows	Comentarios
			ro.
Ficheros y E/S	stat, fstat	GetFileSize	Longitud del fichero en bytes.
Ficheros y E/S	stat, fstat	GetFileTime	Fechas relevantes para el fichero.
Ficheros y E/S	stat, fstat	GetFileType	Obtiene el tipo de un fichero o dispositivo.
Ficheros y E/S	stdin, stdout, stderr	GetStdHandle	Devuelve un dispositivo de E/S estándar.
Ficheros y E/S	link, symlink		Windows no proporciona enlaces.
Ficheros y E/S	readv		Lectura múltiple.
Ficheros y E/S	writev		Escritura múltiple.
Ficheros y E/S	read	ReadFile	Lee datos de un fichero.
Ficheros y E/S	truncate, ftruncate	SetEndOfFile	Fija la longitud de un fichero.
Ficheros y E/S	fcntl	SetFileAttributes	Cambia los atributos de un fichero.
Ficheros y E/S	lseek	SetFilePointer	Devuelve el apuntador de posición del fichero.
Ficheros y E/S	utime	SetFileTime	Modifica las fechas de un fichero.
Ficheros y E/S		SetStdHandle	Define un manejador de E/S estándar.
Ficheros y E/S	write	WriteFile	Escribe datos a un fichero.
Ficheros y E/S		CreateFileMapping	Define la proyección de un fichero en memoria.
Ficheros y E/S	mmap	MapViewOfFile	Proyecta un fichero en memoria.
Ficheros y E/S		OpenFileMapping	Abre un fichero proyectado en memoria.
Ficheros y E/S	munmap	UnmapViewOfFile	Elimina la proyección de memoria de un fichero.
Directorios	mkdir	CreateDirectory	Crea un nuevo directorio.
Directorios	closedir	FindClose	Cierra un directorio.
Directorios	opendir	FindFirstFile	Abre un directorio. En Windows también lee la primera entrada.
Directorios	readdir	FindNextFile	Extrae la siguiente entrada de directorio.
Directorios	getcwd	GetCurrentDirectory	Devuelve el nombre del directorio de trabajo.
Directorios	rmdir, unlink	RemoveDirectory	Borra un directorio.
Directorios	chdir, fchdir	SetCurrentDirectory	Cambia el directorio de trabajo.
Seguridad		DeleteAce	Borra una entrada de control de acceso de una ACL.
Seguridad	stat, fstat, lstat	GetAce	Devuelve una entrada de control de acceso de una ACL.
Seguridad	stat, fstat, lstat	GetAclInformation	Obtiene la información de una ACL.
Seguridad	stat, fstat, lstat, access	GetFileSecurity	Devuelve el descriptor de seguridad de un fichero.
Seguridad	stat, fstat, lstat	GetSecurityDescriptor	Devuelve el descriptor de seguridad de un usuario.
Seguridad	getlogin	GetuserName	Devuelve el nombre de sistema de un usuario.
Seguridad		InitializeAcl	Inicia la información de una ACL.
Seguridad	umask	InitializeSecurityDescriptor	Inicia el descriptor de seguridad de un usuario.
Seguridad	getpwnam, getgrnam	LookupAccountName	Devuelve el nombre de sistema de una cuenta.
Seguridad	getpwuid, getuid, geteuid	LookupAccountSid	Devuelve el identificador de sistema de una cuenta.
Seguridad	setuid, seteuid, setreuid		Activan los distintos UID de un fichero en UNIX.

Tema	UNIX	Windows	Comentarios
Seguridad	setgid, setegid, setregid		Activan los distintos GID de un fichero en UNIX
Seguridad	getgroups, setgroups, initgroups	OpenProcessToken	Grupos suplementarios.
Seguridad	chmod, fchmod	SetFileSecurity	Cambian permisos de ficheros.
Seguridad		SetPrivateObjectSecurity	Cambian permisos de objetos privados.
Seguridad	umask	SetSecurityDescriptor-Dacl	Cambian máscara de protección por defecto.
Seguridad	chown, fchown, lchown	SetSecurityDescriptor-Group	Cambian el propietario de un fichero.

