Conceptos estructurales y funcionales del sistema operativo

Remo Suppi Boldrito
Josep Lluís Marzo i Làzaro

PID_00180052

UOC
Universitat Oberta de Catalunya

Índice

Introducción..... 5

Objetivos..... 6

1. Estructura interna de un sistema operativo..... 7
1.1. Estructura monolítica 7
1.2. Estructura jerárquica o por capas 10
1.3. Estructura de máquina virtual 11
1.4. Estructura por funciones 12
1.5. Conclusiones 17

2. El núcleo del sistema operativo..... 19
2.1. El núcleo de Unix 21
2.2. El núcleo Linux 22
2.3. Configuración y compilación del núcleo Linux 26
2.4. Compilación del núcleo en Debian (Debian Way) 29
2.5. Proceso de arranque en Linux 31
2.6. El núcleo de Windows NT 33

3. Soporte del hardware..... 37
3.1. Registros 37
3.2. La entrada y la salida 38
3.3. Controladores de entrada/salida 39
3.4. La transferencia directa a la memoria, DMA 41
3.5. El reloj del sistema 44
3.6. Protección 45
3.7. Modos de ejecución del procesador 46
3.8. Mecanismos de acceso al núcleo del sistema operativo 47
3.9. Jerarquía de la memoria 54
3.9.1. La unidad de gestión de la memoria 54
3.10. Instrucciones privilegiadas 55
3.11. Clases de dispositivos en Unix/Linux 57
3.12. Estructura de datos dinámica en Linux 59
3.13. Multinúcleo, multiprocesador y multiordenador 64

Resumen..... 67

Actividades..... 69

Ejercicios de autoevaluación..... 69

Solucionario..... 70

Glosario..... 71

Bibliografía..... 72

Introducción

En este módulo se describen las diferentes estructuras que podemos encontrar en el núcleo de un sistema operativo (SO) en función de cómo está organizado internamente. Basándonos en la estructura interna se describen las diferentes estructuras más interesantes, como, por ejemplo, organización monolítica, organización por capas o jerárquica, estructura de máquina virtual, cliente-servidor (*microkernel*), organización por funciones/objetos, etc.

A partir de las distintas estructuras del núcleo del SO, se presenta un análisis del núcleo desde la funcionalidad y la capacidad para llevar a cabo una gestión del sistema en lo que se refiere a los procedimientos que gestionan recursos comunes del sistema de cierta importancia.

Siguiendo en esta línea, a continuación se muestran las particularidades que encontramos en el modo de de ejecución del núcleo del SO, y se justifica la necesidad de disponer de soporte hardware en el momento de gestionar determinados procedimientos, ya sea por cuestiones de seguridad como por el aumento de eficiencia si se realizan en este ámbito (además, sería imposible gestionar algunos de ellos por medios software, como ser verá más adelante).

Finalmente, se realiza un estudio de la gestión del sistema y qué soporte hardware está involucrado en esta teniendo en cuenta que no es un estudio exhaustivo de cada circuito o registro de soporte ni de las diferentes necesidades de los distintos procedimientos de gestión. En todos estos aspectos se verán ejemplos relacionados con el sistema operativo GNU/Linux.

Objetivos

Los materiales didácticos de este módulo presentan los aspectos fundamentales para que alcancéis los siguientes objetivos:

1. Conocer las diferentes estructuras internas que puede tener el núcleo de un SO y las respectivas características diferenciales, y saber diferenciarlas en función de su especialización.

2. Saber identificar los elementos de hardware necesarios para implementar algunos servicios específicos del SO.

3. Conocer las modalidades de ejecución de los procesos con respecto a sus privilegios, especialmente el modo del núcleo del SO.

4. Conocer las razones por las que el SO necesita el soporte de hardware para controlar el sistema. Algunas razones tienen que ver con la eficiencia y otras se relacionan con la naturaleza del control.

5. Entender la importancia del soporte de hardware del ordenador en el momento de efectuar determinadas funciones del SO y saber qué tipos de procedimientos necesitan este soporte.

6. Poder diferenciar la ejecución de operaciones normales de la ejecución de instrucciones privilegiadas, y conocer qué tipo de gestión de los servicios de SO permiten estas instrucciones.

1. Estructura interna de un sistema operativo

Podemos establecer una primera clasificación de los sistemas operativos (SO) en relación a cómo prestan su servicio o cómo ejecutan su actividad (a veces se define como modo de trabajo). Una taxonomía utilizada en este sentido es la que los clasifica como por lotes (secuenciales), multiprogramados, de tiempo compartido, de tiempo real, distribuidos, de red, cliente-servidor, etc.

Sin embargo, en este apartado el interés se centra en el modo como están diseñados y construidos. Por ello la mayoría de los autores y diseñadores de sistemas operativos establecen clasificaciones en función de la estructura interna del núcleo, y entre las más interesantes y conocidas podemos definir las estructuras como: monolíticas, jerárquica o por capas, máquina virtual, por funciones/cliente-servidor[1]. A continuación se describirá cada una de las estructuras y se tendrán en cuenta sus principales aspectos.

(1)También conocida por su nombre en inglés, *microkernel*.

Véase también

Se pueden consultar las definiciones de *sistema operativo por lotes*, *multiprogramado*, *distribuido*, *en tiempo real*, *en tiempo compartido*, *de red*, etc. en el glosario del módulo "Introducción a los sistemas operativos" de la asignatura *Sistemas operativos*.

1.1. Estructura monolítica

Conocida frecuentemente como "sin estructura", en la **estructura monolítica** el SO[2] está diseñado como un conjunto de procedimientos que se pueden llamar entre sí sin ninguna limitación. Los sistemas operativos en los que el núcleo presenta una organización monolítica se caracterizan por que no tienen una estructura clara y prefijada para cada uno de los servicios que han de proporcionar.

(2)SO es la sigla de *sistema operativo*.

El diseño más clásico de un SO con estructura monolítica es pensar en la función del procedimiento que se va a realizar, editar y compilar los diferentes programas por separado y enlazarlos todos al final para formar un único objeto ejecutable.

Para evitar que el sistema sea totalmente un caos con múltiples puntos de accesos, salidas, recursividades, etc., se establecen unos puntos de entrada en el SO bien definidos que aportan una capa de organización y permiten interactuar con él. Este pseudoorden se completa con unas llamadas al núcleo y con parámetros prefijados para facilitar la interacción con el núcleo independientemente del modo como se haya programado en su interior, como podéis ver en la figura 1.

Como es de esperar, la lógica funcional presenta una estructura mínima:

- Existe un programa principal que atiende las peticiones de servicio y las envía a los procedimientos de servicio, que forman una capa interior del SO.
- Los procedimientos de servicio llaman a rutinas de código mucho más específico que se pueden ejecutar desde cualquier procedimiento de servicio que lo solicite.
- El hardware puede ser accedido desde diferentes puntos del código en virtud de cuál sea el servicio solicitante.

Se debe tener en cuenta que una organización y formalización de la entrada y salida de datos de las funciones resultantes pasaría a ser una estructura por capas, que es la siguiente que se analizará.

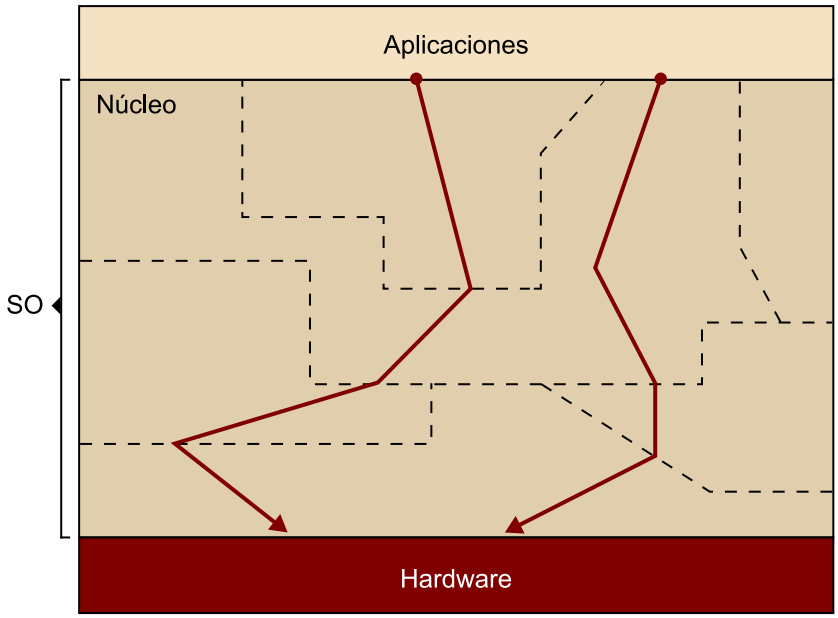
Como **ventajas** principales podemos mencionar las siguientes:

- Simplicidad en la comunicación entre módulos mediante llamadas.
- Alta optimización de código.
- Interfaz simple y homogénea.
- Coste reducido en el diseño y desarrollo.

Entre sus **desventajas** principales se pueden mencionar las siguientes:

- Complejidad.
- Difícil depuración.
- Gestión de errores y dificultades en su modificación (básicamente por su multiplicidad de puntos de llamada y accesos).

Figura 1. Sistema operativo monolítico



Unix, Linux y FreeBSD

Como ejemplo de sistema operativo de núcleo monolítico están Unix, Linux y FreeBSD (si bien Linux se considera una estructura monolítica híbrida). Estos sistemas tienen un

núcleo grande y complejo, que engloba todos los servicios del sistema. Está programado de forma no modular y tiene un rendimiento mayor que un micronúcleo. Sin embargo, cualquier cambio que se deba realizar en cualquier servicio requiere la recompilación del núcleo y el reinicio del sistema para aplicar los nuevos cambios.

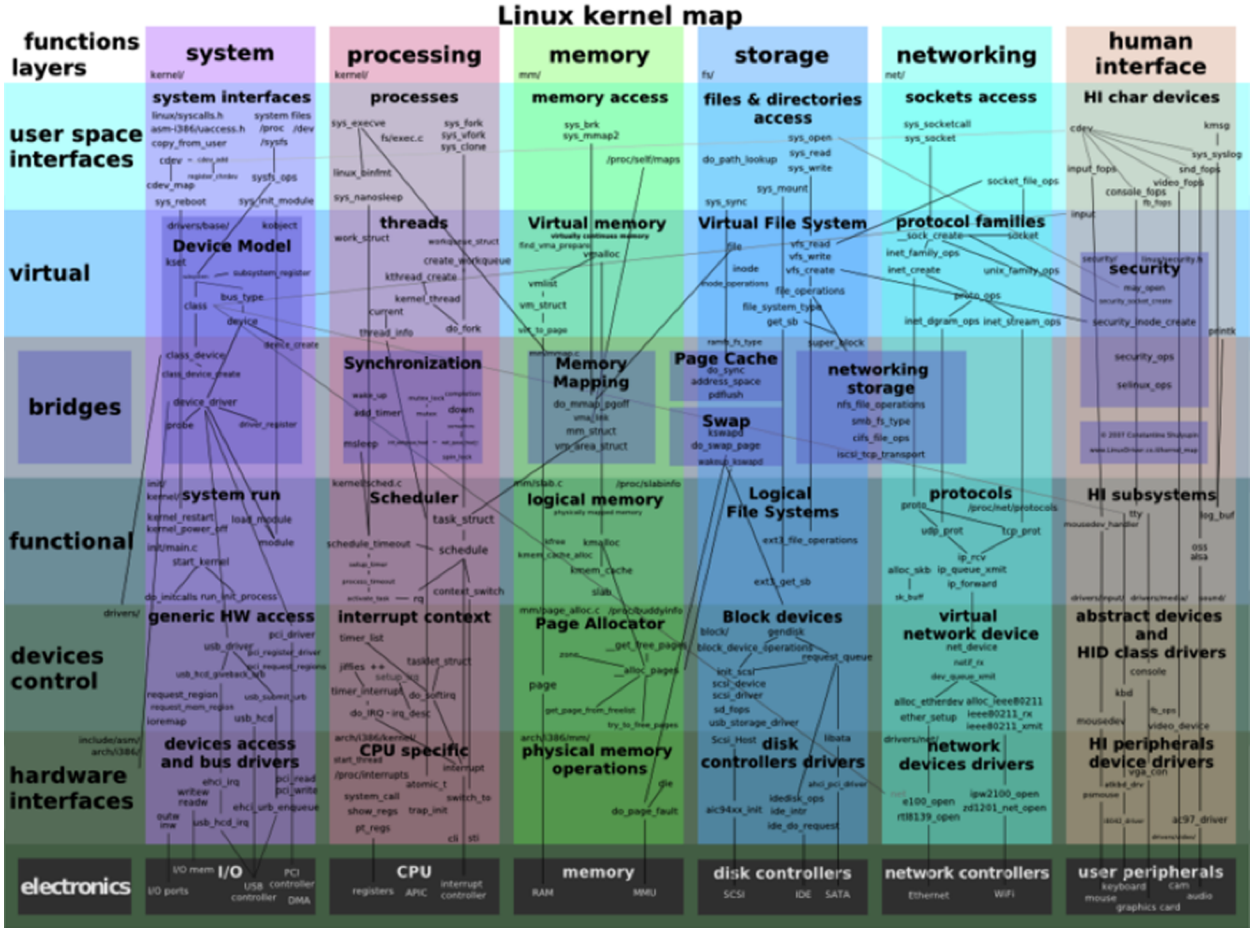
Hay varias ramificaciones de este diseño, que se han ido adaptando a nuevas necesidades, como, por ejemplo, el sistema de módulos ejecutables en tiempo de ejecución, que le brinda al modelo de núcleo monolítico algunas de las ventajas de un micronúcleo. Dichos módulos pueden ser compilados, modificados, cargados y descargados en tiempo de ejecución, de manera similar a los servicios de un micronúcleo, pero con la diferencia de que se ejecutan en el espacio de memoria del núcleo. Además, el módulo pasa a formar parte del núcleo, usando la misma interfaz de programación (API) y no se emplea un sistema de mensajes como en los micronúcleos. Linux, FreeBSD y varios derivados de Unix utilizan este mecanismo.

Entre los sistemas operativos que cuentan con núcleos monolíticos (si bien algunos de ellos se consideran híbridos) se encuentran:

- Núcleos tipo Unix: Linux, Syllable, Unix (BSD –FreeBSD, NetBSD, OpenBSD–, Solaris).
- Núcleos tipo DOS: DR-DOS, MS-DOS, Familia Microsoft Windows 9x (95, 98, 98SE, Me), o los basados en el núcleo NT –que se considera híbridos– (2k, XP, Vista, Server2003/08, W7).
- Núcleos Mac OS (hasta Mac OS 8.6).
- OpenVMS.
- XTS-400.

Es interesante ver una imagen del mapa interactivo del núcleo Linux:

Figura 2. Mapa interactivo del núcleo Linux



Detalle de la imagen disponible en web <http://www.makelinux.net/kernel_map>

1.2. Estructura jerárquica o por capas

En la evolución de los primeros SO y a medida que iban creciendo las necesidades de los usuarios, una línea de diseño fue organizar y estructurar el software del SO en forma jerárquica o por niveles. Los primeros diseños dividieron el SO en pequeñas partes para llegar a una unidad perfectamente definida y con una interfaz muy clara con el resto de los elementos/unidades, lo que constituía una estructura jerárquica o de niveles en los sistemas operativos. El primero de ellos fue denominado THE (*Technische Hogeschool*, Eindhoven), de Dijkstra, que se utilizó con fines didácticos.

La forma más fácil de diseño es por jerarquía de capas/niveles o anillos concéntricos. La idea básica es que una capa solo se puede comunicar con las capas contiguas. Dentro de esta jerarquía, las funciones más cercanas al hardware están en las capas inferiores, y las implicaciones de usuario, en las superiores.

Así, la asignación del procesador y de la memoria irá a capas "bajas" y las utilidades de usuario o la gestión de ficheros irán a capas "altas". La interfaz se logra, al igual que en los SO monolíticos, por puntos bien definidos en las capas para hacer llamadas, y acompañados de los parámetros que hay que pasar.

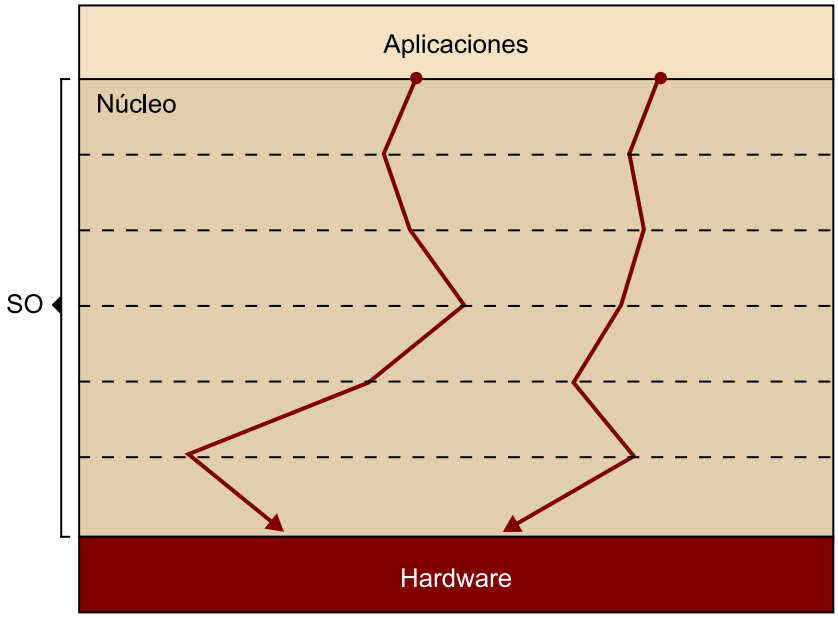
Aplicaciones de la estructura por capas

La técnica que se utiliza en la estructura por capas también se aplica a otros sistemas informáticos, como los protocolos de comunicaciones.

La **ventaja** fundamental de esta organización es la independencia existente entre capas. Esto significa que se puede modificar el código de una capa sin la obligación de tener en cuenta el código del resto de las capas, excepto en los puntos de entrada. Una excepción sería cuando el código de una capa no es necesario: entonces, en lugar de crear un procedimiento vacío, se puede saltar la capa y hacer la llamada directamente a una capa más lejana.

La estructura por capas permite describir un SO de manera más clara que en la estructura monolítica, aunque el sistema no esté estrictamente separado por capas.

Figura 3. Sistema operativo por capas



1.3. Estructura de máquina virtual

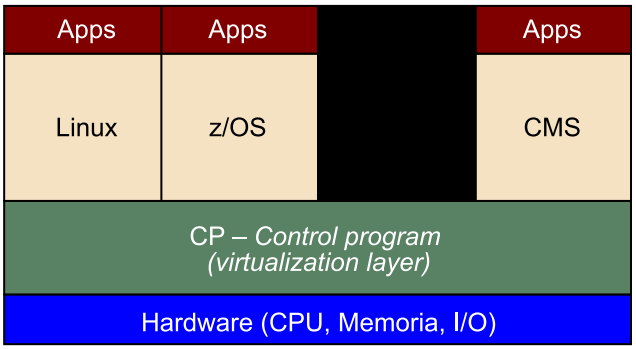
La **estructura de máquina virtual** consiste en una estructura de sistemas operativos que presentan una interfaz a cada proceso, mostrando una máquina que parece idéntica a la máquina real subyacente.

Estos sistemas operativos separan dos conceptos que suelen estar unidos en el resto de los sistemas: la multiprogramación y la máquina extendida. El objetivo de los sistemas operativos de máquina virtual es el de integrar distintos sistemas operativos que pueden atender diferentes necesidades sobre el mismo hardware. El núcleo de estos sistemas operativos se denomina monitor virtual y tiene como misión llevar a cabo la multiprogramación, presentando a los niveles superiores tantas máquinas virtuales como se soliciten. Estas máquinas virtuales no son máquinas extendidas, sino una réplica de la máquina real, de manera que en cada una de ellas se pueda ejecutar un sistema operativo diferente, que será el que ofrezca la máquina extendida al usuario.

Virtual machine operating systems

Un ejemplo de estos sistemas fue el mítico Virtual machine operating systems (VM, conocido también como VM/CMS) de IBM y utilizado en IBM mainframes System/370, System/390, zSeries en 1972 y que posteriormente evolucionó a CP/CMS operating system. En la actualidad, este sistema ha evolucionado a IBM z/VM Version 6.1 (2009), que necesita una arquitectura hardware z/Architecture 2 y se encuentra en los sistemas IBM System z10 (2009). Estas ideas han sido fundamentales en la virtualización de SO en la actualidad y, a pesar de diferentes ideas y conceptos, la esencia de la estructura subyace en los diseños más modernos.

Figura 4. Estructura de la máquina virtual VM/CMS



1.4. Estructura por funciones

Otra estructura conocida es una **estructuración por las funciones** que desarrolla el SO. En este caso, las agrupaciones se llevan a cabo según el tipo de servicio que se desea dar, sin tener en cuenta la proximidad o la distancia del hardware, como se ha realizado en la estructura por capas descrita anteriormente. Estas agrupaciones se pueden crear a partir de servicios de entrada/salida, la gestión de procesos, la gestión de la memoria, etc., teniendo en cuenta que cuando se construye el SO esta organización da como resultado una estructura vertical.

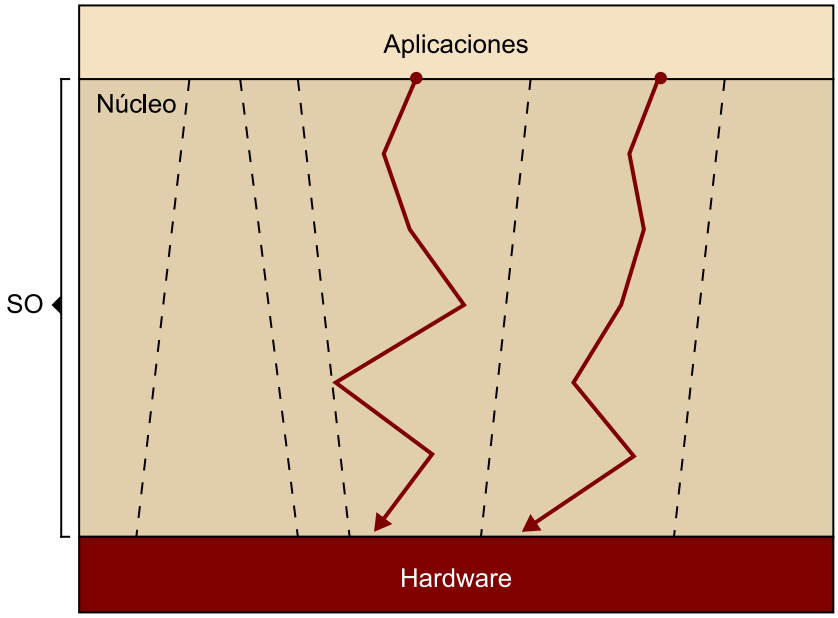
Es importante tener en cuenta que la organización vertical no se realiza de modo estricto, ya que hay servicios con fronteras difusas, como, por ejemplo, el de gestión de entrada/salida y el de ficheros, o los de gestión de procesos y de gestión de la memoria. A pesar de esta indefinición en la frontera, sí es necesario tener en cuenta que, al igual que en los otros SO, deben existir los puntos de acceso a las llamadas y los parámetros en forma bien definida.

Desde el punto de vista de la implementación, los SO con estructura funcional se pueden construir con el modelo cliente/servidor o con la programación de objetos. En ambos casos, el objetivo es el mismo: dar a las aplicaciones una colección de servicios que son servidos por ciertos procedimientos con entidad propia, sean servidores, sean objetos.

Una mención especial a un tipo de construcción de esta organización son los denominados *cliente-servidor*[3], que –más allá de su auge en algunos sistemas operativos actuales– permite que pueda ser ejecutado en la mayoría de los ordenadores, ya sean grandes o pequeños. Una de las características principales es que este tipo de organización sirve para toda clase de aplicaciones, por lo que puede ser considerado de propósito general y cumple con los mismos objetivos que los SO convencionales.

(3)Llamados también *microkernels* ya que, como se verá más adelante, esta categoría entra dentro de una estructura cliente-servidor.

Figura 5. Sistema operativo por funciones



El núcleo tiene como misión establecer la comunicación entre los clientes y los servidores. Los procesos pueden ser tanto servidores como clientes. Por ejemplo, un programa de aplicación normal es un cliente que llama al servidor correspondiente para acceder a un archivo o realizar una operación de entrada/salida sobre un dispositivo concreto. A su vez, un proceso cliente puede actuar como servidor para otro. Esta estructura ofrece gran flexibilidad en cuanto a los servicios posibles en el sistema final, ya que el núcleo provee solamente funciones muy básicas de memoria, entrada/salida, archivos y procesos, dejando a los servidores la mayoría de los servicios que el usuario final o programador puede utilizar. Estos servidores deben tener mecanismos de seguridad y protección, que, a su vez, serán filtrados por el núcleo que controla el hardware.

Micronúcleos, nanonúcleos y exonúcleos

El **paradigma del micronúcleo** tuvo una gran relevancia académica durante los años ochenta y principios de los noventa, dentro de lo que se denominó *self healing computing*, esto es, sistemas independientes que fuesen capaces de superar por sí mismos errores de software o hardware.

Es interesante leer un famoso debate surgido entre Linus Torvalds (creador de Linux) y Andrew Tanenbaum (profesor de de la Universidad Libre de Ámsterdam, y conocido por ser el creador de Minix, una réplica gratuita del sistema operativo Unix con propósitos educativos, y por sus libros sobre ciencias de la computación) sobre el micronúcleo y la portabilidad teniendo como base las críticas a Linux por parte de Tanenbaum y conocido en la comunidad como "Linux is obsolete[4]".

(4)http://oreilly.com/catalog/opensources/book/appa.html

Andrew Tanenbaum es un investigador muy respetado en el ámbito del diseño de sistemas operativos y en 1992 decidió hacer unos comentarios sobre el núcleo Linux y su obsolescencia, ya que no era un micronúcleo. Este debate es interesante, pues da una visión muy precisa del momento y de por qué Linus abandonó la idea de micronúcleos para Linux. En la discusión también participa Ken Thompson (uno de los fundadores de Unix) y David Miller (que es un gran *hacker* del núcleo Linux ahora), así como otras personas que han adquirido relevancia en el mundo de los SO y sus arquitecturas. Es importante decir que cuando se produjo el debate (1992), el 386 era el chip habitual (el 486 no había salido al mercado), Microsoft era una pequeña empresa que vendía DOS y Word para DOS, con Lotus 123 en el mercado de las hojas de cálculo y WordPerfect en el de procesamiento de textos, y no existía ni Yahoo, ni Hotmail, ni Google, etc.

Un micronúcleo, en un principio, pretendía ser una solución a la creciente complejidad de los sistemas operativos. Las principales **ventajas** de su utilización son las siguientes:

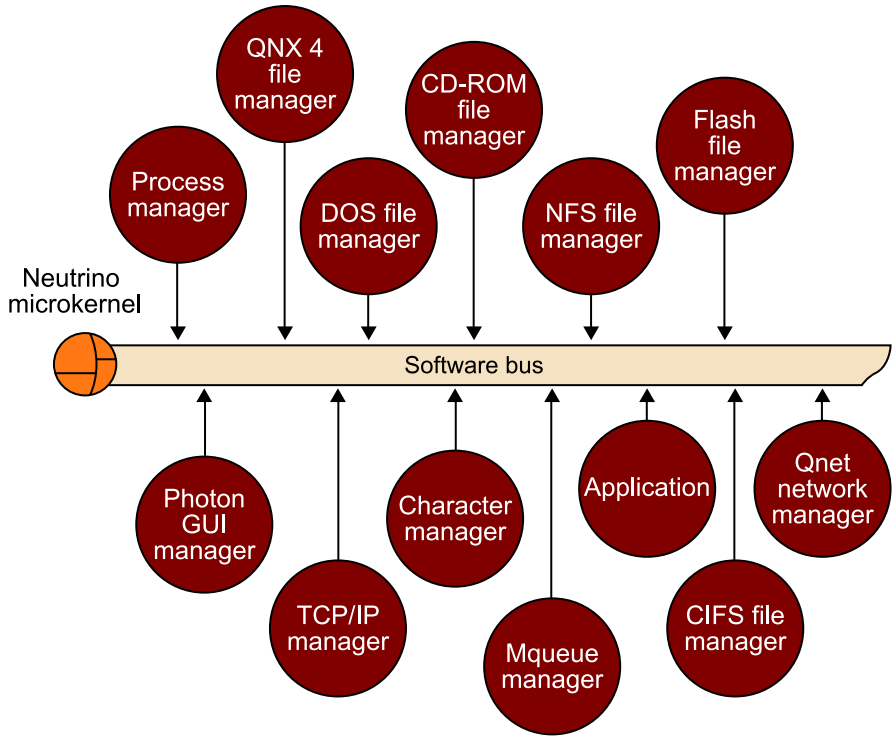
- Reducción de la complejidad.
- Descentralización de los fallos (un fallo en una parte del sistema no lo colapsaría por completo).
- Facilitación para crear y depurar controladores de dispositivos.

Entre sus principales **desventajas** están las siguientes:

- Complejidad en la sincronización de todos los módulos que componen el micronúcleo y su acceso a la memoria.
- Integración con las aplicaciones.

Entre los SO con micronúcleo más conocidos podemos citar: AIX, Minix, Hurd, NeXTSTEP (algunos lo consideran un núcleo híbrido), Netkernel, RadiOS, QNX, Symbian, SymbOS. La figura 6 muestra la imagen de QNX/Neutrino Microkernel.

Figura 6. QNX/Neutrino Microkernel



Fuente: *The Philosophy of QNX* <http://www.swd.de/documents/manuals/neutrino/intro_en.html>

Desde el punto de vista funcional, el **micronúcleo** es un tipo de núcleo de un SO que facilita un conjunto mínimo de llamadas al sistema para obtener los servicios básicos: espacios de direcciones, comunicación entre procesos y planificación básica. Los restantes servicios (gestión de memoria, sistema de archivos, operaciones de E/S, etc.) que se encuentran en las otras arquitecturas de núcleo conocidas se ejecutan en un modo de procesos servidores en espacio de usuario.

No obstante, se debe tener en cuenta que los procesadores y las arquitecturas modernas de hardware están optimizadas para sistemas de núcleo que pueden mapear toda la memoria, lo cual mejora la tolerancia a fallos y eleva la portabilidad entre plataformas de hardware, según los defensores de esta tendencia. Sus detractores le achacan, fundamentalmente, mayor complejidad en el código, menor rendimiento o limitaciones en distintas funciones.

Una línea de trabajo interesante y complementaria a los micronúcleos son los **núcleos híbridos**, que fundamentalmente son micronúcleos con una parte de código "no esencial" en espacio de núcleo para que este se ejecute más rápido de lo que lo haría si estuviera en espacio de usuario.

Preocupados por el bajo rendimiento de los micronúcleos, muchos desarrolladores de los SO "modernos" adoptaron esta línea de trabajo, antes de que se demostrara que los micronúcleos pueden tener muy buenas prestaciones con la arquitectura adecuada y la sincronización optimizada. La mayoría de los SO "modernos" pertenecen a esta categoría, y de ellos los más populares son Windows NT, XNU y DragonFlyBSD, que es el primer sistema BSD que adopta una arquitectura de núcleo híbrido sin basarse en Mach.

XNU

XNU es ahora conocido como Darwin. Es el núcleo de Mac OS X y está basado en un micronúcleo modificado, debido a la inclusión de código del núcleo de FreeBSD en el núcleo original basado en Mach.

Normalmente se tiende a confundir el término *núcleo híbrido* con los núcleos monolíticos, ya que en los primeros hay una unión de conceptos tanto de arquitectura como de mecanismos propios del diseño monolítico y del diseño de los micronúcleos. En estos, generalmente, se observa el paso de mensajes y la migración de código no esencial hacia el espacio de usuario, pero se mantiene una cantidad de código no esencial en el propio núcleo por razones de rendimiento y prestaciones. Las ideas de un enfoque más pragmático de los objetivos que perseguía la organización cliente-servidor ha llevado a incluir el código ensamblador[5] en parte del núcleo y confiar al procesador una serie de funciones que normalmente se hacían por software, con lo que se obtienen una nueva serie de micronúcleos con un rendimiento muy alto comparativamente con los anteriores.

[(5)]En inglés, *assembler*.

Dos ideas complementarias a las organizaciones micronúcleos son los conceptos de *nanokernel*, o *picokernel*, y *exokernel*.

Los **nanokernels** se caracterizan por que la cantidad de código que se ejecuta en el modo privilegiado del hardware es muy pequeña (el término *picokernel* ha sido empleado a veces para resaltar porciones aún más pequeñas).

El término *nanokernel* fue definido por J. Shapiro en el artículo "The KeyKOS NanoKernel Architecture" como una respuesta sarcástica a Mach. En estas arquitecturas existe una capa de virtualización bajo un sistema operativo, lo que es más bien conocido como un hipervisor, y cuenta con una capa de abstracción de hardware que forma la parte más baja de nivel de núcleo, que a veces se utiliza para proporcionar la funcionalidad en tiempo real.

El **exokernel** es un núcleo del sistema operativo desarrollado por el MIT basado en la idea de que los desarrolladores puedan tomar tantas decisiones como sea posible acerca de abstracciones del hardware.

Los exonúcleos son pequeños, ya que la funcionalidad se limita a garantizar la protección y el multiplexado de los recursos, y son mucho más simples que los micronúcleos convencionales.

En este tipo de arquitecturas, las aplicaciones se consideran como bibliotecas de funcionamiento y pueden solicitar direcciones específicas de memoria, bloques de disco, etc. El núcleo solo asegura que el recurso solicitado está disponible, y la aplicación puede acceder a él. Este acceso al hardware de bajo nivel permite al programador implementar abstracciones eficientes (y de diferentes grados) y lograr mejoras aceptables en el rendimiento de un programa. Estos núcleos pueden ser considerados como basados en el principio *end-to-end*, pero para los sistemas operativos, en lugar de Internet, que es donde se aplica generalmente este principio.

1.5. Conclusiones

Si bien hemos considerado ejemplos de SO, podemos encontrar que en su gran mayoría no son diseño o implementaciones puras de estos modelos/arquitectura. Cada sistema operativo se orienta hacia una estructura determinada, pero generalmente encontramos una gran variedad de adaptaciones en función de los objetivos y las necesidades de cada uno de ellos. Por ello podemos concluir que cada SO utiliza lo mejor de cada estructura para lograr sus objetivos.

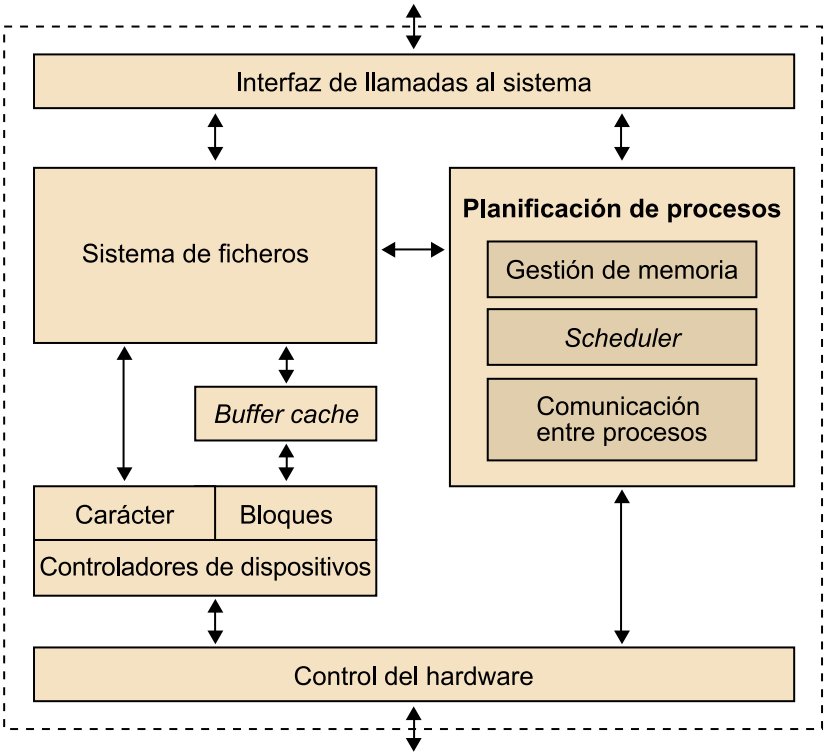
En general, se puede observar que cuando los servicios deben controlar principalmente dispositivos de hardware, es conveniente que las funciones estén divididas por capas u objetos. De esta manera, se dispone de código independiente del hardware y se facilita la configuración de los equipos. Pero, por el contrario, cuando se llega a la gestión más interna de los recursos del sistema (procesador, memoria, etc.), lo más importante es que el código sea eficiente, es decir, es mejor que la estructura se pueda derivar a por funciones.

La mayoría de los SO actuales son básicamente monolíticos o monolíticos híbridos, pero están dotados de una cierta estructura interna, por capas o por funciones, adecuada a cada servicio.

Un aspecto muy importante para todos los tipos de SO es que la definición de las fronteras entre los distintos módulos del software deben ser precisas, ya que estas fronteras deben determinar los puntos de entrada a las rutinas de servicio y facilitan la actualización, depuración y evolución de las funciones/servicios. Obviamente, esto debe ir acompañado por una definición estricta y no ambigua de los parámetros y una alta estabilidad entre las fronteras para que el código sea transportable cuando se hacen evoluciones o cambios de versión.

La figura 7 muestra en el nivel de bloques el núcleo del SO Unix:

Figura 7. Estructura del núcleo de UNIX

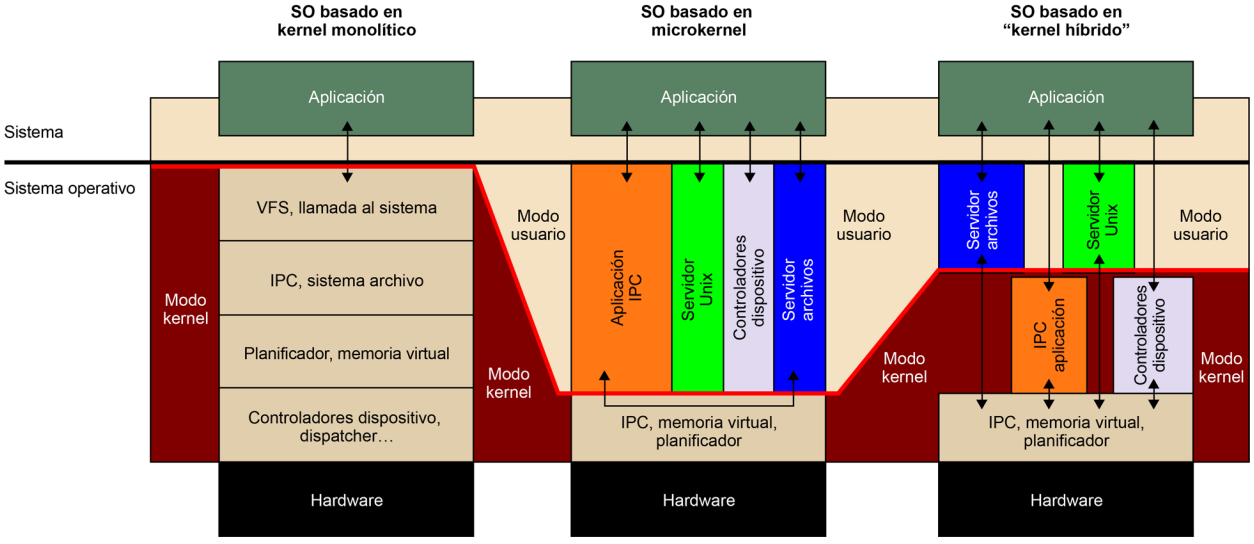


Unix

Unix es un sistema monolítico, aunque se presenta estructurado por funciones.

La figura 8 muestra una comparación entre un núcleo monolítico, una organización cliente-servidor y un núcleo híbrido en el nivel de funciones y módulos del sistema operativo.

Figura 8. Estructura de núcleo del SO UNIX en el nivel de bloques



Fuente: Adaptado de http://es.wikipedia.org/wiki/Archivo:OS-structure2.svg.

2. El núcleo del sistema operativo

Como concepto general, se puede decir que el núcleo, o *kernel* (de la raíz germánica *Kern*), es la parte del sistema operativo que se encuentra permanentemente cargada en la memoria (en su mayor parte, ya que los sistemas operativos modernos utilizan mecanismos para poder descargar parte del código que no utilizan y cargarlos solo cuando es necesario para así liberar memoria principal).

Es decir, podemos concluir que el **núcleo**, o ***kernel***, es un conjunto de procesos (que son programas en ejecución) que permiten el acceso a los distintos procesos de usuario (u otros procesos complementarios), al hardware del ordenador y de manera segura y ordenada, gestionando los recursos mediante servicios conocidos como llamadas al sistema.

Entre sus funciones está decidir cómo los programas se cargan en memoria, cómo se ejecutan (en qué orden y a qué tienen acceso) y durante cuánto tiempo (para evitar que un programa en ejecución –proceso– monopolice la CPU), con lo que gestionan un acceso de tiempo compartido[6] a todos los procesos pendientes de ejecución. Este modo de trabajo (por medio de llamadas al sistema) permite esconder la complejidad y proporciona una interfaz limpia y uniforme al hardware subyacente, lo que facilita su uso al programador de aplicaciones.

[6]En inglés, *timesharing*.

Proceso

Un proceso es un programa en ejecución que incluye el código binario, los espacios de memorias asignados y las estructuras de datos que el SO necesita para que un programa se pueda ejecutar.

El núcleo es un área de ejecución privilegiada y, por tanto, está dotado de las protecciones necesarias para evitar que se pueda ejecutar código fuera de su control. Cuando una rutina que se está ejecutando en el área de usuario pide un servicio de sistema –es decir, hace una llamada al sistema, por ejemplo en Unix un *pid= fork()* para crear un proceso hijo–, accede al núcleo mediante una excepción[7]. Esta excepción a la ejecución normal es una interrupción de la ejecución del programa en curso; por medio del vector de interrupciones se ejecuta el código correspondiente, que permitirá el cambio de contexto y el procesador cambiará el estado de ejecución a modo sistema, donde el propio sistema operativo puede supervisar todo lo que se pide que se haga y, si está permitido, lo podrá realizar, ya que este modo posee más privilegios que el modo usuario.

[7]En inglés, *trap*; también es un salto no programado.

Véase también

La utilización de los *traps* puede consultarse en la asignatura *Sistemas operativos*.