BIBLIOGRAFÍA

En esta sección se muestra la bibliografía referenciada a lo largo de los capítulos del libro. Está ordenada alfabéticamente por orden de referencia.

[Álvarez 2004] Gonzalo Álvarez Marañón; Pedro Pablo Pérez García. *Seguridad Informática Para La Empresa Y Particulares*. Editorial McGraw-Hill. 2004.

[Bach 1986] M. J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, 1986.

[Burns 2003] A. Burns y A. Wellings. Sistemas de Tiempo Real y Lenguajes de Programación. 3ª Edición. Addison-Wesley. 2003

[Knowlton 1965] K. C. Knowlton, "A Fast Storage Allocator", *Communications of the ACM*, vol. 8, núm. 10, 1965.

[Maekawa 1987] Maekawa, M., Oldehoeft, A. E., Oldehoeft, R. R., *Operating Systems, Advanced Concepts*. The Benjamin/Cummings Publishing Company, 1987.

[Smith 1985] A.J. Smith, *Disk Cache-Miss Ratio Analysis and Design Considerations*, ACM Transactions on Computer Systems, Vol. 3, No. 3, August 1985, pp. 161-203.

[Abernathy 1973] D.H. Abernathy, et al, *Survey of Design Goals for Operating Systems*, Operating Systems Review Vol. 7, No. 2, April 1973, pp. 29-48; OSR Vol. 7, No. 3, July 1973, pp. 19-34; OSR Vol. 8, No. 1, Jan. 1974, pp. 25-35.

[Accetta 1986] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, M. Young. *Mach: A New Kernel Foundation for UNIX Development*. Proceedings Summer 1986. USENIX Conference. pp. 93-112. 1986

[Akyurek 1995] S. Akyürek, and K. Salem, *Adaptive Block Rearrangement*, ACM Transactions on Computer Systems, 13(2), pp. 89-121, May 1995.

[Andrews 1996] M. Andrews, *C++ Windows NT Programming*, 735 pages, M&T Press, 1996.

[Arnold 1993] D. Arnold, *UNIX Security -A Practical Tutorial*, McGraw-Hill, 1993.

[Bach 1986] Bach, M. J., *The Design of the UNIX Operating System*. Prentice-Hall 1986.

[Bech, 1998] M. Beck, N. Boheme, M. Dziadzka, et al. *Linux Kernel Internals*. Addison-Wesley, 1988. Segunda Edicicón.

[Beck 1996] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner, *Linux Kernel Internals*, Addison-Wesley, 1996.

[Beck 1998] Beck, M. et al. *Linux Kernel Internals*. Addison-Wesley, 2º edición, 1998.

[Beck 1999] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner, *Linux Kernel Internals*, 2nd. Edition, Addison-Wesley, 1996.

[Belady 1969] L. A. Belady et al., "An Anomaly in Space-Time Characteristics of Certain Programms Running in a Paging Machine", *Communications of the ACM*. vol. 12, núm. 6. 1969.

[Ben Ari 1990] Ben Ari, *Principles of Concurrent and Distributed Programming*. Prentice Hall 1990

[Birman 1994] K. P. Birman, R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* New York: IEEE Computer Society Press, 1994

[Birman 1996a] K. P. Birman, R. van Renesse. *Software for Reliable Networks.* Scientific American 274:5, pp. 64-69, mayo 1996

[Birman 1996b] K. P. Birman. *Building Secure and Reliable Networks Applications*. Manning Publications Co. 1996

[Birrel 1984] A.D. Birrel, B. J. Nelson. *Implementig Remote Procedure Calls.* ACM Transactions on Computer Systems, Vol 2, pp. 39-59, febrero 1984

[Biswas 1993] P. Biswas, K.K. Ramakrishnan, and D. Towsley, ”Trace Driven Analysis of Write Caching Policies for Disks”, *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement, and Modeling of Computer Systems*, pp. 13-23, May 10-14 1993.