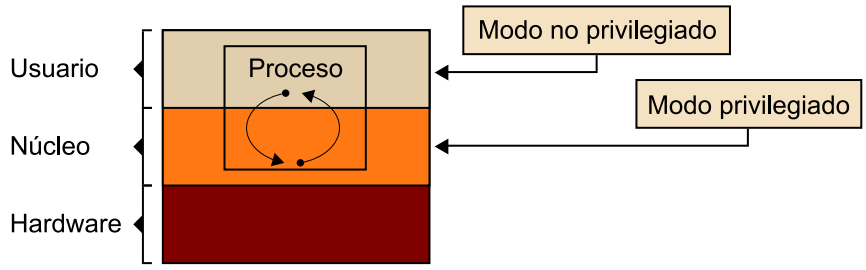
En el **modo kernel/supervisor/sistema**, como es denominado, se pueden verificar los permisos del solicitante, verificar si tienen los permisos adecuados y realizar la acción solicitada en caso afirmativo o denegarla en caso negativo. El código de las llamadas al sistema puede realizar cualquier tipo de acción protegida. Esto es necesario para que el sistema las realice y ejecute de manera segura y controlada. Como es el punto de entrada a las actividades protegidas que puede hacer el núcleo, nada puede escapar de pasar por esta "puerta", todo debe ser controlado y su acceso, muy seguro (existen técnicas conocidas como de "penetración" que intentan encontrar agujeros de seguridad o errores en el SO para saltarse este control y poder ejecutar sin el control del núcleo o en modo privilegiado).

Acciones protegidas

Algunos ejemplos de acciones protegidas son cambiar las protecciones de los ficheros, crear/borrar/modificar usuarios, gestionar las estructuras del SO o procesos –PCB, tablas de gestión de la memoria, entrada/salida, etc.– o toda la gestión de procesos/entrada-salida.

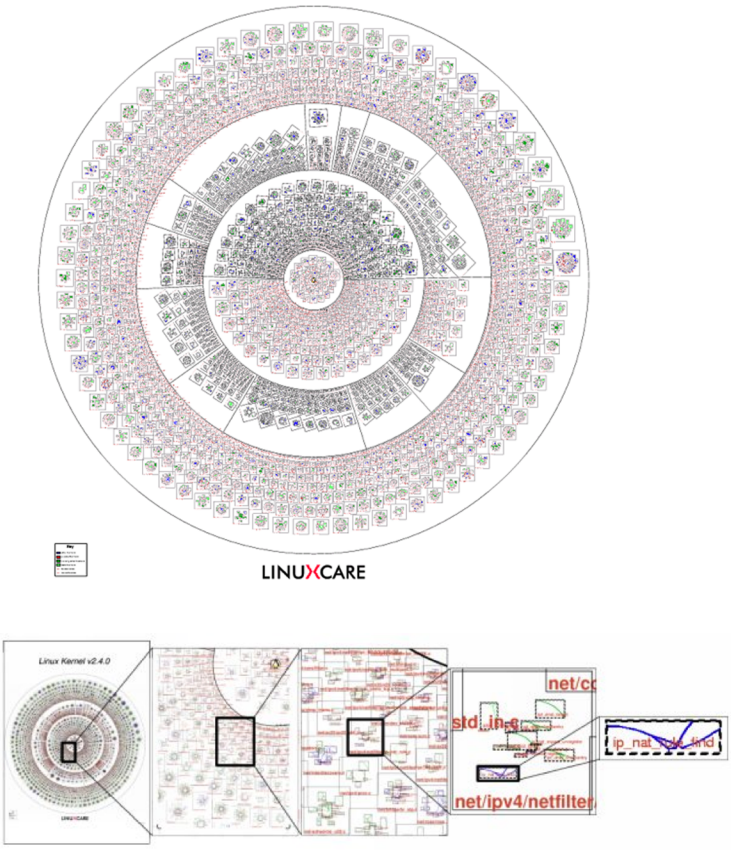
La figura 9 muestra este "cambio de contexto" cuando se ejecuta una llamada al sistema y el cambio o captura de la interrupción es transparente al usuario y se realiza utilizando un registro del procesador al que el usuario no tiene acceso. Aquí solo hemos usado dos niveles –usuario y sistema/supervisor–, pero los SO actuales cuentan con diferentes niveles –soportados por el hardware–, que generalmente van incrementando sus privilegios en función de su posición.

Figura 9. Modos de ejecución de un proceso



La figura 10 muestra el núcleo Linux en el nivel de llamadas y niveles de ejecución realizado por el proyecto Free Code Graphing Project, donde se pueden encontrar las herramientas e instrucciones para generarlo.

Figura 10. Núcleo Linux en el nivel de llamadas y niveles de ejecución



Fuente: Free Code Graphing Project <http://fcgp.sourceforge.net/>

2.1. El núcleo de Unix

El **núcleo Unix** es un programa escrito casi en su totalidad en lenguaje C, con excepción de una parte del manejo de interrupciones (que está en lenguaje ensamblador), y da soporte multiusuario y multitarea concurrente utilizando algoritmos de distribución equitativa de los recursos.

Entre sus acciones principales el núcleo se encarga de:

- Gestión de procesos, asignación de tiempos de ejecución y sincronización.
- Gestión de memoria principal y de intercambio para que los procesos puedan ser ejecutados en función de los criterios programados.
- Administración/gestión de: sistema de archivos, acceso/protección y administración de usuarios, y comunicación entre usuarios y entre procesos.
- Supervisión/gestión y administración de: dispositivos de entrada/salida, intercambio de datos, control y sincronización entre memoria y periféricos.

En su mayor parte reside siempre en la memoria principal y tiene el control del ordenador, es decir, ningún otro proceso puede interrumpirlo y solo pueden llamarlo para que proporcione algún servicio de los mencionados.

Transferencia de datos

Cada vez que se presiona una tecla, se interrumpe la CPU, el sistema operativo pasa a modo supervisor y la rutina correspondiente del núcleo se encarga de efectuar la operación de transferencia devolviendo al programa la tecla en un *buffer* en el espacio de usuario.

El núcleo consta de dos partes principales: la sección de control de procesos y la de control de dispositivos. La primera asigna recursos, programas, procesos y atiende sus requerimientos de servicio; la segunda, supervisa la transferencia de datos entre la memoria principal y los dispositivos del ordenador.

Inicialización del hardware

Uno de los primeros dispositivos que se inicializa es el reloj, que proporciona interrupciones periódicas para que ningún proceso monopolice la CPU y prepara las estructuras de datos, por ejemplo, las de almacenamiento temporal para transferencia de información entre dispositivos y procesos, la de descriptores de archivos, las de memoria principal, etc.

Al inicio del SO se carga el código del núcleo en la memoria principal desde el dispositivo de arranque (disco, memoria USB, etc.) y esta operación se denomina *bootstrap*. Como primara parte se inicializa el hardware. A continuación se inicializa un proceso especial, llamado **proceso 0** (también llamado *intercambiador*) mediante una llamada al sistema (*fork*); dado que Unix crea los procesos hijos por duplicación del padre (y cargando el código ejecutable correcto), es necesario crear el padre de todos asignando una estructura de datos y asignando los registros a una sección especial de la memoria, denominada tabla de procesos (PCB), que contendrá los descriptores de cada uno de los procesos existentes en el sistema.

Después de crear el proceso 0, se hace una copia de este, con lo que se crea el proceso 1, que será encargado de "poner en marcha" los otros procesos del núcleo. Es decir, se inicia una cadena de activaciones de procesos, entre los cuales destaca el planificador de CPU[8], que será el responsable de decidir qué proceso se ejecutará y cuáles van a entrar o salir de la memoria central. A partir de ese momento se conoce el proceso número 1 como proceso de inicialización del sistema (**init**).

(8)En inglés, *dispatcher*.

Este proceso (*init*) será el responsable de crear las estructuras de datos para atender a los usuarios y ejecutar el proceso de *login* (de inicio de conexión), que establecerá la interacción entre los diferentes usuarios del sistema por medio del intérprete del comandos (*shell*).

2.2. El núcleo Linux

El **núcleo Linux** es un sistema operativo de código abierto donde el usuario puede modificar, corregir y generar nuevas versiones y actualizaciones rápidamente (y con todas las garantías legales de sus acciones) siempre y cuando tenga los conocimientos adecuados para hacerlo.

Esto permite a los usuarios críticos controlar mejor sus aplicaciones y el propio sistema, así como poder montar sistemas con el SO original generando sus propias versiones con las partes que se desee o personalizarlo para cumplir los objetivos propuestos (ya que Linux también permite disponer de un sistema operativo con código abierto, desarrollado por una comunidad de progra-

madores coordinados a través de Internet y accesible por disponer del código fuente y abundante documentación) o para la producción final de los sistemas GNU/Linux adaptados a necesidades individuales o de un determinado colectivo.

Al disponer del código fuente, se pueden aplicar mejoras y soluciones de manera inmediata, a diferencia del software propietario, donde se debe esperar a las actualizaciones del fabricante. Se puede además personalizar el núcleo tanto como sea necesario, lo cual es un requisito esencial, por ejemplo, en aplicaciones de alto rendimiento, críticas en el tiempo o en soluciones con sistemas empotrados (como dispositivos móviles).

El núcleo Linux fue desarrollado en sus inicios por un estudiante finlandés llamado Linus Torvalds, en 1991, con la intención de realizar una versión parecida a Minix (sistema operativo didáctico de A. Tanenbaum). La primera versión publicada oficialmente fue la de Linux 1.0 en marzo de 1994 (para i386). Linux 1.2 fue publicado en marzo de 1995 con versiones para diferentes arquitecturas, como Alpha, Sparc y Mips. Linux 2.0 (1996) añadió más arquitecturas y fue la primera versión en incorporar soporte multiprocesador. En la actualidad (2011), se desarrolla sobre la rama 2.6 del núcleo (concretamente la versión estable es la 2.6.31 junio del 2011 y se trabaja en la *mainline* de RC1 de la versión 3.0), en la que se ha agregado gran cantidad de soporte a hardware, mejoras en el soporte multiprocesador, mejor respuesta de la planificación de CPU, el uso de hilos[9] en el núcleo, soporte de arquitecturas de 64 bits, soporte de virtualización y una mejor adaptación a dispositivos móviles.

[9] En inglés, *threads*.

El código de Linux ha crecido en complejidad pero ha alcanzado un grado de madurez y estabilidad notables. Solo basta con ver la relación entre versiones (años) y líneas de código (en miles): V0.01 (1991) 10, V1.0 (1994) 176, V2.0 (1996) 649, V2.4 (2001) 3.378, V2.6 (2003) 5.930. De las diez mil líneas de la primera versión se ha pasado (como se puede comprobar) a casi seis millones en la rama 2.6 y donde las últimas versiones de esta rama se mueven entre los diez y quince millones de líneas.

En la rama del 2.6, durante su desarrollo se aceleraron de manera significativa los trabajos del núcleo, ya que tanto Linus Torvalds como Andrew Morton (que mantienen varias de las ramas de Linux 2.6 en desarrollo) se incorporaron (durante el 2003) al Open Source Developement Laboratory (OSDL), un consorcio de empresas cuyo fin es promocionar el uso de Open Source y GNU/Linux en la empresa (en el consorcio se encuentran, entre otras muchas empresas con intereses en GNU/Linux: HP, IBM, Sun, Intel, Fujitsu, Hitachi, Toshiba, Red Hat, Suse, Transmeta, etc.). En la actualidad, OSDL ya no existe y se ha reconvertido en la fundación The Linux Foundation (http://www.linuxfoundation.org).

La numeración a partir de la versión 2.6 ha cambiado en relación con las anteriores, si bien conservan algunos aspectos básicos: la versión viene indicada por unos números X.Y.Z, donde normalmente X es la versión principal –que

representa los cambios importantes del núcleo–, Y es la versión secundaria –y habitualmente implica mejoras en las prestaciones del núcleo con las siguientes consideraciones: Y es par en los núcleos estables e impar en los desarrollos o pruebas–, y Z es la versión de construcción, que indica el número de la revisión de X.Y, en cuanto a parches o correcciones realizadas.

Las distribuciones no suelen incluir la última versión del núcleo, sino la que los integradores de la distribución hayan probado con más frecuencia y puedan verificar que es estable para el software y los componentes que ellos incluyen. La versión así definida con cuatro números es la que se considera estable (*stable*). También se usan otros esquemas para las distintas versiones de prueba (normalmente no recomendables para entornos de producción), como sufijos -rc (*release candidate*), -mm, que son núcleos experimentales con gran introducción de parches que suponen nuevas prestaciones adicionales, como pruebas de diferentes técnicas novedosas, o los -git, que son una especie de "foto" diaria del desarrollo del núcleo. Estos esquemas de numeración están en constante cambio para adaptarse al modo de trabajar de la comunidad del núcleo y a sus necesidades para acelerar el desarrollo.

Versión estable

Un ejemplo de versión estable es: stable: **2.6.39.1** 2011-06-03, como podéis ver en http://kernel.org.

Para obtener el último núcleo publicado (que normalmente se denomina *vanilla* o *pristine*), hay que acudir al archivo de núcleos Linux (http://kernel.org) o al repositorio local (http://www.es.kernel.org). También podrán encontrarse aquí algunos parches al núcleo original, que corrigen errores detectados a posteriori de la publicación del núcleo.

Las características del núcleo Linux que se pueden destacar son:

- **Núcleo de tipo monolítico**: básicamente es un gran programa creado como una unidad, pero conceptualmente dividido en varios componentes lógicos.

- Soporte para carga y descarga de porciones del núcleo bajo demanda llamadas **módulos** y suelen ser características del núcleo o controladores de dispositivo.

- **Hilos de núcleo**: se utilizan varios hilos[10] de ejecución internos al núcleo, que pueden estar asociados a un programa de usuario o a una funcionalidad interna del núcleo (aspectos especialmente cuidados desde la versión 2.6 y adaptado a las arquitecturas *multicore*).

(10)En inglés, *threads*.

- Soporte de **aplicaciones multihilo**: soporte de aplicaciones de usuario de tipo multihilo[11], ya que muchos paradigmas de computación de tipo cliente/servidor necesitan servidores capaces de atender múltiples peticiones simultáneas dedicando un hilo de ejecución a cada petición o grupo de ellas.

(11)En inglés, *multithread*.

- Núcleo de tipo apropiativo (a partir de la versión 2.6): como con la mayoría de los sistemas operativos de propósito general, Linux siempre ha prohibido al planificador de procesos la interrupción de la ejecución de una llamada al sistema. Por lo tanto, una vez que una tarea se encuentra en una llamada al sistema, controla el procesador hasta que esta regrese, sin importar el tiempo que pueda tardar. A partir del *kernel* 2.6, el *kernel* es apropiativo, por lo que una tarea del núcleo ahora pueda ser suspendida y otra tarea de mayor prioridad podría continuar su ejecución. En el núcleo 2.6, el código del *kernel* contiene puntos de apropiación[12], que son instrucciones que permiten al planificador ejecutarse y (posiblemente) bloquear el proceso en curso con el fin de programar la ejecución de un proceso de mayor prioridad. Con esto, Linux 2.6 evita demoras injustificadas en las llamadas al sistema con la comprobación periódica de los puntos de apropiación. Durante estas pruebas, el proceso en curso puede bloquearse y dejar que otro proceso se ejecute. De esta forma, en Linux 2.6, el núcleo se puede interrumpir en mitad de una tarea y permitir que otras de mayor prioridad puedan continuar su ejecución. Esta característica es necesaria para SO que trabajan en tiempo real y con esta versión del núcleo ya no es necesaria una versión del núcleo específica (como ocurría anteriormente) de Linux para RealTime (RT).

(12)En inglés, *preemption points*.

- Soporte para multiprocesador, tanto lo que se denomina **multiprocesamiento simétrico** (SMP) como *multicore* (de 2 hasta 64 CPU). Del rendimiento del SMP[13] depende en gran medida la adopción de Linux en los sistemas empresariales en la faceta de sistema operativo para servidores.

(13)SMP es la sigla de *symmetric multi-processing*, multiprocesamiento simétrico.

- **Sistemas de archivos**: el núcleo tiene una excelente arquitectura de los sistemas de archivos, ya que el trabajo interno se basa en una abstracción de un sistema virtual (VFS[14]), que puede ser adaptada fácilmente a cualquier sistema real. Como resultado, Linux es quizá el sistema operativo que más sistemas de ficheros soporta, desde su propio ext2 inicial, hasta msdos, vfat, ntfs, sistemas con *journal* como ext3, ext4, ReiserFS, JFS(IBM), XFS(Silicon), NTFS, iso9660 (CD), udf, etc.

(14)VFS es la sigla de *virtual file system*.

Dentro de los aspectos no técnicos se puede destacar que:

- Linux **es código libre y abierto**: junto con el software GNU, y el incluido en cualquier distribución, se puede tener un sistema tipo Unix completo prácticamente por el coste del hardware; y por la parte de los costes de la distribución GNU/Linux, se puede obtener sin coste.

- Linux **es personalizable**: la licencia GPL permite leer y modificar el código fuente del núcleo (siempre que se tengan los conocimientos adecuados) y es posible ejecutarlo en hardware antiguo bastante limitado (es posible crear un servidor de red con un i386 con 4 MB de RAM).

- Linux es un **sistema de altas prestaciones**: el objetivo principal en Linux es la eficiencia y se intenta aprovechar al máximo el hardware disponible.

- **Alta calidad y estabilidad**: los sistemas GNU/Linux son muy estables, con una baja proporción de fallos, y reducen el tiempo dedicado a mantener los sistemas.

- El núcleo es bastante **reducido y compacto**: es posible colocarlo, junto con algunos programas fundamentales, en un USB de 256 Mb o más pequeño incluso.

- Linux es **compatible con una gran parte de los sistemas operativos**, puede leer ficheros de prácticamente cualquier sistema de ficheros y puede comunicarse por red para ofrecer y recibir servicios de cualquiera de estos sistemas. Además, también con ciertas librerías puede ejecutar programas de otros sistemas (como MS-DOS, Windows, BSD, Xenix, etc.) en la arquitectura x86 o virtualizar máquinas completas.

- Linux **dispone de un amplísimo soporte**: no hay ningún otro sistema que tenga la rapidez y cantidad de actualizaciones que Linux, ni en los sistemas propietarios. Para un problema determinado, hay infinidad de listas de correo y foros que en pocas horas pueden permitir solucionar cualquier problema.

El único problema está en los controladores de hardware recientes, que muchos fabricantes todavía se resisten a proporcionar, si no es para sistemas propietarios. Pero esto está cambiando poco a poco y varios de los fabricantes más importantes de sectores como tarjetas de vídeo (NVIDIA, ATI) e impresoras (Epson, HP) comienzan ya a proporcionar los controladores para sus dispositivos, bien sean de código abierto, o bien binarios usables por el núcleo.

2.3. Configuración y compilación del núcleo Linux

La personalización del núcleo es un proceso que se debe planificar y realizar con cuidado, además de necesitar conocimientos exhaustivos de los aspectos de configuración, ya que, en caso de no hacerlo correctamente, puede quedar inutilizado o inestable. Por ello se recomienda hacer previamente copias de seguridad de los datos de usuarios, datos de configuraciones o una copia de seguridad completa del sistema. También es recomendable disponer de algún CD/USB de arranque (o distribución LiveCD con herramientas de rescate) que permita recuperar el sistema en caso de problemas. Actualmente muchos de los LiveCD de las distribuciones ya proporcionan herramientas de rescate suficientes para estas tareas, aunque también existen algunas distribuciones especializadas para ello.

Para compilar un núcleo de la versión 2.6.x (recomendable), se deben descargar los paquetes fuentes de kernel.org o los proporcionados por la distribución verificando que en el nombre del archivo figure la versión 2.6.xx (donde xx es el número de revisión del núcleo estable). Se realizará la extracción en el directorio /usr/src/, que será el directorio que se usará para la compilación. Si el paquete obtenido es de tipo gzip (tar.gz):

```
zip -cd linux-2.6.39.1.tar.gz | tar xvf -
```

o, si es de tipo bzip2 (tar.bz2):

```
bzip2 -dc linux-2.6.39.1.tar.bz2 | tar xvf -
```

Es recomendable leer el fichero README, que está situado en el directorio raíz de los paquetes fuente. Una vez extraídos los archivos fuente en el directorio de compilación, se comprueban los paquetes del entorno de compilación necesario. Normalmente es necesario instalar algunos paquetes, como, por ejemplo, build-essentials, libncurses-dev, libqt3-dev, libgtk2-dev (los nombres dependen de la distribución), que incorporan el entorno básico de compilación (gcc) y las necesidades para la construcción de los menús posteriores de compilación de los archivos fuente (*make menuconfig* o *xconfig*).

Todo el proceso de configuración/compilación se puede hacer como usuario normal; solo algunas partes muy concretas, como la instalación final del núcleo o de módulos, hay que hacerlas usando el usuario *root*.

Construcción como usuario normal

Es el procedimiento recomendable para no generar problemas de seguridad si los fuentes no son de confianza.

Pasamos a iniciar el proceso en el directorio de los fuentes; por ejemplo, con /usr/src/linux-2.6.39.1 se realiza la limpieza de cualquier actividad anterior:

```
ln -s /usr/src/linux-2.6.39.1 /usr/src/linux
cd /usr/src/linux
make clean
make mrproper
```

El siguiente paso es la configuración de parámetros, que será guardada en el archivo .config y que se puede usar de configuraciones anteriores que se hayan realizado o partir de la configuración del núcleo actual existente (que se puede verificar con *uname -r*). También si se tiene una versión configurada se puede partir de ella, ya que estará en /boot/config-version-kernel y solo se deberá copiar este fichero como .config en el directorio raíz de los paquetes fuente del núcleo con:

```
cp /boot/config-'uname -r' ./.config
```

La configuración se realiza con el comando *make opción*, donde *opción* puede ser:

- *make config*: interfaz de texto.
- *make menuconfig*: interfaz basada en menús textuales.
- *make xconfig*: interfaz gráfica basada en toolkit Qt de KDE.
- *make gconfig*: interfaz gráfica basada en toolkit Gtk de Gnome.
- *make oldconfig*: se basa en el fichero .config previo y pregunta textualmente por las opciones nuevas que no estaban previamente en la configuración antigua (de .config).

Por lo que si se selecciona la primera opción, será:

```
make gconfig
```

respondiendo a las opciones de configuración (se puede consultar la ayuda contextual ofrecida por el programa) y a continuación se procederá a la construcción de la imagen binaria del núcleo:

```
make
```

Es necesario indicar que para la generación de estos núcleo genéricos (habitualmente denominados *vanilla* o *pristine*), también existen procedimientos más simples.

Procedimientos más simples

En Debian es posible sustituir el actual *make* por *make deb-pkg*, que obtendrá paquetes binarios .deb instalables en el sistema y actualizará el *bootloader* y los ficheros necesarios.

Posteriormente, se deberán compilar los módulos indicados en la configuración y su posterior instalación (en /lib/modules/version-kernel):

```
make modules modules_install
```

A continuación se debe copiar la imagen a su posición final, suponiendo i386 como arquitectura (id con cuidado en este punto, ya que hay detalles en los nombres y directorios que cambian en función de la distribución GNU/Linux):

```
cp arch/i386/boot/bzimage /boot/vmlinuz-2.6.xx.img
```

Luego se deberá crear la imagen de disco RAM initrd (necesaria en la mayoría de los casos) con las utilidades necesarias según la versión de la distribución:

```
mkinitramfs -o /boot/initrd-2.6.39.1.img
/lib/modules/2.6.39.1
```

Y, finalmente, actualizar el *bootloader* (LiLo o Grub 1.x o Grub 2.x) según sea el que utilice la distribución, y mover los archivos complementarios (vmlinuz, system.map e initrd) a /boot, que puede realizarse también normalmente con el proceso (en root):

```
make install
```

Pero hay que tener en cuenta que esta opción realiza todo el proceso y actualizará el *bootloader* quitando antiguas configuraciones o modificándolas. Por ello, es necesario guardar las configuraciones del *bootloader* (LiLo en /etc/lilo.conf o Grub1.0 en /etc/grub/menu.lst o Grub2.x en /etc/grub.d/10_linux).

Grub2.0

Sobre Grub2.0 podéis consultar su guía <http://www.guia-ubuntu.org/index.php?title=GRUB>

Respecto a la creación del initrd, en Fedora/Red Hat este se creará automáticamente con la opción *make install*. En Debian se deberá o bien usar los comandos del siguiente subapartado, o bien crearlo explícitamente con *mkinitramfs* o una utilidad denominada update-initramfs, especificando la versión del núcleo (se asume que este se llama vmlinuz-version dentro del directorio /boot):

```
update-initramfs -c -k 'version'
```

Una vez realizadas estas secuencias de comandos, se puede reiniciar el ordenador con *shutdown -r now*, y elegir el nuevo núcleo y, si todo ha ido bien, se podrá comprobar con *uname -r* para verificar la nueva versión del núcleo mirando /var/log/messages y el comando *dmesg*, para examinar el registro de salida de mensajes producidos por la carga del nuevo núcleo en el arranque y detectar si ha aparecido algún problema de funcionalidad o con algún dispositivo concreto.

Ubuntu, Fedora y Debian

Existe una guía completa <https://help.ubuntu.com/community/Kernel/Compile> para compilar Linux en las distribuciones Ubuntu. También hay una guía de Fedora <http://fedoraproject.org/wiki/Building_a_custom_kernel> y otra guía para Debian <http://kernel-handbook.alioth.debian.org/>.

2.4. Compilación del núcleo en Debian (Debian Way)

En Debian, además del método general comentado en el subapartado anterior, hay que añadir la configuración por el método denominado Debian Way[15]. Es un procedimiento que permite construir el núcleo de una manera flexible y rápida, adaptada a la distribución.

(15) http://kernel-handbook.alioth.debian.org/index.html#contents

Para el proceso se necesitará una serie de utilidades que será necesario instalar: kernel-package, ncurses-dev, fakeroot, WGET y bzip2. Debian ha cambiado varias veces la gestión de sus paquetes asociados a los paquetes fuente del núcleo. A partir de la versión 2.6.12, es habitual encontrar en el repositorio Debian una versión linux-source-version que contiene la versión de los fuentes del núcleo con los últimos parches aplicados. Esta versión del paquete de los fuentes del núcleo es la que usaremos para crear un núcleo personalizado.

A partir de la versión mencionada (2.6.12) se incluyó en los repositorios de Debian un nuevo paquete, denominado simplemente linux-2.6, que incluye los paquetes fuente y utilidades preparadas para generar el núcleo en la mencionada Debian Way. Este paquete fuente es usado para crear los paquetes bi-

narios de la distribución asociados al núcleo y también es el indicado para usar en caso de querer aplicar parches al núcleo actual de la distribución, o por si se desea probar modificaciones del núcleo en el nivel de código.

Para la compilación del núcleo actual de acuerdo al segundo procedimiento, se emplea el segundo paquete: apt-get source linux-2.6.

En este caso descargará y descomprimirá los paquetes fuente dejándolos en un árbol de directorios a partir del directorio linux-2.6-version. A continuación mostramos las herramientas que se necesitarán:

```
apt-get install build-essential fakeroot
apt-get build-dep linux-2.6
cd linux-2.6-version
fakeroot debian/rules binary
```

Con lo cual se dispondrá de los paquetes binarios de núcleo (ficheros *.deb) disponibles para la instalación (vía *dpkg*).

Si se considera la primera opción comentada de los paquetes donde se desea modificar y cambiar ciertas opciones del núcleo para crear una versión personal de este, se deberá realizar un proceso semejante mediante un paso de personalización típico (por ejemplo, mediante *make menuconfig*). Los pasos son, en primer lugar, la obtención y preparación del directorio (es equivalente a obtener los paquetes fuentes desde kernel.org):

```
apt-get install linux-source-2.6.xx
tar -xvjf /usr/src/linux-source-2.6.xx.tar.bz2
cd linux-source-2.6.xx
```

A continuación, se realiza la configuración de parámetros:

```
make menuconfig
```

Y la construcción final del núcleo:

```
make clean
make KDEB_PKGVERSION=custom.1.0 deb-pkg
```

Donde se crea un identificador para el núcleo construido (custom.1.0) que se añadirá al nombre del paquete binario del núcleo, posteriormente visible en el arranque con el comando *uname –a*. El proceso finalizará con la obtención del paquete asociado a la imagen del núcleo, que se podrá finalmente instalar:

```
dpkg -i ../linux-image-2.6.xx_custom.1.0_i386.deb
```

Este comando descomprimirá e instalará el núcleo y generará una imagen initrd adicional si fuera necesario. Además, configurará el *bootloader* con el nuevo núcleo por defecto y con *shutdown -r now* se podrá probar el arranque con el nuevo núcleo.

Nueva configuración del *bootloader*

Es muy importante no olvidar hacer antes una copia de seguridad del *bootloader* para no perder ninguna configuración estable.

Es importante comentar que una característica de Debian es la existencia de utilidades para añadir módulos dinámicos de núcleo proporcionados por terceros. En particular, la utilidad *module-assistant* permite automatizar todo este proceso a partir de los paquetes fuente del módulo. Para ello, es necesario disponer de los *headers* del núcleo instalado (disponible en el paquete linux-headers-version) o de los paquetes fuente que utilizamos en su compilación. A partir de aquí *module-assistant* puede utilizarse interactivamente y seleccionar entre una amplia lista de módulos registrados previamente en la aplicación, y puede encargarse de descargar el módulo, compilarlo e instalarlo en el núcleo existente.

También, se puede utilizar en la línea de comandos (*m-a* es el comando en línea de *module-assistant*):

```
m-a prepare
m-a auto-install nombre_módulo
```

Este preparará el sistema para posibles dependencias, descarga de los fuentes del módulo, compilación y, si no hay problemas, instalación en el presente núcleo.

2.5. Proceso de arranque en Linux

En Linux la **secuencia de arranque** comienza por el inicio desde la BIOS, pasa al gestor de arranque y luego al núcleo[16]. El núcleo inicia el planificador (para permitir la multitarea) y ejecuta el primer espacio de usuario (es decir, fuera del espacio del núcleo) y el programa de inicialización (que establece el entorno de usuario y permite la interacción del usuario y el inicio de sesión), momento en el que el núcleo se inactiva hasta que sea llamado externamente.

(16)En inglés, *kernel*.

La etapa del cargador de arranque puede no ser necesaria, ya que existen determinadas BIOS que pueden cargar y pasar el control a Linux sin hacer uso del cargador (el proceso de arranque será diferente dependiendo de la arquitectura del procesador y la BIOS), pero los pasos más comunes son:

a) BIOS: tareas de inicio específicas del hardware.

b) Carga y ejecución del código de la partición de arranque del dispositivo de arranque designado, que contiene un gestor de arranque Linux (normalmente se hace por fases).

c) El gestor de arranque presenta al usuario un menú de opciones posibles de arranque.

d) Carga el sistema operativo, que se descomprime en la memoria, y establece las funciones del sistema como del hardware esencial y la paginación de memoria, antes de llamar a la función *start_kernel()*.

e) *start_kernel()* realiza la configuración del sistema antes de continuar por separado el proceso inactivo y planificador, y el proceso de *init* (que se ejecuta en el espacio de usuario).

Configuración del sistema

La configuración del sistema incluye procesos como interrupciones, gestión de memoria, la inicialización de dispositivos, controladores, etc.

f) El planificador toma control efectivo de la gestión del sistema, y el núcleo queda inactivo.

g) *init* ejecuta secuencias de comandos[17] necesarios para configurar todos los servicios y estructuras que no sean del sistema operativo, y finalmente el proceso que presentará la pantalla de inicio de sesión.

[17]En inglés, *scripts*.

Durante el apagado, se llama a *init* para cerrar todas las funcionalidades del espacio de usuario de una manera controlada, de nuevo por medio de secuencias de comandos, tras lo cual el *init* termina y el núcleo ejecuta el apagado.

Es interesante mencionar la función del cargador de arranque[18], que es un programa diseñado exclusivamente para cargar un sistema operativo en memoria. Como en la mayoría de las arquitecturas, este programa se encuentra en el MBR[19], que es de 512 bytes. Este espacio no es suficiente para cargar un sistema operativo y por ello el cargador de arranque consta de varias etapas. Las primeras operaciones las realiza la BIOS, donde se localiza el sector de arranque (o MBR) y se carga el cargador de este sector (normalmente una parte de LILO o GRUB, que son los cargadores habituales en Linux).

[18]En inglés, *bootloader*.

[19]MBR es la sigla de *master boot record*, sector de arranque.

LILO y GRUB

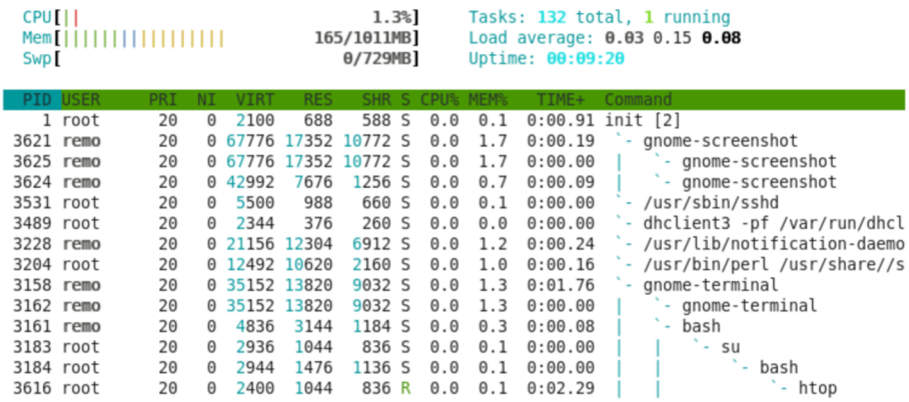
LILO y GRUB difieren en algunos aspectos, ya que el primero (LILO) "no conoce" los sistemas de archivos, por lo que utiliza desplazamientos de disco sin procesar y la BIOS para cargar los datos. Se carga el código del menú y, a continuación, en función de la respuesta, carga el sector MBR del disco (como en Microsoft Windows) o la imagen del núcleo Linux. GRUB, por el contrario, "entiende" la estructura de los sistemas de archivos comunes ext2, ext3 y ext4 y almacena sus datos en un archivo de configuración en una partición del sistema (en vez de en el MBR). Por ello GRUB permite modificar los parámetros o el arranque *on fly* si está mal configurado o corrupto.

Para la carga en GRUB (cargador más utilizado por sus ventajas en las actuales distribuciones), la primera etapa del cargador la lee la BIOS desde el MBR y esta carga el resto del gestor de arranque (segunda etapa). Si la segunda etapa está en una unidad de tamaño superior a 1.024 cilindros o tipo LBA grande, se carga una fase intermedia 1.5, que contiene código adicional para permitir cilindros por encima de 1.024. La segunda etapa del gestor de arranque ejecuta y muestra el menú de inicio de GRUB, que permite al usuario elegir un sistema operativo y examinar y modificar los parámetros de inicio. A continuación de

elegir un sistema operativo, se carga y se le pasa el control. GRUB soporta métodos de arranque directo, arranque *chain-loading*, LBA, ext2, ext3, ext4 y hasta "un presistema operativo en máquinas x86 totalmente basado en comandos".

La figura 11 muestra la ejecución del comando *htop* (para instalar este comando en Debian, haced ***apt-get install htop***), en el que se puede ver el árbol de procesos, el proceso *init* como padre de todos los de información general de los procesos y carga de los recursos.

Figura 11. Ejecución del comando *htop*



2.6. El núcleo de Windows NT

La familia de los sistemas operativos Windows NT (WNT) de Microsoft está formada por Windows 2000, XP, Server2003/2008, Vista y W7. Todos ellos disponen de multitarea apropiativa y soportan multiprocesamiento simétrico. La arquitectura de WNT[20] es considerada una arquitectura híbrida porque el núcleo contiene tareas tales como el Windows Manager y el IPC Manager, pero existen otras tareas que se ejecutan en modo usuario y el punto de ruptura entre unas y otras ha ido cambiando de versión en versión. No obstante, *user-mode driver framework* en Windows Vista y *user-mode thread scheduling* en Windows 7 han traspasado más funcionalidad del modo supervisor al modo usuario y solo a partir de Server 2008 existen versiones de 64 bits.

[20]WNT es la sigla de Windows NT.

La arquitectura de WNT es modular y está basada en dos capas principales:

a) Modo usuario: los programas y subsistemas están limitados a los recursos del sistema a los que tienen acceso.

b) Modo núcleo: acceso total a la memoria del sistema y a los dispositivos externos (se considera un núcleo híbrido, aunque es necesario remarcar que este término está en discusión, ya que este núcleo es esencialmente un núcleo monolítico que está estructurado al estilo de un micronúcleo) con los siguientes componentes:

- Un núcleo híbrido.
- Una capa de abstracción de hardware (HAL).
- Controladores[21].
- Executive: sobre el cual son implementados todos los servicios de alto nivel.
- Librerías dinámicas (como, por ejemplo, ntoskrnl.exe).

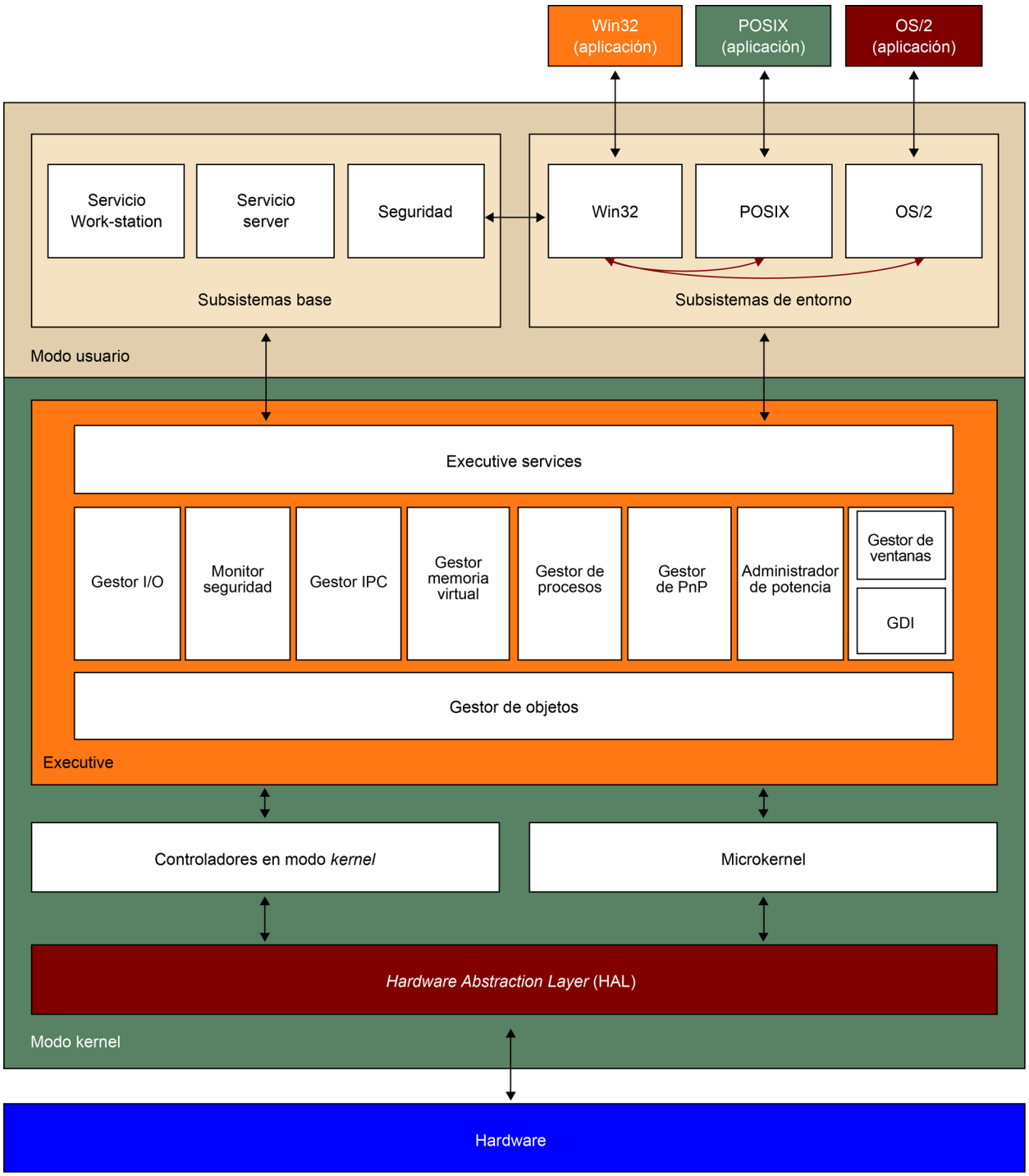
(21)En inglés, *drivers*.

Una parte importante de la arquitectura es el *executive* que realiza la interfaz de todos los subsistemas del modo usuario (entrada/salida, la gestión de objetos, la seguridad y la gestión de procesos). El núcleo se sitúa entre la capa HAL[22] y el Executive para proporcionar sincronización multiprocesador, hilos, gestión de interrupciones y envío de excepciones. En el núcleo también existen tres niveles de controladores en el modo núcleo: de alto nivel, intermedios y de bajo nivel. El modelo de *windows driver model* (WDM) se encuentra en la capa intermedia y fue diseñado principalmente para mantener la compatibilidad entre W98y W2k. Los controladores de bajo nivel son los controladores de dispositivos de WNT que actúan directamente al dispositivo o dispositivos PnP.

(22)HAL es la sigla de *hardware abstraction layer*, capa de abstracción de hardware.

La figura 12 muestra los módulos y su relación en un núcleo NT.

Figura 12. Módulos y su relación en un núcleo NT



Fuente: Adaptado de <http://es.wikipedia.org/wiki/Archivo:Windows_2000_architecture.svg>

Como se puede observar, el modo núcleo está formado por servicios *executive*, que a su vez están formados por varios módulos que realizan tareas específicas, controladores de núcleo, un núcleo y una capa de abstracción del hardware o HAL. El Executive se relaciona con todos los subsistemas del modo usuario y se encarga de la I/O, gestión de objetos, seguridad y gestión de procesos. El gestor de objetos es un subsistema especial del Executive por el que todos los otros subsistemas, especialmente las llamadas al sistema, deben pasar para obtener acceso a los recursos de W2K, conocidos como servicio de infraestructuras de gestión de recursos.

El núcleo del SO se encuentra entre la HAL y el Executive y proporciona sincronización multiprocesador, hilos y envío/planificación de interrupciones, gestión de interrupciones y envío de excepciones; también es responsable de la inicialización de controladores de dispositivos. Además, W2k utiliza los controladores de dispositivo del modo núcleo para permitirle interactuar con los dispositivos hardware. Todos los dispositivos son vistos por el modo usuario como un objeto archivo en el gestor de entrada/salida que los define tanto como objetos archivo, dispositivo o controlador.

La HAL se encuentra entre el hardware físico del ordenador y el resto del sistema operativo y fue diseñada para ocultar las diferencias de hardware, y por tanto proporciona una plataforma consistente en la que las aplicaciones pueden ejecutarse. La HAL incluye código dependiente del hardware que controla las interfaces de E/S, controladores de interrupciones y múltiples procesadores.