[Black 1990] D. L. Black, "Scheduling Support for Concurrency and Parallelism in the Mach Operating System", *IEEE Computer*, vol. 23, núm. 5, págs. 35-43, 1990.

[Bovet 2005] Bovet, D. P., Cesati, M., *Understanding the Linux Kernel*. O’Reilly, tercera edición, 2005.

[Caresick 2003] Anna Caresik, Oleg Kolesnikov y Brian Hatch, Redes privadas virtuales (VPN) con Linux, Pearson Educación, 2003

[Carpinelli , 2001] J. D. Carpinelli. *Computer Systems Organization and Architecture*. Addison-Wesley, 2001.

[Carracedo 2004] Justo Carracedo, Seguridad en redes telemáticas, McGraw Hill, 2004

[Carretero 2000] J. Carretero, J. Fernández, F. García, “Enhancing Parallel Multimedia Servers through New Hierarchical Disk Scheduling Algorithms”**,** *VECPAR'2000, 4th International Meeting on Vector and Parallel Processing*, Oporto, junio 2000.

[CCEB 1994] Common Criteria Editorial Board, *Common Criteria for Information Technology Security Evaluations*, version 0.6, April 1994.

[CCP 2006] Common Criteria for Information Technology Security Evaluation. Part 3: Security assurance requirements. Version 2.3. http://www.commoncriteriaportal.org/. 2005.

[Chase 1994] H.E Chase, H. E. Levy, M. J. Feely, E. D. Lazowska, “Sharing and adressing in a single address space system”, *ACM Transactions on Computer Systems*, vol. 12, núm, 3, 1994.

[Cherry, 2004] S. Cherry. *Edholm's law of bandwidth*. Spectrum, IEEE. Volume: 41, Issue: 7. Julio 2004, págians 58-60.

[Coffman 1971] Coffman, p. ej., Elphick, M.J., Shoshani, A., “System Deadlocks”. *Computing Surveys*, vol. 3, núm. 2, junio 1971, págs. 67-78.

[Coine 1999] Robert A. Coiné, HPSS Tutorial, Technical Report, IBM Global Government Industry, http://www.sdsc.edu/hpss, 1999.

[Corbet 2005] J. Corbet, et al., Linux Device Drivers, O’Reilly, tercera edición, 2005.

[Coulouris 2005] G. Coulouris, J. Dollimore, T. Kindberg *Distributed Systems. Concepts and Design*. Cuarta edición. Addison-Wesley, 2005,

[Crowley, 1997] C. Crowley. *Operating Systems, A Design-Oriented Approach*. Irwin, 1997

[Curry 1992] D. Curry, *UNIX System Security - A Guide for Users and System Administrators*, Addison-Wesley, 1992.

[Daniel 2005] Daniel P. Bovet, Marco Cesati. *Understanding the Linux Kernel***,** O'Reilly & Associates Tercera edición 2005

[Davy 1995] W. Davy, "Method for Eliminating File Fragmentation and Reducing Average, Seek Times in a Magnetic Disk Media Environment", *USENIX*, 1995.

[Deitel 1993] H.M. Deitel, *Sistemas Operativos (2ª edición)*, Addison-Wesley, 1993

[Deitel, 1994] H. M. Deitel, M. S. Kogan. *The Design of OS/2*. Prentice-Hall, 1994

[Dekker 1999] E. N. Dekker and J. M. Newcomer, "*Developing Windows NT Device Drivers: A Programmer's Handbook*", Addison Wesley, 1999.

[deMiguel, 2004] P. de Miguel. *Fundamentos de los computadores*, 9ª edición. Thomson-Paraninfo 2004.

[Denning 1982] D. Denning, *Cryptography and Data Security*, Addison-Wesley, 1982.

[Dijkstra 1965] Dijkstra, E.W., "Cooperating Sequential Processes". *Technical Report EWD-123*, Technological University, Eindhoven, the Netherlands, 1965.