3. Soporte del hardware

Como se ha explicado anteriormente, una de las funciones principales del SO es esconder las complejidades del hardware subyacente a usuario presentando una "máquina virtual simplificada" que permite trabajar sin conocimiento de lo que está ocurriendo por debajo. Esta interacción entre las diferentes capas de aplicación y SO no es posible sin la ayuda del hardware específico, ya que es necesario, por un lado, mantener un entorno fiable y seguro y, por otro, se debe realizar una gestión eficiente de los recursos.

En el funcionamiento normal, el SO cede el control de ejecución (durante un tiempo predeterminado) a un código de usuario o de otras aplicaciones. No obstante, mientras se ejecuta este código, el sistema operativo no está en ejecución, y pueden darse eventos asíncronos (no previsibles), como una tecla que se pulsa; entonces, habrá que gestionar este evento de manera temporal hasta que el SO vuelva a tener el control. Otros eventos de los que se deberá ocupar el hardware se darán al aplicar privilegios en ciertas operaciones máquina y para todo ello es necesario determinado soporte del hardware, que se realiza por medio de registros de la CPU y de los dispositivos. Finalmente, es necesario disponer dispositivos de hardware auxiliar para realizar todas las actividades complementarias del sistema y que pueden estar integradas en placa base (por ejemplo, el controlador del teclado, USB o disco IDE) o necesitar un controlador externo, como el de vídeo (tarjeta gráfica) o controladora SCSI.

Hardware auxiliar

La atención a las interrupciones de dispositivos periféricos o la detección de accesos ilegales a la memoria son ejemplos de eventos que se deberán atender mediante hardware auxiliar.

3.1. Registros

El modo como se pude ejecutar un programa por parte de la CPU está asociado a un conjunto de información que se genera de manera automática (y que puede utilizarse) y que se almacena en diferentes registros hardware, ya sea en la propia CPU o en el hardware auxiliar (coprocesadores, dispositivos de E/S, etc.).

Muestra de ello son el contador de programa (PC[23]), el registro de instrucciones (IR) o los registros generales (Rn), que son de tipo general y que encontramos en casi todos los sistemas; pero hay otros más específicos, como los que sirven para controlar la memoria o el sistema de interrupciones, y que guardan una estrecha relación con el SO.

(23)PC es la sigla de *program counter*.

Los registros de control son una parte de la información del entorno que se guarda en el bloque de control de proceso (PCB). Aunque dependen de la arquitectura específica, se puede mencionar los más comunes:

1) Registros de control de acceso a la memoria (MAR[24]): registros que facilitan la tarea de la unidad de gestión de la memoria.

[24]MAR es la sigla de *memory addres registrer*.

2) Apuntador de pila (SP[25]): los procesos trabajan con una estructura de datos de tipo vector gestionada como una pila. El procesador tiene un registro que apunta a la pila activa (*top*), donde mediante instrucciones específicas (*push* y *pop*) se salvan los registros del procesador/datos/parámetros o se recuperan, y este registro se actualiza con el entorno general de cada proceso.

[25]SP es la sigla de *stack pointer*.

3) Registro de estado del procesador[26] o indicadores[27]: este registro agrupa una serie de bits asociados a diferentes estados del procesador, que conviene consultar en cualquier instante. Entre estos indicadores, podemos destacar los que informan el modo de ejecución actual del procesador.

[26]En inglés, *program status word*.

[27]En inglés, *flags*.

3.2. La entrada y la salida

Los **mecanismos de entrada/salida** (**E/S**[28]) de un ordenador tienen por objetivo el intercambio de información entre periféricos y la memoria o los registros del procesador. Es de vital importancia para el sistema en su conjunto la concurrencia de E/S con el procesamiento y el impacto en la memoria virtual.

[28]La expresión E/S hace referencia a los mecanismos de entrada/salida.

El dispositivo periférico está compuesto por el dispositivo y su controlador, que contiene una serie de registros incluidos en el mapa de E/S del ordenador, a los que se pueden acceder mediante instrucciones de máquina de E/S. Normalmente, se dispone un registro de datos en los que se realiza el intercambio de información y un registro de estado para gestionar el control de la información.

El disco es para el SO el periférico más importante, puesto que sirve como espacio de intercambio[29] a la memoria virtual y almacenamiento permanente. Los parámetros más importantes desde el punto de vista del SO para un disco son:

[29]En inglés, *swap*.

- **Organización de la información**: cómo esta se organiza en sectores sobre la superficie del disco.
- **Tiempo de acceso**: es la suma del tiempo de búsqueda, es decir, cuánto tiempo tarda el dispositivo en posicionar la cabeza de lectura/escritura sobre la pista que contiene el dato solicitado, más la latencia, es decir, el tiempo que tarda el dato solicitado en la pista en pasar por debajo de la cabeza lectora.
- **La velocidad de transferencia**.

Dado que los periféricos son sensiblemente más lentos que el procesador, es necesario implementar métodos y técnicas que permitan la **concurrencia entre E/S y procesamiento**, ya que si no, el procesador estará en un ciclo de espera inútil durante una operación de E/S. Por ello los procesadores incluyen mecanismos de E/S por interrupciones y E/S por acceso directo a la memoria (DMA[30]), mediante los cuales el procesador no debe estar pendiente de la E/S directamente y puede estar ejecutando otro programa (obviamente, todo ello debe ser soportado por el SO, como se verá más adelante).

(30)DMA es la sigla de la expresión inglesa *direct access to memory*.

3.3. Controladores de entrada/salida

De modo general, analizaremos como hardware de los controladores entrada/salida el dispositivo SCSI (*small computer system interface*), cuyo controlador fue estandarizado en 1984 por el American National Standards Institute (ANSI). De este dispositivo encontramos:

- **SCSI 1**: 8 bits / 5 MBps / conector genérico de 50 pines / 7 dispositivos.

- **SCSI 2**: Versión **Fast** (8bits / 10 MBps / conector 50 pines alta densidad / 7dispositivos) y versión **Wide** (16 bits / conector genérico 68 pines, alta densidad / 15 dispositivos).

- **SCSI 3**: existen diferentes tipos:
  - **SPI** (*parallel interface* o Ultra SCSI): 16 bits / 20 MBps / conector genérico 34 pines de alta densidad / 15 dispositivos.
  - **Ultra Wide**: 16 bits / 40 MBps / 15 dispositivos.
  - **Ultra 2**: 16 bits / 80 MBps / conector genérico 68 pines y alta densidad / 15 dispositivos.
  - **FireWire**: IEEE 1394.
  - **SSA** (*serial storage architecture*, de IBM).
  - **FC-AL** (*fibre channel arbitrated loop*): con fibra óptica (hasta 10 km) o coaxial (hasta 24 m) a 100 MBps.

Todos ellos utilizan CCS (*command common set*), que es un conjunto de comandos de bloques que permite la compatibilidad entre dispositivos e interfaz. SCSI[31] 1, SCSI 2 y SCSI 3.1 (SPI) conectan los dispositivos en paralelo, mientras que SCSI 3.2 (FireWire), SCSI 3.3 (SSA) y SCSI 3.4 (FC-AL) conectan los dispositivos en serie, y en todos ellos la controladora cuenta como un dispositivo (identificador 7 o 15 según el tipo).

(31)SCSI es la sigla de *small computer system interface*.

La controladora ocupa una única ranura (*slot*) de la placa madre que la conecta con los buses del sistema y es la que puede gestionar hasta 7/15 buses diferentes (identificados del 0 al 6 o del 0 a 15); y las velocidades de transferencia de estos dispositivos son desde 5 Mb/s en el estándar a 100 Mb/s en el de fibra óptica. Para evitar las pérdidas por atenuación y errores del bus, el primer dispositivo y el último deben tener unas terminaciones o conectores finales, que equilibran

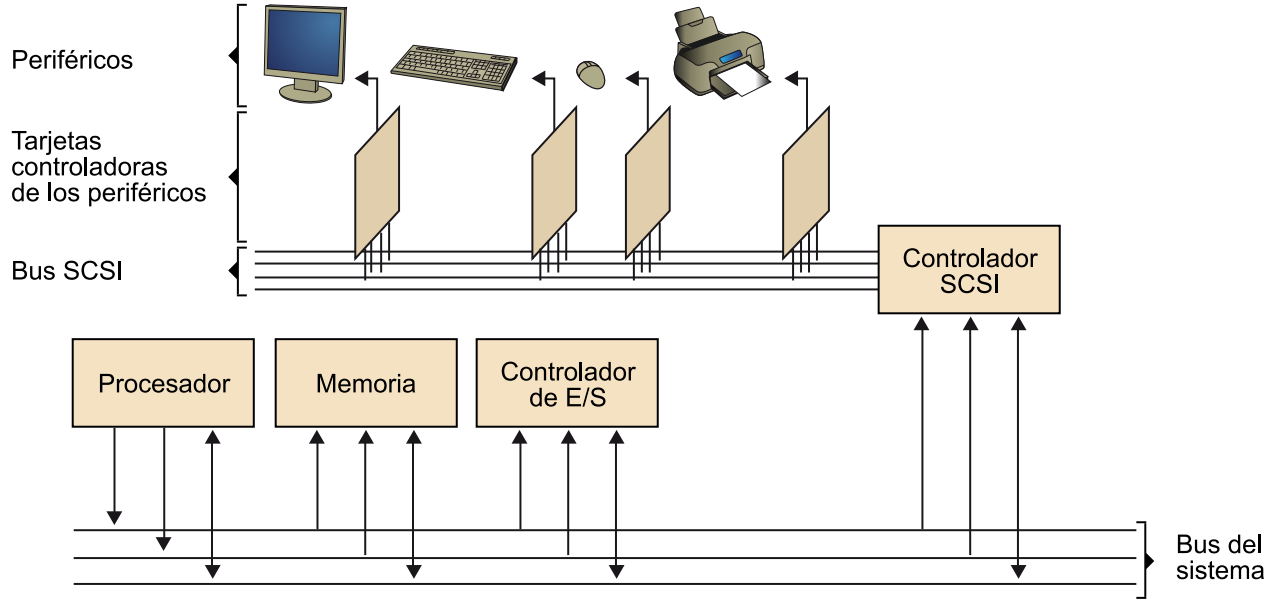
Longitud máxima del bus SCSI

Algunos ejemplos de diferentes versiones: 6 metros para SCSI1, 12 metros para Ultra2 o 10 kilómetros para fibra.

el bus desde el punto de vista de la transmisión de señales a gran velocidad. La longitud máxima del bus SCSI dependerá de la versión, pero se recomienda adecuar en función del tipo y el entorno a la longitud mínima posible de la permitida.

Los dispositivos SCSI pueden ser con interfaz y dispositivo internos o dispositivos externos y permiten toda una serie de combinaciones entre diferentes tipos de conexiones o dispositivos de E/S (discos, impresoras, escáner, etc.).

Figura 13. Gestión de periféricos con el controlador SCSI



Una interfaz muy común actualmente es la iSCSI (Internet SCSI), un estándar que permite el uso del protocolo SCSI sobre redes TCP/IP. iSCSI es un protocolo de la capa de transporte definido en las especificaciones SCSI 3 (que complementa los otros protocolos de la capa de transporte y que son SCSI *parallel interface* y FC-AL canal de fibra). La adopción del iSCSI en entornos de producción corporativos se ha acelerado en los últimos años gracias a la consolidación y reducción de precio de controladoras de Gigabit Ethernet. La fabricación de dispositivos de almacenamientos basados en iSCSI (SAN, *storage area network*) es menos costosa y está resultando una alternativa a las soluciones SAN[32] basadas en canal de fibra.

[32]SAN es la sigla de *storage area network*.

El protocolo iSCSI utiliza TCP/IP para sus transferencias de datos y, a diferencia de otros protocolos de red diseñados para almacenamiento, como por ejemplo el canal de fibra (que es la base de la mayor parte de las redes de áreas de almacenamiento), solamente requiere un simple y una sencilla interfaz *ethernet* (o cualquier otra red compatible TCP/IP) para funcionar. Esto permite una solución de almacenamiento centralizada de bajo coste sin la necesidad de realizar inversiones costosas ni sufrir las habituales incompatibilidades asociadas a las soluciones canal de fibra para SAN. Las críticas de iSCSI argumentan que este protocolo tiene peor rendimiento que el canal de fibra, ya que se ve afectado por la sobrecarga que generan las transmisiones TCP/IP (cabeceras de paquetes

básicamente), pero las pruebas que se han realizado muestran un excelente rendimiento de las soluciones iSCSI SAN cuando se utilizan enlaces Gigabit Ethernet.

En el contexto de almacenamiento, iSCSI permite a un ordenador utilizar un iniciador iSCSI (*initiator*) para conectar a un dispositivo SCSI (*target*), como puede ser un disco duro o una cabina de cintas en una red IP para acceder a ellos en cuanto a bloque. Desde el punto de vista de los *drivers* y las aplicaciones de software, los dispositivos parecen estar conectados realmente como dispositivos SCSI locales. Los dispositivos iSCSI no deben ser confundidos con los dispositivos *network-attached storage* (NAS), que incluyen software en el servidor para controlar las peticiones de acceso simultáneo desde los diferentes *hosts*.

Linux soporta iSCSI desde el 2005 en diferentes interfaces y redes y también adaptadores iSCSI *host bus* (HBA), que son tarjetas de red que incorporan un motor con la capacidad de proceso iSCSI integrada. Los HBA iSCSI son tratados por el sistema operativo como controladores SCSI convencionales y, en estos casos, el HBA no formará parte de la pila de red del sistema.

Para configurar un iniciador de **iSCSI en Debian**, es interesante seguir las indicaciones de la guía[33].

(33) http://wiki.debian.org/iSCSI/open-iscsi
http://www.open-iscsi.org/docs/README

3.4. La transferencia directa a la memoria, DMA

Dado que la CPU no está preparada para mover grandes cantidades de información entre los dispositivos que la utilizan/generan y la memoria principal, es necesario disponer de hardware adicional para hacerlo. Este dispositivo, denominado **coprocesador de DMA** (*direct memory access*), permite hacer transferencias de datos en bloques entre los periféricos (normalmente discos) y la memoria principal.

La tarea que hay que realizar es muy simple: se trata de copiar una cantidad determinada de octetos desde/hacia el periférico hacia/desde posiciones consecutivas de memoria principal. Como se ha comentado anteriormente, la intervención del procesador (CPU) presenta diferentes inconvenientes:

- La CPU no está preparada para esta tarea, y por ello no es eficiente, ya que toda la información debe pasar por sus registros. Aunque estos pueden ser de 64 bits, este proceso implica dos accesos al bus de datos, pues se deben cargar los datos del periférico al registro y posteriormente guardarlos en memoria.

- Durante la transferencia, y mientras no se haya traspasado la información a la memoria, no es posible continuar la ejecución del proceso (está suspendido a la espera de una operación de entrada/salida).

Para hacer más eficiente estas operaciones, se utiliza un coprocesador de DMA. Una transferencia DMA consiste principalmente en copiar un bloque de memoria de un dispositivo a otro. Esta operación no ocupa al procesador y como resultado este puede ser planificado para efectuar otras tareas. Las transferencias DMA son esenciales para aumentar el rendimiento de aplicaciones que requieran muchos recursos. Cabe destacar que, aunque no se necesite a la CPU para la transacción de datos, sí que se necesita el bus del sistema (tanto bus de datos como bus de direcciones), por lo que existen diferentes estrategias para regular su uso, lo que permite que no quede totalmente acaparado por el controlador DMA.

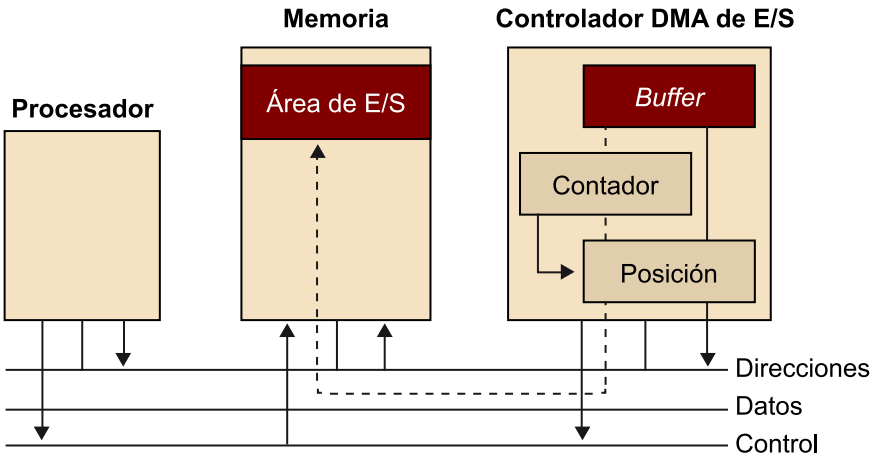
Durante las operaciones del DMA, el rendimiento del sistema puede verse afectado debido a que este dispositivo hace un uso intensivo del bus y, por lo tanto, dado que la CPU no puede leer datos de memoria, debe esperar a que finalice dicha tarea sin ejecutar ninguna instrucción. Una posible solución la aporta la memoria caché dentro de la CPU, que permite a esta seguir trabajando mientras el DMA mantiene ocupado el bus. En computadores que no disponen de memoria caché, el DMA debe realizar su tarea evitando ocupar el bus de datos mientras la CPU accede a memoria, y existen dos tipos de transferencias de datos del DMA:

a) Transferencias modo ráfaga: cuando la CPU concede el bus al DMA, este no lo libera hasta que finaliza su tarea completamente. Este tipo de transferencia se emplea en sistemas que disponen de una memoria caché en la unidad de procesamiento, pues así la CPU puede seguir trabajando utilizando la caché.

b) Transferencias modo robo de ciclo: cuando la CPU concede el bus al DMA, este lo vuelve a liberar al finalizar de transferir cada palabra, y luego debe solicitar de nuevo el permiso de uso del bus a la CPU. Esta operación se repite hasta que el DMA finaliza la tarea. Este tipo de transferencia se suele utilizar en sistemas que no disponen de memoria caché en la unidad de procesamiento, ya que de este modo, aunque la transferencia de datos tarda más en realizarse, la CPU puede seguir ejecutando instrucciones.

Se debe diferenciar entre el tiempo de espera de un ciclo de utilización del bus del sistema y una interrupción. Cuando el dispositivo DMA captura el bus, no hay un cambio de contexto por parte del procesador; simplemente ha de esperar a que el bus se libere para volver a tenerlo. Aunque la actividad del procesador disminuye en cierta medida, esto es mucho más eficiente que tener que cambiar de contexto para atender una interrupción. La figura 14 muestra el esquema del controlador de DMA y su relación entre la memoria y la CPU.

Figura 14. Transmisión de datos mediante el controlador DMA



Una operación de E/S por DMA se establece ejecutando una corta rutina de inicialización. Consiste en varias instrucciones de salida para asignar valores iniciales a:

- **AR**: dirección de memoria del area de datos de E/S (*buffer* de entrada/salida).
- **WC**: contador de palabras de datos que se van a transferir.

Una vez inicializado, el DMA procede a transferir datos entre la memoria intermedia y el dispositivo de E/S. Se realiza una transferencia cuando el dispositivo de E/S solicite una operación de DMA a través de la línea de petición del DMAC (DMACoprocessor). Después de cada transferencia, se decrementa el valor de WC y se incrementa el de AR. La operación termina cuando WC = 0, entonces el DMAC (o el periférico) indica la conclusión de la operación enviando al procesador una petición de interrupción.

La secuencia de eventos detallada será:

1) El procesador inicializa el DMAC programando AR y WC.

2) El dispositivo de E/S realiza una petición de DMA al DMAC.

3) El DMAC le responde con una señal de aceptación.

4) El DMAC activa la línea de petición de DMA al procesador.

5) Al final del ciclo del bus en curso, el procesador pone las líneas del bus del sistema en alta impedancia y activa la cesión de DMA.

6) El DMAC asume el control del bus.

7) El dispositivo de E/S transmite una nueva palabra de datos al registro intermedio de datos del DMAC.

8) El DMAC ejecuta un ciclo de escritura en memoria para transferir el contenido del registro intermedio a la posición M[AR].

9) El DMAC decrementa WC e incrementa AR.

10) El DMAC libera el bus y desactiva la línea de petición de DMA.

11) El DMAC compara WC con 0:

- Si *WC* > 0, se repite desde el paso 2.
- Si *WC* = 0, el DMAC se detiene y envía una petición de interrupción al procesador.

Las tendencias actuales incorporan procesadores de DMA de alto rendimiento en los chips de apoyo a los procesadores (llamados *chipset*) con funciones avanzadas, como el caso de los procesadores Intel Xeon, que incluyen la nueva tecnología de DMA denominada I/O Acceleration Technology (I/OAT), y que tienen estrategias específicas para mantener la coherencia de la memoria caché[34] del procesador y los dispositivos de E/S, así como una utilización eficiente de memoria caché circulares y canales específicos de DMA.

[34] En inglés, *buffer*.

3.5. El reloj del sistema

El reloj del sistema es un caso especial de dispositivo de entrada, ya que se gestiona como el resto de los dispositivos pero se puede considerar que es un dispositivo totalmente hardware. Este dispositivo está basado en un circuito electrónico que genera impulsos de alta frecuencia (y que tiene como referencia un cristal de cuarzo para garantizar su estabilidad) y la frecuencia de estos impulsos se reduce hasta obtener las frecuencias necesarias para gestionar los registros de tiempo.

El objetivo del **reloj** es mantener una referencia temporal en el sistema y trabaja de manera autónoma e independiente del funcionamiento de la CPU en su función principal de ejecutar instrucciones.