[Dijkstra, 1968] E. W. Disjkstra. *The Structure of THE Multiprogramming System*. Communications of the ACM.Vol 11, pp. 341-346. Mayo 1968

[Farrow 1990] R. Farrow, *UNIX System Security - How to Protect your Data and Prevent Intruders*, Addison-Wesley, 1990.

[Fidge 1988] C. Fidge. *Timestamps in Message-Passing Systems That Preserve the Partial Ordering.* Proceedings of the Eleventh Australian Computer Science Conference. 1988

[Fites 1989] P. Fites et al., *Control and Security of Computer Information Systems*, Computer Science Press, 19811.

[Folk 1987] M.J. Folk and B. Zoellick, *File Structures*, Addison-Wesley, 1987.

[Forouzan 2007] B. A. Forouzan. *Data Communications and Networking.* Cuarta edición. McGraw-Hill, 2007.

[Foster 2002] I. Foster. *What is the Grid? A Three Point Checklist.* I. GRIDToday, July 20, 2002.

[Foster 2006] I. Foster. *Globus Toolkit Version 4: Software for Service-Oriented Systems*.IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779, pp 2-13, 2006.

[Galli 2000] L. D. Galli. *Distributed Operating Systems : Concepts and Practice*. Prentice-Hall, 2000.

[Galli, 1999] L. D. Galli. *Distributed operating systems: concepts and practice*. Prentice-Hall, 1999.

[Garfinkel 1996] Simson Garfinkel and Eugene H. Spafford. *Practical UNIX & Internet Security*. O'Reilly & Associates, 2nd edition, Abril 1996.

[Gingell 1987] R. A. Gingell, et al., "Shared Libraries in SunOS", *Summer Conference Proceedings*, USENIX Association, 1987.

[Goodheart 1994] B. Goodheart and J. Cox, *The Magic Garden Explained: The Internals of UNIX System V Release 4, an Open Systems Design*, pp. 664, Prentice-Hall, 1994.

[Gorman 2004] M. Gorman, *Understanding the Linux Virtual Memory Manager*, Bruce Perens' Open Source Series, 2004.

[Grimshaw 1997] A. S. Grimshaw, Wm. A. Wulf, and the Legion Team. *The Legion Vision of a Wordwide Virtual Computer.* Communications of the ACM, 40(1), enero 1997

[Grosshans 1986] D. Grosshands, *File Systems Design and Implementation*, Prentice Hall, 1986.

[Habermann 1969] Habermann A.N., Prevention of System Deadlocks. *Communications of the ACM*, vol. 12, núm. 7, julio 1969, págs. 373-377.

[Hamacher, 2002] V. C. Hamacher, Z. Zvonko Vranesic y S. Zaky. *Computer Organization*, 5.ª edición. McGraw Hill 2002.

[Hart 2004] J. M. Hart, *Windows System Programming*, Addison-Wesley, tercera edición, 2004.

[Hart, 1998] J. M. Hart. *Win32 System Programming*. Addison-Wesley, 1998.

[Havender 1968] Havender J.W., "Avoiding Deadlocks in Multitasking Systems". *IBM Systems Journal*, vol. 7, núm. 12, 1968, págs. 74-84.

[Hennessy, 2002] J. L. Hennessy, D. A. Patterson y D. Goldberg. *Computer Architecture. A Quantitative Approach*, 3.ª edición. Morgan Kaufmann Publishers. 2002.

[Hoare 1974] Hoare, C.A.R. *Monitors: An Operating Systems Structuring Concept*. Communications of the ACM, vol 17, num 10. Oct 1974, pags. 549-557

[Holt 1972] Holt, R.C., "Some Deadlock Properties of Computer Systems". *Computing Surveys*, vol. 4, núm. 3, septiembre 1972, págs. 179-196.

[Howard 1973] Howard J.H., "Mixed Solutions for the Deadlock Problem". *Communications of the ACM*, vol. 16, núm. 7, julio 1973, págs. 427-430.

[Hunt 2005] Galen Hunt et al., "An Overview of the Singularity Project", Microsoft Research, Microsoft Research Technical Report, MSR-TR-2005-135, 2005, http://research.microsoft.com/os/singularity

[IEEE 1988] IEEE, *IEEE Standard Portable Operating System Interface for Computer Environments*, IEEE Computer Society Press, 1988.

[IEEE 1999] Proceedings of the IEEE. Marzo 1999. Número especial sobre *Memoria compartida Distribuida*

[IEEE, 1996] *Information technology. Portable Operating System Interface (POSIX). System Application Program Interface (API)*. IEEE Computar Society, 1996.

[IEEE, 2004] *The Single UNIX Specification*, Version 3, 2004 Edition IEEE Std 1003.1

[Jacob 1998a] B. Jacob, T. Mudge, "Virtual Memory: Issues of Implementation", *Computer*, vol. 31, núm. 6, págs. 33-43, 1998.

[Jalote 1994] P. Jalote. *Fault Tolerance in Distributed Systems*. Prentice Hall, 1994.

[Kernighan 1978] B. Kernighan, and D. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

[Krakowiak 1988] S. Krakowiak, *Principles of Operating Systems*, MIT Press. 1988.

[Lambert 1999] P. Lambert, *Implementing Security on Linux*, Journal of System Administration, Vol. 8, No. 10, pp. 67-70, October 19911.

[Lamport 1978] L. Lamport. *Time, Clocks, and the Ordering of Events in a Distributed System.* Communications of the ACM 21:7. Pp. 558-565, abril 1978

[Lampson 1980] Lampson B., Redell D. *Experience with Processes and Monitors in Mesa*. Communications of the ACM, Feb. 1980.

[Levine 2000] J. R. Levine, *Linkers and Loaders*, The Morgan Kaufmann Series in Software Engineering and Programming, 2000.

[Levine 2003a] Gertrude Neuman Levine, "Defining deadlock", *ACM SIGOPS Operating Systems Review*, vol. 37, núm. 1, enero 2003, págs. 54-64.

[Levine 2003b] Gertrude Neuman Levine, "Defining deadlock with fungible resources", *ACM SIGOPS Operating Systems Review*, vol. 37, núm. 3, julio 2003, págs. 5-11.

[Li 1986] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* Tesis Doctoral, Universidad de Yale, 1986

[Liu 2002] Jane W.S. Liu. *Real-Time Systems*. Prentice-Hall. 2000.

[Love 2005] Love, R.. *LinuxKernel Development*. Novell Press, 2005.

[Maekawa 1987] M. Maekawa el al., *Operating Systems: Advanced Concepts*, Benjamin-Cummings, 1987.

[Mattern 1989] F. Mattern. *Tkime and Global States in Distributed Systems.* Proceedings of the International Workshop on Parallel and Distributed Algorithms. Amsterdam. 1989.

[Mauro 2000] Mauro J., McDougall R.. *Solaris Internals. Core Kernel Architecture*. Prentice Hall 2005.

[McKusick 1996] McKusick M. K., Bostic K., Karels M. J. and Quaterman J. S. *The Design and Implementation of the 4.4 BSD Operating System*. Addison-Wesley 1996

[McKusick 2004] McKusick M., Neville-Neil, G. V. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley, 2004.

[Mealy, 1966] G. H. Mealey, B. I. Witt, W. A. Clark. *The Structural Structure of OS/360*. IBM System Journal, vol 5. Num 1 1966.

[Megiddo 2004] N. Megiddo, D. S. Modha, "Outperforming LRU with an Adaptive Replacement Cache Algorithm", *IEEE Computer Magazine*, págs. 58-65, 2004.

[Milenkovic, 1992] M. Milenkonvic. *Operating Systems: Concepts and Design*. McGraw-Hill, 1992

[Mosberger 2004] D. Mosberger, S. Eranian, *IA-64 Linux Kernel: design and implementation*.. Prentice Hall, 2004.

[MPI www] Message Passing Interface Forum. http://www.mpi-forum.org.