Por ello se pueden diferenciar dos tareas principales a las que está dedicado su funcionamiento:

- Mantener un reloj interno convencional, en el sentido de que indique una referencia temporal de la cual se pueda derivar el día y la hora de modo constante (aunque el sistema se detenga temporalmente).

- Generar una serie de interrupciones de hardware que activan una rutina del núcleo de manera síncrona (cada cierta cantidad de milisegundos) y que será la base por la cual el SO cada cierto tiempo podrá obtener el control de la CPU.

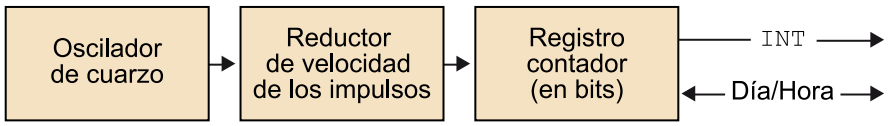
La función vital del reloj es que permite hacer un sistema multiprogramado y de tiempo compartido, ya que con cada interrupción el SO comprueba si el proceso actual en ejecución ha agotado su cuota de procesador o no. De este modo, se van contabilizando los tiempos antes de hacer un cambio de contexto y de asignar el procesador a otro proceso, contabilizando en las estructuras de datos del proceso (PCB) el tiempo gastado en el proceso que deja la CPU.

No se debe confundir el reloj del sistema con el reloj del procesador (mucho más rápido), que es el que marca la frecuencia de ejecución (ciclos) de las instrucciones máquina. Dado que este reloj está vinculado a la potencia de procesador (instrucciones por segundo), existen técnicas para manipular de diferentes formas el procesador (*overclocking*), pero se debe ir con cuidado, ya que pueden inutilizar totalmente el procesador (un aumento de la frecuencia de ejecución de instrucciones genera más calor, que el procesador no puede disipar y acaba quemándose). El reloj del sistema cuenta con una batería para mantener actualizada la fecha/hora del sistema, mientras que el reloj del procesador se detiene cuando se para el ordenador.

Overclocking

Práctica no recomendada para aumentar la frecuencia del reloj del procesador que puede provocar su destrucción (extraído de Wikipedia).

Figura 15. Esquema de funcionamiento del reloj del sistema



Las llamadas más interesantes en Linux para gestionar el tiempo son: *time()*, *stime()*, *times()*, *gettimeofday()*, *settimeofday()*, *settimer()*, *gettimer()*, *adjtimer()*, *sleep()*, *usleep()*, *nanosleep()*, que se encuentran todas en kernel/time.c o en kernel/sched.c. Como comandos interesantes y vinculados al reloj encontramos ***date***, ***sleep***, ***rtcwake***, ***hwclock*** y ***time***.

3.6. Protección

Una de las funciones más importantes del SO es la protección de unos usuarios contra otros (no por malicia ni por descuido). Sin embargo, deben existir mecanismos hardware que comprueben esta protección durante la ejecución. Por ello, el sistema deberá tener:

1) Mecanismos de protección del procesador: basado en los niveles de ejecución y controlando el acceso a los dispositivos periféricos de manera controlada, impidiendo que un usuario pueda acceder a ellos. Tal y como se ha visto anteriormente, los diseños de los SO se esfuerzan en mantener los periféricos bajo control y a su vez hacer que se pueda acceder a ellos mediante abstracciones (por ejemplo, HAL), o en la capa de usuario implementando mecanismos que les permitan a la vez ser eficientes/simples pero en forma controlada por el SO.

2) Mecanismos de protección de memoria: son los necesarios para evitar que un programa acceda a posiciones no permitidas del mapa de memoria. Existen varias formas, como por registros límites, zonas, capas o por medio de –como hemos visto– la memoria virtual a través de la zona que se le revela por medio de los mecanismos de traslación y la unidad de administración de memoria.

3.7. Modos de ejecución del procesador

Un procesador tiene diferentes modos de ejecución (usuario, supervisor, núcleo, etc.) pero para simplificarlo se considerará dos modos básicos: el **modo usuario** y el **modo sistema/supervisor**. El modo de ejecución determina los privilegios que tiene un proceso para acceder a la memoria o para ejecutar determinadas instrucciones privilegiadas, como la instrucción *halt*, que sirve para detener el procesador.

Estos modos de ejecución están sustentados por los bits de registro de indicadores, del que ya se ha hablado, para gestionar de modo eficiente y sin problemas de control los diferentes cambios de modo. Las rutinas que se ejecutan en un determinado modo pueden ejecutarse en modo protegido siendo inaccesible su código/datos a otras rutinas de un rango inferior, pero cada rutina de un rango superior sí puede acceder a código o datos de un rango inferior.

Una consecuencia fundamental de esto es que cada proceso puede tener tantas pilas de ejecución[35] como posibles modos de ejecución, y se puede acceder a cada pila solo desde su propio modo de ejecución o de los modos con más privilegio que el actual. En este sentido, podrá existir una pila exclusiva para el SO, que se utilizará de manera exclusiva para los servicios de interrupción asíncronos y que se deberá manejar desde el modo de máximos privilegios (modos sistema/supervisor).

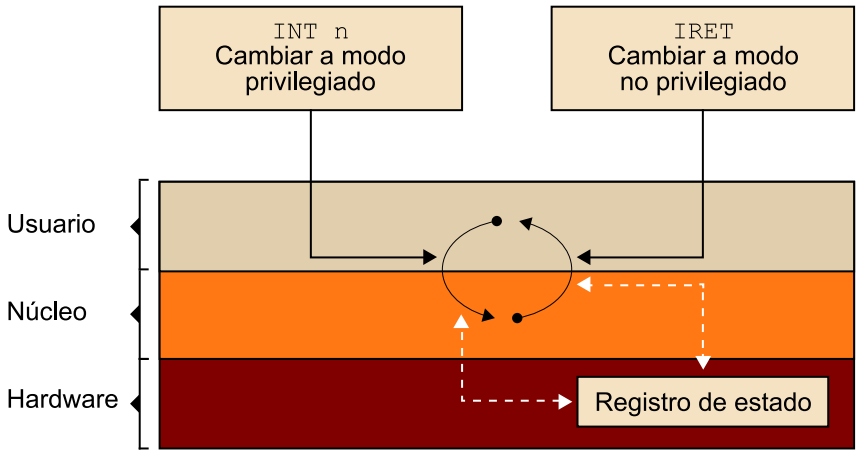
(35) En inglés, *stacks*.

La información del modo actual de ejecución se guarda en el registro de estado del procesador en un conjunto de bits equivalente a los diferentes estados que tenga el procesador (2 bits significa 4 estados posibles). Los bits de estados son tratados por instrucciones privilegiadas y normalmente el procesador ejecuta el código en el nivel de privilegio más bajo (modo usuario). Así, para ejecutar determinadas funciones, por ejemplo la entrada/salida, el programa de usuario

llama a un servicio de sistema; si la rutina se debe ejecutar en un modo que tenga más privilegios, ya sea porque tiene que acceder a áreas privilegiadas de memoria o ejecutar instrucciones de E/S, se cambiará el nivel de ejecución y el control pasará al sistema operativo, en el que las rutinas se podrán ejecutar con el nivel de privilegios adecuados.

Estos cambios de modo se efectúan con la ayuda de una interrupción software (la instrucción *INT n*) y, de modo similar, cuando se termina la rutina de servicio del núcleo se restaura el registro del estado original por orden de la instrucción *IRET* (instrucción de retorno de una interrupción), con lo que se recupera el modo de ejecución anterior.

Figura 16. Cambio de modo de ejecución



3.8. Mecanismos de acceso al núcleo del sistema operativo

1) Interrupciones

Los mecanismos de acceso al núcleo del sistema operativo se basan esencialmente en la gestión de interrupciones durante la ejecución de un proceso. Para ello, la CPU debe tener capacidad de capturar y procesar estas interrupciones haciendo la secuencia siguiente:

- Detectar la interrupción durante la ejecución de instrucciones.
- Almacenar el estado y los registros y redirigir la petición hacia la rutina correspondiente.
- Al finalizar la rutina de atención de la instrucción, recuperar el estado continuar la ejecución justo en el punto en el que se ha dejado.

Toda esta secuencia de acciones necesita hardware específico, ya sea para detectar y seleccionar el dispositivo externo que ha interrumpido, ya sea para procesar las interrupciones de software generadas por el mismo sistema pero a través de la ejecución de una instrucción específica del procesador. El siste-

ma de soporte a las interrupciones varía en los detalles, pero muchas de sus características son comunes a todos los ordenadores y sobre todo en una descripción genérica en la parte que afecta al SO.

La CPU generalmente contiene una o más líneas de interrupción que activan el indicador de interrupciones que se encuentra en el registro de control del procesador. Estas líneas también poseen una máscara de interrupciones que permiten que el procesador omita las interrupciones, así como también líneas de interrupciones no enmascarables para evitar que el procesador las pueda ignorar.

Procesador Intel Pentium

En un procesador Intel Pentium, el registro de indicadores tiene 32 bits de los cuales se utilizan 13 bits (son compatibles con versiones anteriores del procesador) y donde las funciones que podemos encontrar son de condiciones sobre las operaciones (*sign, zero, carry, auxiliary carry, overflow, parity*), relativas a operaciones de la CPU (*interrupts, single-step, strings*) y relativas a privilegios (*I/O privilege, nested task*). En este caso, *I/Oprivilege* utiliza 2 bits en modo protegido para determinar qué instrucciones pueden ser ejecutadas y *nested task* es utilizada para mostrar un enlace entre dos tareas.

Cuando se produce la interrupción (no enmascarable) se desarrollan un conjunto de operaciones que se describen a continuación:

a) Para detectar una interrupción, la CPU, al finalizar el ciclo de ejecución de una instrucción, consulta el estado de interrupciones, que avisa de que hay una interrupción pendiente. Si hay una interrupción, la CPU consulta la máscara de interrupciones para saber si la debe atender o no. Si está habilitada (máscara inactiva), la interrupción es atendida y se continúa con la secuencia de atención; si está la máscara activa, la interrupción se ignora. Las interrupciones están priorizadas y son apropiativas, es decir, si durante el ciclo de interrupción llega otra más prioritaria, se suspende la actual para atender a la de más prioridad (esto permite atender por ejemplo una petición de *swap* de memoria antes que leer una tecla desde el *buffer* del teclado). Antes de hacer nada, el sistema guarda el estado en la pila del sistema para saber después dónde continuar la ejecución.

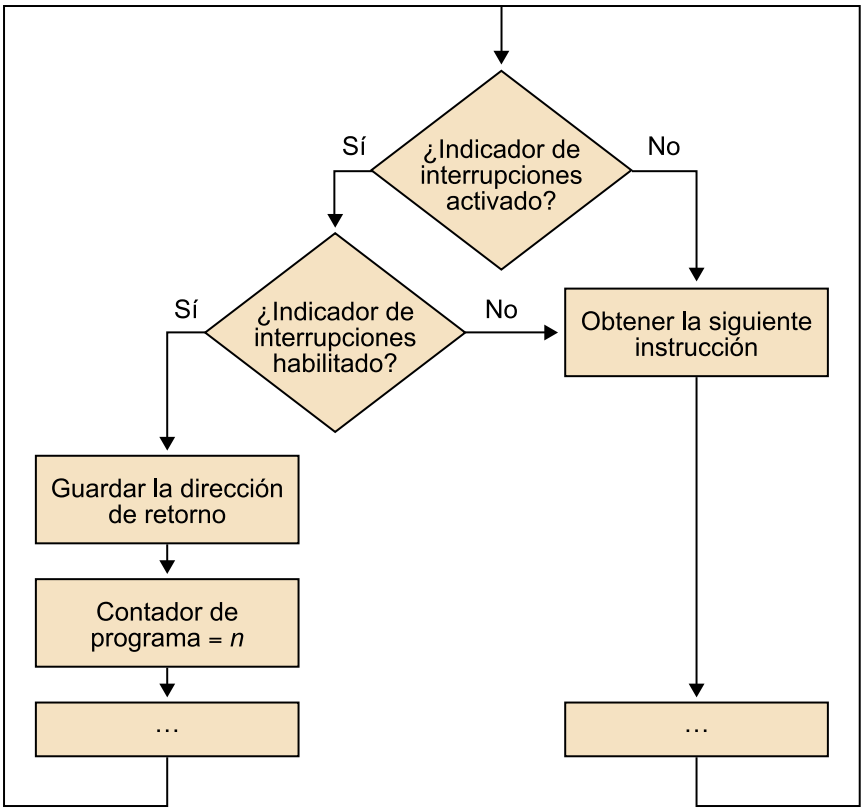
b) A continuación, la CPU lee si es una interrupción externa, el número de dispositivo solicitante (a través del bus de datos) o decodifica el tipo y los registros de la CPU si es una interrupción software para determinar cuál es el vector de interrupciones al que debe acceder. Con este valor entra en la tabla de interrupciones e, indexando desde el inicio de esta, accede al vector de interrupciones, donde estará la rutina de atención (los saltos no programados y las excepciones utilizan mecanismos similares a los que hemos descrito para las interrupciones de la ejecución).

Linux

En Linux, en el archivo /proc/interrupts, se puede consultar la asignación de vectores de interrupción disponibles en el sistema.

c) A continuación, se ejecuta la rutina de atención y después se ejecuta una instrucción (*IRET*) que recarga de la pila del sistema el contador del programa y los de los registros internos, y se continúa con la siguiente instrucción posterior a cuando se produjo la interrupción.

Figura 17. Esquema de la gestión de interrupciones



2) Saltos no programados

Como se ha comentado anteriormente, el procedimiento que puede utilizar el usuario para hacer una petición al SO es el salto no programado[36], que se activa cuando el procesador ejecuta una instrucción de lenguaje máquina llamada *trap*. En la ejecución de esta instrucción se pueden distinguir tres partes:

[36] En inglés, *trap*.

- **Cambio de modo de ejecución**: el procesador pasa de ejecutar código en modo usuario a ejecutarlo en modo sistema para permitir el acceso a datos código del SO y la posibilidad de ejecutar instrucciones privilegiadas.

- **Ejecución de una rutina de atención**: se transfiere el control a una rutina que procesará el código deseado, la cual funcionará como una rutina de interrupción guardando el contexto, ejecutando el servicio solicitado y al finalizar restaurará el contexto del proceso que ejecutó la llamada.

- **Cambio de modo de ejecución**: el procesador vuelve a modo usuario para continuar la ejecución del proceso.

Aspectos importantes de la llamada a las instrucciones *trap* son cómo se determina la dirección de la rutina, cómo se realiza el paso de parámetros, y qué pila se utilizará, y cómo se producirá el retorno al proceso que ejecuta la excepción *trap*, que serán tratados a continuación:

a) Determinación de la dirección de la rutina de atención: una posibilidad es delegar la responsabilidad al programador del código del proceso, quien debería indicar las direcciones de las rutinas de atención en las instrucciones *trap*. Es importante señalar que esta solución es similar a la llamada a un procedimiento mediante la instrucción en ensamblador *CALL*, pero con la diferencia de que se produce un cambio de modo de ejecución en el procesador. Esta solución presenta algunos problemas, como el conocimiento por parte del programador de la arquitectura/SO a bajo nivel, posibilidades de error al introducir datos incorrectos y sus correspondientes cadenas de excepciones por accesos incorrectos, poca flexibilidad en cambios del sistema operativo, ya que el código que funciona en una versión podría no funcionar en otra y se debería reescribir/compilar el código nuevamente, etc.

b) Determinación en tiempo de compilación: otra solución es que esta tarea la haga el compilador, donde el programador indicaría el servicio al que quiere acceder y el compilador resolvería la dirección correspondiente mediante una tabla de traslación entre la indicación del programador y la dirección correspondiente con las ventajas correspondientes (código ejecutable estable, no errores), pero no otros (cambio de filosofía del compilador en relación con traslaciones, cambios de ubicación implica recompilación, etc.).

c) Determinación en tiempo de enlazado (*link*): otra posible solución al problema de la determinación de la dirección pasaría por tener unas librerías responsables de hacer los saltos no programados. El programador llamaría a la rutina correspondiente al salto no programado que quiere ejecutar y el código de la rutina de la librería haría la entrada al sistema. Como ventajas, podemos enunciar que los cambios de las rutinas no implican recompilar el código: el compilador continúa usando la misma política de llamada y resolución de símbolos, pero tiene como inconvenientes que los cambios en las rutinas significarían cambiar la librería y volver a enlazar los programas (aunque se podría resolver si son librerías dinámicas).

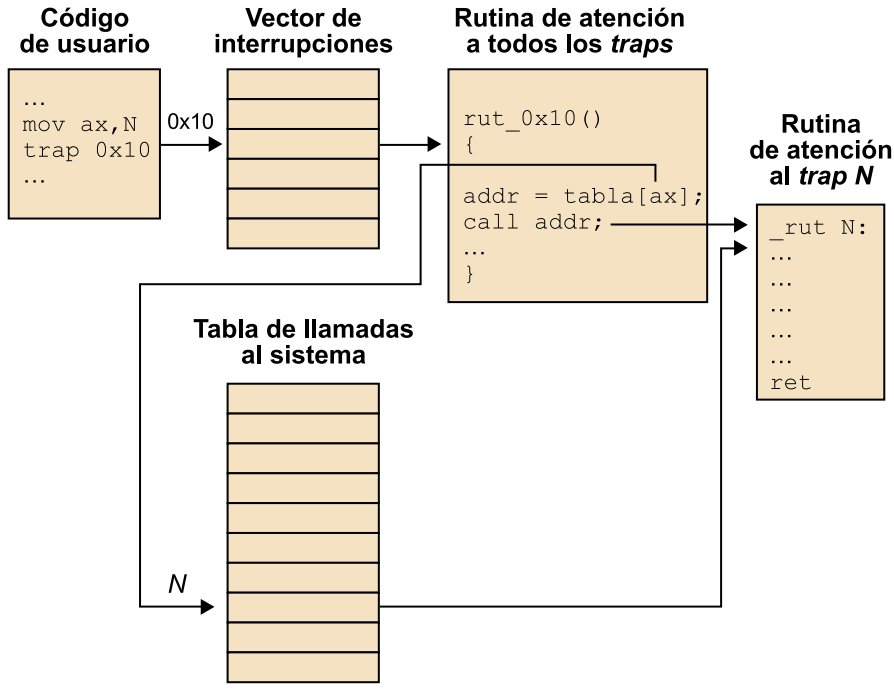
d) Determinación en tiempo de ejecución (por hardware): en esta solución, el programador no puede acceder a la dirección de la rutina (es decir, ni conoce dónde está) y para acceder identifica las llamadas al sistema mediante un índice, y parametriza la instrucción *trap* con este índice (por ejemplo, *trap 10*). Esto obliga a que en el espacio de datos del SO debe existir una tabla que relacione cada uno de estos índices con la dirección de la memoria donde se encuentra la rutina de atención (equivalente al vector de interrupciones). En

resumen, la ejecución de la instrucción *trap N* se encarga de consultar esta tabla en la posición *N*, de obtener la dirección contenida en la tabla y de pasar el control a la instrucción inicial de la rutina.

Si cambia la configuración de la máquina o en caso de que se incorpore una versión nueva del SO, solo será necesario cambiar el contenido de la tabla y reajustar las direcciones. Con ello, los programas no se deberán compilar ni enlazar, ya que los identificadores continúan siendo los mismos y el procedimiento es análogo al tratamiento de interrupciones software. El problema estará en el tamaño de la tabla y las posibilidades del índice si el SO tiene una gran cantidad de llamadas al sistema.

e) Determinación en tiempo de ejecución (combinando hardware y software): el problema anterior por restricciones del hardware se puede solucionar haciendo que la relación entre el identificador de la llamada al sistema y la dirección de la rutina se realice parcialmente por software, utilizando una única entrada del vector de interrupciones para todos los saltos no programados, y guardando en un registro del procesador o en la pila del proceso el identificador de la llamada al sistema que se quiere invocar. La figura 18 muestra un esquema y la secuencia seguida por este método:

Figura 18. Esquema del funcionamiento por determinación en tiempo de ejecución (combinando hardware y software)



En este caso, el acceso al vector de interrupciones se efectúa por hardware (ejecución de la instrucción *trap N*), mientras que el acceso a la tabla de llamadas al sistema se hace por software y se utiliza una única entrada del vector de interrupciones que da servicio a todos los saltos no programados del SO. Esta es la solución que utilizan la mayoría de los SO.

Un aspecto importante en los saltos no programados es el modo como se efectúa el paso de parámetros, que se puede hacer igual que en la rutinas convencionales: utilizando los registros del procesador para pasar los parámetros de la llamada al sistema (solución válida si el número es menor que el número de registros del procesador), o utilizar la pila del proceso, lo cual presenta el problema de que entre el momento en el que se hace la llamada y la ejecución de la rutina de atención al salto no programado se ejecutan varias rutinas intermedias que escribirán información en la pila y, por tanto, los parámetros se encontrarán a profundidades diferentes en función de la rutinas intermedias y lo que hayan puesto en la pila. Otro problema que puede ocasionar este método es que como hay un cambio de modo de ejecución en el procesador (de modo usuario a modo sistema), seguramente cambiará la pila (dependerá de la arquitectura del SO) y puede ocurrir que los parámetros estén en otra pila que puede ser complicado ubicar.