[Mullender 1990] S. Mullender, G. Van Rossum, A. S. Tanenbaum, R. Van Renesse, H. Van Staveren. *Amoeba: A Distributed Operating Systems for the 1990s*. IEEE Computer, vol 23, Nº 5 pp. 44-53, mayo 1990

[Mullender 1993] S. Mullender (Ed.). *Distributed Systems.* Segunda edición, ACM Press, Nueva York, 1993

[Nagar 1997] R. Nagar, *Windows NT File System Internals*, pp. 774, O'Reilly & Associates Inc., 1997

[Newton 1979] Newton G., "Deadlock Prevention, Detection and Resolution: An Annotated Bibliography". *Operating Systems Review*, vol. 13, núm. 2, abril 1979, págs. 33-44.

[Nutt 2004] G. Nutt, *Operating Systems: A Modern Perspective*, tercera edición, Addison Wesley, 2004.

[Orfali 1999] R. Orfali, D. Harkey. J. Edwards. *Client/Server Survival Guide.* Wiley Computer Publishing, 1999, tercera edición

[Organick 1972] E. I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press, 1972.

[Otte 1996] R. Otte, P. Patrick, M. Roy. *Understanding CORBA*. Prentice-Hall, 1996

[Ousterhout 1989] J.K. Ousterhout, and F. Douglis, *Beating the I/O Bottleneck: A case for Log Structured File Systems*, ACM Opearing Systems Review, Vol. 23, No. 1, pp. 11-28, Jan 1989.

[Patterson, 2004] D. A. Patterson y J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface*, 3.ª edición. Morgan Kaufmann Publishers. 2004.

[Pfleeger 1997] C.P. Pfleeger, *Security in Computing*, 2nd Edition, Prentice Hall, 1997.

[Protic 1998] J. M. Protic, M. Tomasevic, V. Milutinovic. *Distributed Shared Memory: Concepts and Systems*. IEEE Computer Society Press, Los Alamitos, California, 1998.

[QNX, 1997] QNX Software Systems Ltd. *QNX Operating Systems, System Architecture*. 1997.

[Rashid 1987] R. Rashid, et al. "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures", *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 1987.

[Ready, 1986] J. F. Ready. VRTX: *A Real-Time Operating System for Embedded Microprocessor Applications*. MICRO, Vol. 6, No. 4, Aug. 1986, pp. 8-17.

[Ricart 1981] G. Ricart, A. K. Agrawala. *An Optimal Algorithm for Mutual Exclusion in Computer Networks.* Communications of the ACM, vol 24, pp. 9-17, enero 1981

[Richter 1994] J. Richter, *Advanced Windows NT*, Microsoft Press, 1994

[Rockind 1985] M.J. Rochkind, *Advanced UNIX Programming*, Prentice-Hall, 1985.

[Roizer 1988] M. Roizer, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont., F. Herrmann, C. Kaiser, P. Leonard, S. Langlois, W. Neuhauser. *Chorus Distributed Operating System*. Computing Systems, vol 1, pp. 305-379. octubre 1988

[Rosenberry 1992] W. Rosenberry, D. Kenney, G. Fisher. *Understanding DCE*. O'Reilly, 1992

[Rubin 1999] W. Rubin, M. Brain, R. Rubin. *Understanding DCOM*. Prentice-Hall 1999.

[Russinovich 2005] Mark E. Russinovich and David Solomon, “Microsoft Windows Internals. 4th Edition“, Microsoft Press, 2005

[Samsom, 1990] S. Samson. *MVS Performance Management*. McGraw-Hill, 1990.

[Seagate 2000] Seagate, *Barracuda ATA II Family: Product Manual*, Seagate, 2000.

[Silberchatz, 2005] A. Silbertchatz, P. Galvin. *Operating Systems Concepts*. Addison-Wesley, 2005, Sexta Edición.

[Silberschatz 2005] Silberschatz, A., Galvin, P. B., *Operating Systems Concepts*. John Wiley & Sons, Inc., séptima edición, 2005.

[Silberschatz 2006] A. Silberschatz andAbraham, Fundamentos de Sistemas Operativos, 7ª Edición, McGraw Hill Intermaericana, 2005.