Por ello, el constructor del sistema operativo debe prestar atención especial a la arquitectura del procesador y a cómo se trabaja con la pila de ejecución, es decir, si es diferente la pila en modo usuario a la pila de modo sistema. Un proceso que está ejecutando el código de atención a una llamada al sistema necesita una pila para guardar el valor inicial de los registros, guardar las variables locales de la rutina de atención, registrar la secuencia de bloques de activación de los procedimientos llamados, etc., pero ¿en qué pila guardará estos datos, los parámetros de la llamada, o dónde devolverá los datos resultantes de la ejecución de la llamada? La respuesta depende de varios aspectos, como, por ejemplo, el sistema de gestión de la memoria (hay arquitecturas en las que un cambio de modo de ejecución provoca un cambio total en el espacio de direcciones accesibles) y las soluciones pasan por o bien tener dos pilas (una por cada modo) y adecuar los accesos para utilizar la correcta o bien dividir el espacio de memoria del proceso en dos partes (una usuario y otra sistema) pero compartiendo la misma pila.

En cuanto al retorno de valores de las llamadas al sistema, normalmente todas las llamadas al sistema devuelven un código que indica si el servicio solicitado es satisfactorio o vuelve con error y esto generalmente será a través de uno o dos registros del procesador. En cuanto a datos relacionados, se pueden retornar en los registros del procesador y si no es posible (por ejemplo, el resultado de la lectura de un fichero de 265 KB), la forma dependerá de la implementación del SO, para lo cual generalmente se utilizan dos posibilidades:

- Retornar la posición de memoria (apuntador a una posición de memoria caché) en la que ha dejado el resultado de la lectura.
- El programador pasa un tipo de datos que se traduce en un apuntador donde desea el resultado (solución empleada en Unix).

Secuencia de cómo funciona un salto no programado en Linux

Linux vincula en tiempo de ejecución los identificadores de llamadas al sistema que quiere hacer el usuario con las direcciones de la memoria donde se encuentran estas rutinas. Como ejemplo, seleccionaremos la llamada *close(fd)* que recibe como parámetro el *file*

descriptor del dispositivo/archivo que se quiere cerrar (se han numerado las líneas para comentarlas posteriormente):

```
1: int close(int fd){
2: int _R_Code;
3: inline(             # código assembler
4: mov _NR_close,%eax  # identificador a eax
5: mov fd,%ebx         # primer parámetro en ebx
6: int $0x80           # instrucción del trap
7: mov %eax,_R_Code    # Resultado en _R_Code
8: );
9: if ( _R_Code >= 0) return _R_Code; # Si no error exit
10:                    # Si error: guardar código del error en errno
11: errno = -_R_Code;
12: return -1;         # retornar aviso error
13: };
```

- **Línea 4 y 5**: se guarda en el registro *eax* el identificador de la llamada al sistema que se desea hacer (*_NR_CLOSE*), y el parámetro (*fd*) de la llamada en el registro *ebx*; si la llamada tuviera más parámetros, los guardaría en otros registros.

- **Línea 6**: interrupción 0x80, que es la destinada a atender todos los saltos no programados.

- **Línea 7 y subsiguientes**: el valor de retorno en *eax* se guarda en la variable *_R_Code* y si es mayor o igual a cero, indica ejecución sin errores, y si es negativo, se guarda en *errno* el código de error y la función retorna –1.

Un ejemplo más complejo es la llamada *socket syscall*, que tiene el siguiente prototipo de llamada: *socket(AF_INET,SOCK_STREAM,IPPROTO_TCP)*; donde veremos que el número de la llamada (*SYS_socketcall*) irá en el registro *eax* y la funciones de esta son identificados vía números de subfunción localizados en /usr/include/linux/net.h y son almacenados en *ebx*. El puntero a los argumentos es almacenado en *ecx* y la llamada es ejecutada con int $0x80 (código de *socket.s*):

```
.include "defines.h"

.globl     _start
_start:
           pushl %ebp                       #Salva registros
           movl %esp,%ebp
           sub $12,%esp                     #Adecua puntero

//         socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
           movl $AF_INET,(%esp)             #inserta argumentos
           movl $SOCK_STREAM,4(%esp)
           movl $IPPROTO_TCP,8(%esp)

           movl $SYS_socketcall,%eax        #función
           movl $SYS_socketcall_socket,%ebx #subfunción
           movl %esp,%ecx
           int $0x80                        #trap

           movl $SYS_exit,%eax              #función exit
           xorl %ebx,%ebx
           int $0x80

           movl %ebp,%esp                   #recupera contexto
           popl %ebp                        #retorna
           ret
```

Es interesante el documento "Implementing a System Call on Linux 2.6" <http://tldp.org/HOWTO/html_single/Implement-Sys-Call-Linux-2.6-i386/>, en el que se describen la secuencia de pasos y las estructuras de datos/archivos para que un programador pueda agregar llamadas al sistema al núcleo Linux (a partir de la versión 2.6) y un ejemplo de cómo crear una llamada a sistemas, compilarla y depurarla.

3.9. Jerarquía de la memoria

Dado que la memoria de alta velocidad tiene un precio elevado y tamaño reducido, la memoria de un ordenador se organiza en una jerarquía que posee una organización piramidal en niveles. Su objetivo es conseguir el rendimiento de una memoria de gran velocidad al coste de una memoria de baja velocidad, basándose en el principio de cercanía de referencias. La velocidad óptima para la memoria es la velocidad a la que el procesador puede trabajar, de modo que no haya tiempos de espera entre ejecución de instrucciones que necesitan la memoria para leerlas o traer/guardar operandos. El coste de la memoria no debe ser excesivo para que sea factible construir un equipo accesible teniendo en cuenta las siguientes premisas: a menor tiempo de acceso, mayor coste; a mayor capacidad, mayor coste, y a mayor capacidad, menor velocidad. Entonces se debe contar con capacidad suficiente de memoria, con una velocidad que sirva para satisfacer la demanda de rendimiento y con un coste que no sea excesivo. Por el principio denominado **cercanía de referencias** o **localidad**, es factible utilizar una mezcla de los distintos tipos y lograr un rendimiento cercano al de la memoria más rápida.

Los niveles que componen la jerarquía de memoria habitualmente son:

- Nivel 0 (menor cantidad, mayor velocidad): registros.
- Nivel 1: memoria caché.
- Nivel 2: memoria principal.
- Nivel 3 (mayor cantidad, menor velocidad): disco duro (con el mecanismo de memoria virtual).

Hay algunos autores ahora que consideran un quinto nivel, el nivel 4: redes.

La gestión de la jerarquía de memoria es compleja, puesto que se deben tener en cuenta las copias de la información que están en cada nivel y realizar las transferencias de información a niveles más rápidos, como la actualización a niveles permanentes. Una parte muy importante corre en el nivel del SO, aunque para hacerla correctamente requiere la ayuda del hardware.

3.9.1. La unidad de gestión de la memoria

Dado que todos los procesadores actuales soportan memoria paginada o segmentada como método de acceso a la memoria (y que serán la base para luego implementar la memoria virtual), es necesario un conjunto de hardware adicional que permita transformar las direcciones que salen del procesador, o direcciones lógicas, en direcciones de la memoria física, o direcciones físicas.

La traducción de memoria en un sistema segmentado es una sencilla operación de suma (verificando anteriormente que el desplazamiento es menor que el tamaño del segmento) con la dirección del procesador más la dirección base del segmento (que se encuentra en una tabla específica junto con el tamaño

del segmento). En el caso de la memoria paginada, se traduce el número de página lógico al número de página física (que existe en la tabla de páginas), se verifica el desplazamiento para verificar que es menor que el tamaño de página y la nueva dirección se conforma con la tabla de páginas física (después de la traslación) más el desplazamiento.

Cualquiera de los dos métodos permite que la memoria asignada a un proceso no tenga que ser contigua y esto se traduce en un mejor uso de la memoria principal. Además, ambos métodos se complementan con mecanismos de protección, es decir, que un código no puede acceder a zonas que no pertenecen a su página o segmento para evitar accesos fuera de la zona permitida que podría corresponder al área del núcleo del SO o al área de otros procesos.

Figura 19. Gestión de espacios de memoria disyuntos

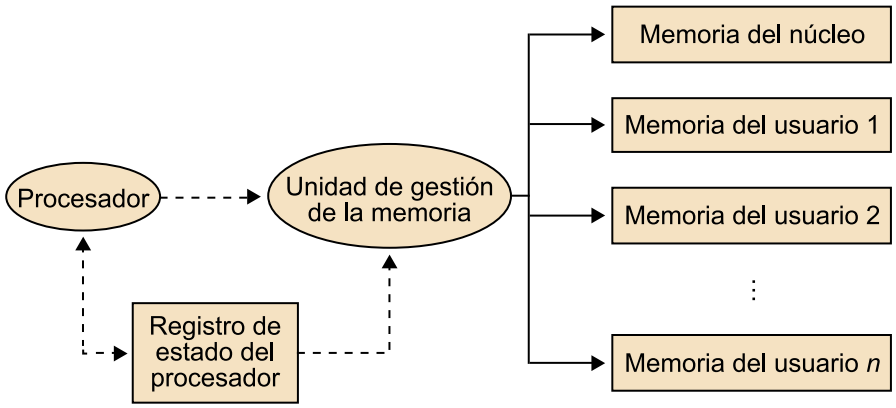
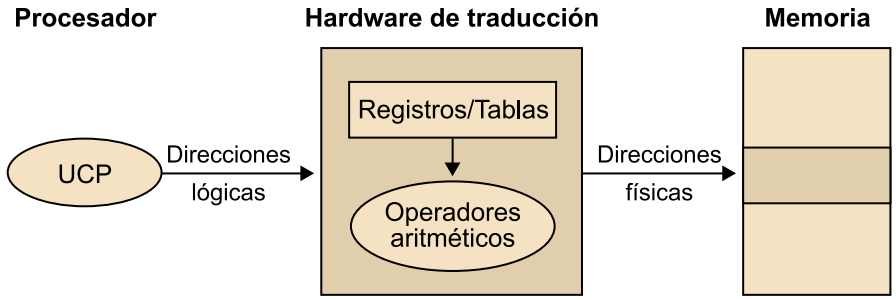


Figura 20. Proceso de traducción de direcciones



El núcleo sí que puede acceder tanto al área de sistema como al área del usuario para manipular los datos; esta posibilidad se gestiona a través de los cambios de modo de ejecución indicado por el registro de estado, que permite que el procesador acceda a una zona de memoria o a otra.

3.10. Instrucciones privilegiadas

Las instrucciones privilegiadas son las utilizadas por el SO y si se ejecutan desde fuera del núcleo se provoca un error que es capturado por una excepción que provoca que el sistema cambie a modo supervisor. Los procedimientos que necesitan instrucciones privilegiadas son los siguientes:

1) Cambio de modo de ejecución: las rutinas del núcleo del sistema se deben ejecutar en un modo privilegiado y no es el usuario quien ejecuta específicamente una instrucción de cambio de modo de ejecución, sino que este hace una llamada a una rutina del núcleo para pedir un servicio del sistema, y es la rutina la que hace la llamada al cambio de modo antes de orientar el flujo de ejecución hacia el procedimiento que ejecuta la rutina de servicio. Es importante señalar que las rutinas de servicio del sistema se ejecutan dentro del contexto de usuario, como si fuesen llamadas a subrutinas normales, pero con privilegios para direccionar a todo el espacio de la memoria. La rutina comprueba los parámetros correspondientes antes de ejecutar el código, y si no son correctos, devuelve el control al usuario y no hace nada aunque tenga los privilegios.

Cuando termina la rutina de núcleo, la instrucción *IRET* restaura el modo original al restaurar el registro de estado del procesador original. La instrucción *IRET* siempre da lugar a una disminución de los privilegios. No se puede utilizar para aumentarlos.

2) Habilitación/inhabilitación de interrupciones: la manipulación de los indicadores del registro de estado del procesador se debe realizar con privilegios, por lo que el código del SO que permite cambiarlos se ejecuta en modo privilegiado.

3) Gestión de la memoria: los aspectos de protección en relación con la gestión de la memoria se realizan con los registros de control de las unidades de gestión de la memoria. Si consideramos una gestión de la memoria basada en segmentos, las instrucciones de manipulación de los valores de estos registros se deben ejecutar en el modo núcleo/supervisor para así proteger que cualquier usuario pueda modificar los valores de los registros de base límite para acceder al área de la memoria no permitida. El modo de ejecución también está ligado a la gestión de la memoria en forma directa, ya que la parte del registro de estado que indica el modo actual de ejecución se utiliza también para determinar si los accesos a la memoria son correctos en cada caso.

Para llevar a cabo gestiones de la memoria más complejas, como la segmentación, paginación, modelos mixtos, memoria virtual, etc., la manipulación de las tablas de segmentos, páginas y marcos se realiza utilizando instrucciones privilegiadas. Como es habitual, solo los modos núcleo/sistema tienen privilegios para manipular estas estructuras de datos por una cuestión de protección para evitar que cualquier código pueda acceder a zonas de memoria que no son suyas o solicitar/liberar recursos de los que no dispone.

4) Modificación del reloj del sistema: los comandos para manipular la fecha/hora se ejecutan solo en modo privilegiado por las connotaciones que tiene para el resto de los usuarios, aunque las instrucciones de consulta sí se pueden ejecutar en modo usuario.

3.11. Clases de dispositivos en Unix/Linux

En un sistema Unix, varios procesos concurrentes realizan diferentes tareas, solicitando servicios a través de las llamadas al núcleo que gestiona y administra todas estas solicitudes. Las llamadas al sistema están definidas libc y proporciona el acceso a las llamadas al sistema. Se pueden invocar las llamadas directamente o también a través de la llamada *syscall()*. Cada llamada al sistema tiene un número definido en <syscall.h> o <unistd.h>. Internamente, la llamada al sistema es invocada por la interrupción software 0x80 para transferir el control al núcleo. La tabla de sistema de llamada se define en el archivo de los fuentes del núcleo Linux, por ejemplo, arch/i386/kernel/entry.S.

Aunque la distinción entre las tareas del núcleo no siempre se puede diferenciar, podemos determinar unas categorías básicas:

1) Gestión de procesos: el núcleo es el encargado de crear los procesos a través del *fork()* y destruirlos con *kill()*, y manejar su conexión con el mundo exterior (entrada y salida) por medio de las diferentes llamadas *ioctl()* o propias de E/S (*read()*, *write()*, etc.). La comunicación entre los diferentes procesos, a través de *señales signal()*, *tuberías pipe()*, o primitivas de comunicación entre procesos (send(), recv()), son de vital importancia para la funcionalidad general del sistema y también son manejadas por el núcleo. Dentro de esta gestión se encuentra el planificador, que es la parte del núcleo que controla cómo los procesos comparten la CPU.

2) Gestión de memoria: la memoria principal es un recurso importante para este tipo de arquitectura y la política para gestionarla es crítica para el sistema. En Unix se gestiona un espacio de dirección virtual para todos los procesos y se cuenta con un conjunto de llamadas, como por ejemplo la *malloc()*, *free()*, *msync()* o *mlock*, entre otras.

3) Sistemas de ficheros: Unix está estrechamente ligado al concepto de sistema de archivos y casi todo en Unix puede ser tratado como un archivo. El núcleo construye un sistema de archivos estructurado en la parte superior de hardware no estructurado y utiliza la abstracción de archivo resultante en todo el sistema. Además, Linux soporta múltiples tipos de sistema de ficheros, es decir, diferentes maneras de organizar los datos sobre el medio físico (ext2, 3, 4, NFS, etc.). Las llamadas más interesantes son *open()*, *close()*, *dup()*, *create()*, *link()*, *unlink()*, *mknod()*, *chmod()*, *stat()*, *lseek()*, *mount()*, *access()*, etc.

4) Control de dispositivos: Muchas de las operaciones del sistema se mapean sobre un dispositivo físico. Con la excepción del procesador, la memoria, y otros pocos dispositivos, todas las operaciones de control del dispositivo se realizan mediante código específico para el dispositivo del que se trate y se denomina controlador de dispositivo[37]. El núcleo debe tener incrustado un controlador de dispositivo para todos los periféricos presentes en un sistema,

Lectura recomendada

A. Rubini; J. Corbet (2001). *Linux Device Drivers* <http://www.xml.com/ldd/chapter/book/index.html>.

desde el disco duro al teclado o el dispositivo gráfico. La escritura de *drivers* en Linux no es una tarea difícil pero implica un conocimiento exhaustivo del sistema.

(37)En inglés, *device driver*.

5) Gestión de la red: la interconexión debe ser gestionada por el SO porque la mayoría de las operaciones de red no son específicas de un proceso, los paquetes entrantes son eventos asincrónicos. Los paquetes deben ser recogidos, identificados y enviados antes de que un proceso pueda hacerse cargo de ellos. El sistema deberá encargarse de la entrega de paquetes de datos por medio de interfaces de red y controlar la ejecución de los programas de acuerdo con su actividad en la red, así como realizar las traslaciones (*routing*) y la resolución de direcciones.

Una de las características de Linux es la posibilidad de ampliar en tiempo de ejecución el conjunto de las prestaciones ofrecidas por el núcleo, agregando funcionalidad mientras el sistema está en funcionamiento mediante código extra y que se denomina **módulo**. El núcleo Linux ofrece soporte para diferentes tipos (o clases) de módulos, incluyendo, pero no limitado, a los controladores de dispositivos. Cada módulo se compone de un código objeto que puede ser dinámicamente vinculado con el núcleo en ejecución por el comando ***insmod*** y puede ser quitado por el comando ***rmmod***.

Los módulos utilizan tres tipos de dispositivos. Cada módulo generalmente se refiere a uno de estos tipos y, por lo tanto, se clasifica como módulo de caracteres, módulo de bloque o módulo de red. Esta división de módulos en diferentes tipos o clases no es rígida y el programador puede optar por construir módulos multidispositivos. No obstante, se recomienda crear un módulo diferente para cada nueva funcionalidad para facilitar la escalabilidad y capacidad de ampliación.

a) Los dispositivos de caracteres: un dispositivo de carácter (*char*) es aquel al que se puede acceder como una secuencia de octetos (como un archivo). El controlador implementa al menos las llamadas al sistema de *open*, *close*, *read*, *write* sobre el dispositivo. El texto de la consola (/dev/console) y los puertos serie (/dev/ttyS0) son ejemplos de dispositivos de caracteres, y se puede acceder por medio de los inodos del sistema de archivos, como por ejemplo /dev/lp0 y dev/tty1/.

b) Dispositivos de bloque: de modo similar que los dispositivos de caracteres, a los dispositivos de bloque se accede por los nodos del sistema de archivos en el directorio /dev. Un dispositivo de bloque es algo que puede albergar un sistema de archivos, tales como un disco. Linux permite leer y escribir en un dispositivo de bloque como en un dispositivo de caracteres, por lo que los dispositivos de bloque y los dispositivos de caracteres solo se diferencian en el modo como se gestionan internamente por el núcleo. Un controlador de bloque permite vincular los puntos de entrada en /dev (que contiene los archivos

especiales para dispositivos y que se crean en el momento de la instalación en función de los dispositivos existentes o posteriormente con /dev/MAKEDEV) y así poder ejecutar un *mount* del sistema de archivos.

c) Interfaces de red: toda transacción de red se realiza a través de una interfaz, un dispositivo capaz de intercambiar datos con otros equipos. Generalmente, una interfaz es un dispositivo de hardware, pero también podría ser un dispositivo de software puro, como la interfaz de *loopback*. La interfaz de red se encarga de enviar y recibir paquetes de datos, gestionado por el subsistema de red del núcleo, sin saber cómo las transacciones individuales se asignan a los actuales paquetes que se transmiten. Al ser un dispositivo orientado a paquetes, no se le asigna un nodo en el sistema de archivos y se le asigna un nombre único como *eth0*, *wlan0*, etc. La comunicación entre el núcleo y un controlador de dispositivo de red es diferente, ya que el núcleo es el que llama a funciones relacionadas con la transmisión de paquetes.

Existen otras clases de módulos en Linux para hacer frente a tipos específicos de dispositivos. Por lo tanto, se puede hablar de módulos de bus serie universal (USB), módulos de dispositivos serie, etc., o de módulos de dispositivos SCSI. En este último caso, y a pesar de que todos los periféricos conectados al bus SCSI aparecen en /dev, ya sea como un dispositivo de caracteres o un dispositivo de bloque, la organización interna del software es diferente.

3.12. Estructura de datos dinámica en Linux

El núcleo Linux tiene dos funciones primarias en relación con los dispositivos: controlar el acceso a los dispositivos físicos y establecer cuándo y cómo los procesos interactuarán con estos dispositivos. El directorio /proc/, que también se llama sistema de archivos proc, contiene una jerarquía de archivos especiales que representan el estado actual del núcleo y permite a las aplicaciones y usuarios consultar el estado dinámico de los datos del núcleo del sistema. Dentro del directorio /proc/ existe una gran cantidad de información con detalles sobre el hardware del sistema y cualquier proceso que se esté ejecutando actualmente, y además algunos de los archivos dentro de /proc/ pueden ser manipulados por los usuarios y las aplicaciones para comunicar al núcleo cambios en la configuración.

En Linux todo se guarda en archivos, ya sean binarios o de texto, pero el directorio /proc/ contiene otro tipo de archivos, llamados archivos virtuales. Son los archivos que componen el **sistema de archivos virtuales**, que poseen cualidades únicas. En primer lugar, la mayoría de ellos tienen un tamaño de 0 octetos a pesar de que pueden contener gran cantidad de información cuando se los visualiza. Además, la mayoría de las configuraciones del tiempo y las fechas reflejan el tiempo y la fecha real, lo que es un indicativo de que están siendo constantemente modificados.