[Smith 1985] A.J. Smith, *Disk Cache-Miss Ratio Analysis and Design Considerations*, ACM Transactions on Computer Systems, Vol. 3, No. 3, August 1985, pp. 161-203.

[Smith 1994] K. Smith and M. Seltzer, *File Layout and File System Performance*, Technical Report, Harvard University, Number TR-35-94, 1994.

[Solomon 1998] D. A. Solomon, *Inside Windows NT*, 2nd. Edition, Microsoft Press, 1998.

[Solomon 2004] Solomon, D. A., Russinovich, M. E. *Microsoft Windows Internals*. 4º edición, Microsoft Press, 2004

[Solomon, 1998] D. A. Solomon. *Inside Windows NT,* 2.ª edición. Microsoft Press, 1998.

[Staelin 1988] C. Staelin, *File Access Patterns*, Technical Report, Department of Computer Science, Princeton University,, Number CD-TR-179-88, September 1988.

[Stallings 2005] Stallings, W., *Operating Systems, Internals and Design Principles*. Prentice-Hall, quinta edición 2005.

[Stallings 2007] W. Stallings. *Data and Computer Communications*. Octava edición, Prentice Hall 2007.

[Stallings, 2001] W. Stallings. *Operating Systems, Internals and Design Principles*, 4.ª edición. Prentice-Hall, 2001.

[Stallings, 2003] W. Stallings. *Computer Organization and Architecture: Designing for Performance*, 6.ª edición. Macmillan. 2003.

[Stevens 1992] W. Stevens, *Advanced Programming in the UNIX Environment*, Addison-Wesley, 1992.

[Stevens 1999] *UNIX Network Programming*. Prentice Hall 1999.

[Tanenbaum 1992] A. Tanenbaum, *Modern Operating Systems*, Prentice-Hall, 1992.

[Tanenbaum 1997] A. Tanenbaum and A. Woodhull, *Operating Systems Design and Implementation*, 2nd Edition, Prentice Hall, 1997.

[Tanenbaum 2001] A. Tanenbaum, *Modern Operating Systems*, Prentice-Hall, segunda edición, 2001.

[Tanenbaum 2002] A. S. Tanenbaum. *Computer Networks*. Cuarta edición, Prentice-Hall 1996.

[Tanenbaum 2006] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos, "Can We Make Operating Systems Reliable and Secure?", *IEEE Computer*, vol. 39, No 5, May 2006.

[Tanenbaum 2007] A. S. Tanenbaum. *Distributed Systems: Principles and Paradigms*. Segunda edición. Prentice-Hall, 2007.

[Tanenbaum, 2005] A. S. Tanenbaum. *Structured Computer Organization*, 5.ª edición. Prentice Hall 2005.

[Tanenbaum, 2006] A. S. Tanenbaum, A. S. Woodhull. *Operating Systems: Design and Implementation*, 3.ª edición. Prentice-Hall, 2006.

[Vahalia 2006] U. Vahalia, UNIX Internals: The New Frontiers. Prentice Hall, segunda edición, 2006.

[Weikum 1992] G. Weikum and P. Zabback, "Tuning of Striping Units in Disk-Array-Based File Systems", *Proceedings of the 2nd International Workshop on Research Issues in Data Engineering*, pp. 80-87, February 2-3 1992.

[Wilkes 1995] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan, "The HP AutoRAID Hierarchical Storage System", *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pp. 96-108, December 3-6 1995.

[Wilson 1995] P. R. Wilson, M. S. Johnstone, M. Neely, D. Boles. "Dynamic Storage Allocation: A Survey and Critical Review". *In Proceedings of the 1995 International Workshop on Memory Management,* 1995.

[Wood 1895] P.H. Wood and S.G. Kochan, *UNIX System Security*, Hayden UNIX System Library, Hayden Books, 1985.

[Zobel 1983] Zobel D., "The Deadlock Problem: A Classifying Bibliography". *Operating Systems Review*, vol. 17, núm. 4, octubre 1983.

ÍNDICE