Los archivos virtuales, tales como /proc/interrupts, /proc/meminfo, /proc/mounts, y /proc/partitions, proporcionan una vista rápida actualizada del hardware del sistema, y otros –como /proc/filesystems y el directorio /proc/sys/– proveen información de configuración del sistema e interfaces. Estos archivos puede ser visualizados con los comandos habituales (*cat*, *more*, *less*) del SO u otros comandos que utilizan específicamente esta información (*lspci*, *apm*, *free* y *top*). Para cambiarlos, simplemente se debe redirigir la entrada al archivo que se desea cambiar; por ejemplo, para cambiar que el núcleo haga reenvío de paquetes se puede hacer *echo 1 > /proc/sys/net/ipv4/ip_forward*.

/proc/ide/

/proc/ide/, por ejemplo, contiene información sobre los dispositivos IDE.

La siguiente lista expone algunos de los archivos más comunes y útiles que se encuentran en el directorio /proc:

- **/proc/acpi**: proporciona información acerca del estado de la administración de la energía avanzada[38] y es usado por el comando *acpi*.

[38]En inglés, *advanced configuration and power interface.*

- **/proc/buddyinfo**: se utiliza principalmente para diagnosticar problemas de fragmentación de memoria. Utilizando el algoritmo buddy, cada columna representa el número de páginas de un cierto orden (de un cierto tamaño) que están disponibles en un momento dado.

- **/proc/cmdline**: muestra los parámetros pasados al núcleo en el momento en el que este se inicia (por ejemplo, *ro root=/dev/hda1 rhgb quiet 3*).

- **/proc/cpuinfo**: identifica el tipo de procesador usado por el sistema.

- **/proc/crypto**: lista todos los códigos de cifrado utilizados por el núcleo Linux, incluyendo detalles adicionales para cada uno.

- **/proc/devices**: muestra los distintos dispositivos de caracteres y de bloque que hay configurados en cada momento (no incluye dispositivos cuyos módulos no están cargados).

- **/proc/dma**: contiene una lista de los canales registrados DMA ISA en uso.

- **/proc/execdomains**: enumera los dominios de ejecución soportados en cada momento por el núcleo Linux.

- **/proc/fb**: dispositivos *frame buffer*, con el número del dispositivo *frame buffer* y su controlador.

- **/proc/filesystems**: lista los tipos del sistema de archivos soportados en cada momento por el núcleo.

- **/proc/interrupts**: número de interrupciones por IRQ en la arquitectura x86.

- **/proc/iomem**: mapa actual de la memoria del sistema para los distintos dispositivos.

- **/proc/ioports**: proporciona una lista de las regiones de puertos registrados que se utilizan en cada momento para la comunicación de entrada y salida con un dispositivo.

- **/proc/kcore**: representa la memoria física del sistema y se almacena en el formato de archivos base. A diferencia de la mayoría de los archivos /proc/, kcore muestra un tamaño. Este valor se da en octetos y es igual al tamaño de la memoria física (RAM) utilizada más 4 Kb.

- **/proc/kmsg**: mantiene los mensajes generados por el núcleo. Luego, estos mensajes son recogidos por otros programas, como por ejemplo /sbin/klogd o /bin/dmesg.

- **/proc/loadavg**: ofrece una vista de la carga promedio del procesador con respecto al sobretiempo de CPU y de E/S, así como también datos adicionales utilizados por *uptime* y otros comandos.

- **/proc/locks**: archivos que están bloqueados por el núcleo. El contenido de este archivo contiene datos internos de depuración y puede variar enormemente, dependiendo del uso del sistema.

- **/proc/mdstat**: información actual sobre la configuración de discos múltiples de RAID.

- **/proc/meminfo**: información sobre el uso de RAM en el sistema.

- **/proc/misc**: controladores registrados en el principal dispositivo de misceláneos, que es el número 10.

- **/proc/modules**: módulos cargados en el sistema.

- **/proc/mounts**: todos los montajes en uso por el sistema.

- **/proc/mtrr**: actual *memory type range registers* (MTRR), en uso dentro del sistema (siempre que la arquitectura del sistema soporte MTRR).

- **/proc/partitions**: información sobre la asignación de bloques de particiones.

- **/proc/pci**: lista completa de cada dispositivo PCI en el sistema.

- **/proc/slabinfo**: información sobre el uso de memoria en el nivel *slab*. Los núcleos de Linux superiores a la versión 2.2 usan *slab pools* para manejar memoria por encima del nivel de página.

- **/proc/stat**: registro de las diferentes estadísticas sobre el sistema desde que fue reiniciado por última vez.

- **/proc/sysrq-trigger**: utilizando el comando *echo* para escribir a este archivo, un usuario *root* remoto puede ejecutar la mayoría de los comandos de petición del sistema[39] remotamente como si estuviese en el terminal local. Para hacer *echo* con los valores a este archivo, /proc/sys/kernel/sysrq debe estar configurado a un valor diferente de 0.

- **/proc/uptime**: contiene información sobre el tiempo que lleva encendido el sistema desde el último reinicio.

- **/proc/version**: muestra la versión del núcleo Linux y *gcc* en uso, así como la versión de la distribución.

También existen una serie de directorios que contienen información, como los **directorios de proceso**, que son un conjunto de directorios nombrados con un número. Se denominan directorios de proceso porque pueden hacer referencia a un ID de proceso y contener información específica para ese proceso (*cmdline*, *cwd*, *environ*, *exe*, *fd*, *stat*, etc.):

- **/proc/self**: es un enlace al proceso en ejecución. Esto le permite verse a sí mismo sin tener que conocer su ID de proceso.

- **/proc/bus/**: contiene información específica sobre los distintos buses disponibles en el sistema. Por ejemplo, en un sistema estándar que contenga buses PCI y USB, los datos en cada uno de estos buses en cada momento están disponibles en un subdirectorio bajo /proc/bus/ con el mismo nombre, tal como /proc/bus/pci/.

- **/proc/driver/**: contiene información acerca de los controladores específicos que el núcleo utiliza. Por ejemplo, rtc es un archivo habitual que proporciona la salida desde el controlador para el reloj del sistema[40], que es el dispositivo que mantiene la hora mientras el sistema está apagado.

- **/proc/fs**: muestra qué sistemas de archivos están siendo exportados. Si está usando un servidor NFS, escribiendo cat /proc/fs/nfsd/exports se podrán visualizar los sistemas de archivos que se comparten y los permisos acordados a esos sistemas de archivos.

(39) En inglés, *system request key*.

(40) En inglés, *real time clock*.

- **/proc/ide/**: contiene información sobre los dispositivos IDE del sistema. Cada canal IDE está representado como un directorio separado, tal como /proc/ide/ide0 y /proc/ide/ide1.

- **/proc/irq/**: se usa para configurar la afinidad de una IRQ con una CPU, lo que le permite conectar una IRQ particular a una sola CPU. De manera alternativa, puede evitar que una CPU manipule cualquier IRQ.

- **/proc/net/**: visión exhaustiva de distintos parámetros y estadísticas de red. Cada directorio y archivo virtual dentro de este directorio describe aspectos sobre la configuración de la red en el sistema.

- **/proc/scsi/**: análogo al directorio /proc/ide/, pero es solo para dispositivos SCSI conectados.

- **/proc/sys/**: proporciona información sobre el sistema, pero también permite al administrador activar y desactivar inmediatamente características del núcleo.

- **/proc/sys/dev/**: proporciona parámetros para dispositivos particulares en el sistema. La mayoría de los sistemas tienen al menos dos directorios: cdrom y raid.

- **/proc/sys/fs/**: compendio de opciones y de información referente a varios aspectos del sistema de archivos, incluyendo la información de cuotas, manipulación de archivos, *inode* y *dentry*.

- **/proc/sys/kernel/**: contiene una variedad de archivos de configuración diferentes que afectan directamente a la operación del núcleo.

- **/proc/sys/net**: contiene varios subdirectorios que tratan sobre redes. Las diferentes configuraciones en el momento en el que el núcleo fue compilado colocan diferentes directorios aquí, tales como appletalk, ethernet, ipv4, ipx y ipv6. Los administradores de sistemas podrán ajustar la configuración de la red en un sistema en funcionamiento alterando los archivos en estos directorios.

- **/proc/sys/vm/**: configuración del subsistema de memoria virtual del núcleo Linux. El núcleo hace un uso extensivo e inteligente de la memoria virtual, conocida comúnmente como espacio *swap*.

- **/proc/sysvipc**: información sobre los recursos System V IPC. Los archivos de este directorio están relacionados con las llamadas al System V IPC de mensajes (*msg*), semáforos (*sem*) y memoria compartida (*shm*).

- **/proc/tty/**: información sobre los dispositivos *tty* disponibles y usados en el sistema.

Es interesante la aplicación lxr.linux.no <http://lxr.linux.no/+trees>, que permite consultar el código de Linux en forma interactiva y acceder a las estructuras de datos en forma simple y navegable.

3.13. Multinúcleo, multiprocesador y multiordenador

Dada la necesidad cada vez más acuciada de ordenadores más potentes y con mayor capacidad de proceso, es cada vez más habitual encontrar procesador con multinúcleo, u ordenadores con más de un procesador o sistemas con más de un ordenador. En los dos primeros casos, el SO tiene una función crucial, ya que los elementos de cómputo están bajo su dominio y en algunos casos comparten todo el hardware y en otros parte de él, pero siempre bajo el mismo SO. En el tercer caso son ordenadores que trabajan cooperativa o colaborativamente en la resolución de un programa (llamado programa paralelo) y bajo el dominio de diferentes sistemas operativos.

En un sentido amplio se puede hablar de ordenadores paralelos clasificándolos según el nivel de paralelismo que admite su hardware: los ordenadores multinúcleo y multiprocesador tienen varios elementos de procesamiento en una sola máquina, mientras que los clústeres, y los *grids/clouds* emplean varios ordenadores para trabajar en la misma tarea.

Un **microprocesador multinúcleo** es aquel que dispone de dos o más unidades de procesamiento en un solo encapsulado, es decir, que comparten bus, memoria principal y el resto de los dispositivos.

Generalmente, los microprocesadores multinúcleo permiten que un dispositivo pueda ejecutar código simultáneo diferente por medio de lo que se denomina paralelismo a nivel de hilos (TLP[41]), también conocido como multiprocesamiento a nivel de chip (CMP[42]) y el SO tendrá un papel relevante para permitir esta funcionalidad.

(41)TLP es la sigla de *thread-level parallelism*.

(42)CMP es la sigla de *chip-level multiprocessing*.

Un **sistema multiprocesador** es un ordenador que cuenta con dos o más microprocesadores (CPU) y por ello el sistema puede ejecutar simultáneamente varios hilos pertenecientes a un mismo proceso o a procesos diferentes.

Los ordenadores multiprocesador presentan problemas de diseño que no se encuentran en ordenadores monoprocesador y vienen dados por el hecho de que dos programas pueden ejecutarse simultáneamente y, potencialmente, pueden interferirse entre sí en lo que se refiere a las lecturas y escrituras en memoria.

La solución pasa por dos tipos de arquitecturas que resuelven estos problemas: la arquitectura NUMA, donde cada procesador tiene acceso y control exclusivo a una parte de la memoria o la arquitectura SMP, donde todos los procesadores comparten toda la memoria. En ambas arquitecturas es fundamental el control y la gestión del SO para mantener la coherencia y el acceso a la información sin generar errores ni fallos.

Las arquitecturas SMP deben enfrentarse al problema de la coherencia de caché, ya que cada microprocesador cuenta con su propia memoria caché local. De manera que cuando un microprocesador escribe en una dirección de memoria, lo hace únicamente sobre su copia local en caché y si otro microprocesador tiene almacenada la misma dirección de memoria en su caché, resultará que trabaja con una copia obsoleta del dato almacenado. Para que un multiprocesador opere correctamente, necesita un sistema operativo especialmente diseñado para ello y la mayoría de los núcleos actuales poseen esta capacidad.

La **computación paralela** es una técnica de programación en la que muchas instrucciones se ejecutan simultáneamente en ordenadores diferentes y se sustenta en el principio de que los problemas grandes se pueden dividir en partes más pequeñas que pueden resolverse de modo concurrente ("en paralelo") sobre diferentes procesadores/ordenadores.

Existen varios tipos de computación paralela: paralelismo a nivel de bit, paralelismo a nivel de instrucción, paralelismo de datos y paralelismo de tareas. En los últimos años, la computación paralela se ha aplicado en la computación de altas prestaciones con interés creciente debido a las restricciones físicas que impiden el escalado en frecuencia de las CPU. La computación paralela se ha convertido en el paradigma más utilizado en la arquitectura de computadores, que van desde los procesadores multinúcleo a los clústeres de ordenadores y hoy en día a los grandes *clouds* de procesamiento masivo. Uno de los principales problemas que han surgido y que se ha transformado en una fuente de preocupación es el consumo de energía de los ordenadores paralelos, antes que el coste de la infraestructura o la complejidad de su programación.

Es importante también tener en cuenta que los programas de ordenadores paralelos son más difíciles de escribir que los secuenciales, ya que la concurrencia introduce nuevos tipos de errores de software (las condiciones de carrera son los más comunes) y la comunicación y la sincronización entre las diferentes subtareas son típicamente las grandes barreras para conseguir un buen rendimiento de los programas paralelos. El incremento de velocidad que consigue un programa como resultado de la paralelización viene dado por la ley de Amdahl. Según esta ley: "Hay un límite de cuánto puede uno mejorar en velocidad una cosa si solo se optimiza una parte de ella"; es decir, si se tiene un programa

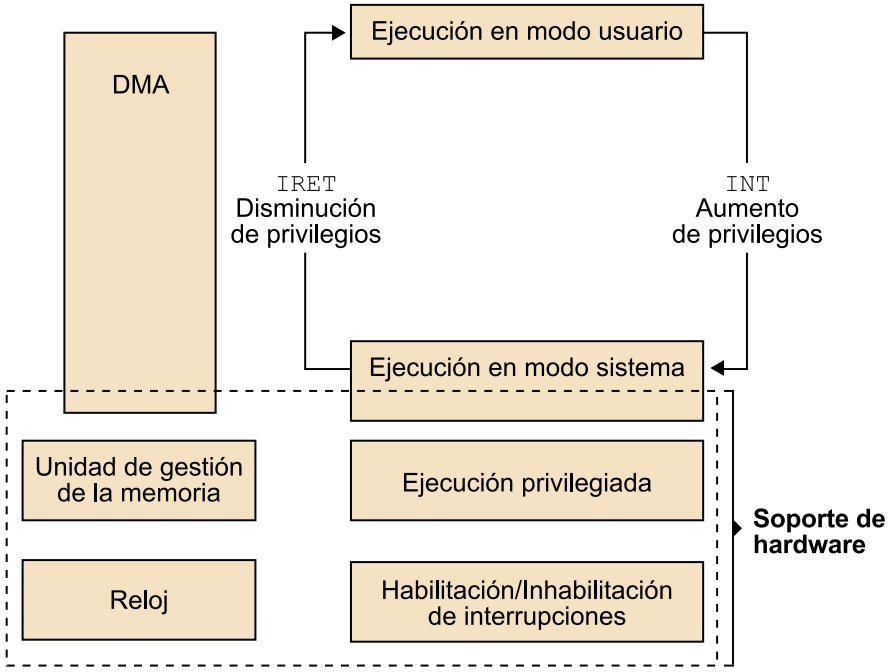
que utiliza el 10% de la CPU y se optimiza reduciendo la utilización en un factor 2, el programa mejorará sus prestaciones (*speed-up*) en un 5%, lo cual puede significar un tremendo esfuerzo no compensado por los resultados.

Resumen

En este módulo se ha explicado el sistema operativo desde el punto de vista de su estructura y se han tenido en cuenta las principales organizaciones que se utilizan para su construcción.

También se ha analizado el soporte de hardware destinado a mantener un entorno de trabajo fiable y una gestión eficiente de los recursos del sistema. El mapa conceptual (figura 21) recopila los elementos que forman este soporte de hardware.

Figura 21. Mapa conceptual del soporte de hardware



Actividades

1. Analizad y encontrad cuáles son las principales estructuras de datos vinculadas a un proceso utilizando http://lxr.linux.no/+trees o el código fuente de Linux.

2. Encontrad cómo se mantiene la hora y el tiempo actual en el código en Linux.

3. Analizad la estructura del planificador de Linux, verificad el valor asignado al tiempo máximo de ejecución de la tarea y analizad las implicaciones que tiene aumentar o reducir este valor.

4. Buscad en la bibliografía recomendada la unidad de gestión de la memoria de un procesador conocido. Observad los registros de control más característicos.

Ejercicios de autoevaluación

1. Suponed que un sistema operativo tiene una estructura interna monolítica. Teniendo en cuenta esta afirmación se puede afirmar que...

a) está formado por un conjunto de servicios en los que no se han definido ni los puntos de entrada a los servicios ni los parámetros. La aplicación de usuario puede definirlos.
b) se ha construido con una estructura mínima; prácticamente es una colección de procedimientos que se pueden comunicar.
c) agrupa los servicios según su funcionalidad.
d) se ha programado a partir de un único código fuente. El inconveniente es que cualquier modificación implica la recompilación de todo el código.
e) está basado en una programación orientada a objetos y en una construcción cliente/servidor.

2. Cuando un proceso de usuario pide un servicio de sistema ejecutando un *trap*, una vez entra en el núcleo...

a) entra en un modo de ejecución sin privilegios, ya que aún continúa en el entorno del proceso de usuario.
b) el área de la memoria a la que puede acceder queda reducida al espacio lógico del núcleo, es decir, a las posiciones bajas de la memoria.
c) es necesario un soporte de hardware para cambiar el aumento de los privilegios correspondiente.
d) puede acceder a todo el espacio de memoria del sistema, dado que desde el núcleo tiene todos los privilegios para hacerlo.
e) nunca puede pedir la ejecución de código del núcleo.

3. La técnica de acceso directo a la memoria (DMA), en relación con la gestión de la transferencia de información entre periféricos y memoria...

a) permite que el periférico tenga prioridad para capturar los buses del sistema.
b) se encarga de crear diferentes espacios disyuntos de memoria para aumentar la eficiencia.
c) puede transferir los bloques de información entre el periférico y la memoria principal sin que tengan que intervenir los buses del sistema.
d) es utilizada para transferir los bloques de información de manera independiente de la actividad del procesador.
e) permite programar los comandos en los registros de control del periférico de manera mucho más rápida.

Solucionario

Ejercicios de autoevaluación

1. b

2. c

3. d

Glosario

direct acces to memory *m* Acceso directo a la memoria por parte de un periférico. Operación que permite hacer transferencias de bloques de información entre un periférico y la memoria sin la intervención del procesador utilizando los buses del sistema cuando no los utiliza el procesador.
sigla **DMA**

DMA *m* Véase ***direct acces to memory***.

instrucción privilegiada *f* Instrucción que solo se puede ejecutar en modo de ejecución de alto nivel, habitualmente en el modo sistema (o núcleo).

MMU *f* Véase **unidad de gestión de la memoria (*memory management unit*)**.

modo de ejecución *m* Modo de operar del procesador que permite el control de ejecución de ciertas instrucciones privilegiadas y el acceso a ciertas áreas del sistema. Los diferentes modos de ejecución forman una estructura jerárquica.

reloj del sistema (*clock*) *m* Conjunto de circuitos electrónicos que, de manera autónoma, mantienen el día y la hora del sistema y generan interrupciones periódicas de hardware que permiten activar las rutinas correspondientes del SO para hacer los cambios de contexto necesarios en un SO de tiempo compartido.

SCSI *m* Véase ***small computer system interface***.

small computer system interface *m* Controlador múltiple de dispositivos que tiene su propio bus para gestionar hasta siete periféricos desde la misma ranura (*slot*) de la placa madre del sistema. Es una interfaz de altas prestaciones.
sigla **SCSI**

***symmetric multi-processing* (multiproceso simétrico)** *m* Tipo de arquitectura de ordenadores en la que dos o más procesadores comparten una única memoria central; también se denomina UMA (*uniform memory access*). Todos los microprocesadores compiten en igualdad de condiciones por dicho acceso, de ahí la denominación de simétrico.
sigla **SMP**

unidad de gestión de la memoria (*memory management unit*) *f* Unidad que traduce las direcciones lógicas facilitadas por el procesador a direcciones físicas de la memoria del sistema.
sigla **MMU**

thread-level parallelism *f* Técnica que permite en un procesador multinúcleo o multiprocesador ejecutar diferentes *threads* (hilos de ejecución).
sigla **TLP**

Bibliografía

Bach, M. J. (1986). *The Design of the UNIX Operating System*. Prentice Hall.

Benvenuti, Ch. (2006). *Understanding Linux Network Internals*. O'Reilly Media.

Carretero Pérez, J. (2010). *Sistemas Operativos: una visión aplicada*. McGraw Hill.

Hennessy, J.; Patterson, D. (2006). *Computer Architecture: A Quantitative Approach* (4.ª ed.). Morgan Kaufman.

Nutt, G. (2006). *Sistemas Operativos*. Pearson Educación.

Rubini, A.; Corbet, J. (2001). *Linux Device Drivers* <http://www.xml.com/ldd/chapter/book/index.html>.

Stallings, W. (2011). *Operating Systems: Internals and Design Principles*. Prentice Hall.

Silberschatz, A. (2000). *Sistemas Operativos*. Addison Wesley Longman.

Tanenbaum, A. (2009). *Sistemas Operativos Modernos*. Prentice Hall.

Vahalia, U. (1995). *UNIX Internals: The New Frontiers*. Prentice Hall.

Mapa interactivo del núcleo de Linux <http://www.makelinux.net/kernel_map> (accedido en junio de 2011)