La gestión de procesos

Porfidio Hernández Budé
Enric Morancho Llena

PID_00180053

UOC
Universitat Oberta de Catalunya

Índice

Introducción

En este módulo presentamos la **gestión de procesos** desde el punto de vista del sistema operativo. Empezamos recordando el concepto de **proceso** y, a continuación, introducimos el concepto de **hilo de ejecución**.

Después de ver estos conceptos básicos nos centramos en el problema que debe resolver cualquier sistema operativo multiprogramado: permitir la ejecución concurrente de cierto número de programas sobre una cantidad de procesadores mucho más reducida. Dividiremos este problema en varias partes y las intentaremos solucionar por separado.

Objetivos

En los materiales didácticos facilitados en este módulo, encontraréis las herramientas necesarias para alcanzar los siguientes objetivos:

1. Aprender los conceptos de *proceso* y de *hilo de ejecución*.

2. Conocer los diferentes estados en que se puede encontrar un proceso y los motivos que provocan un cambio de estado.

3. Entender la necesidad de realizar cambios de contexto para aprovechar mejor el rendimiento del computador.

4. Relacionar la cantidad de procesos que se dan en la máquina con el rendimiento que se consigue de ellos.

5. Entender las funciones de los diferentes niveles de planificación del procesador.

6. Comprender los algoritmos de planificación del procesador y detectar las ventajas y los inconvenientes.

1. Procesos e hilos de ejecución

1.1. Monoprogramación y multiprogramación

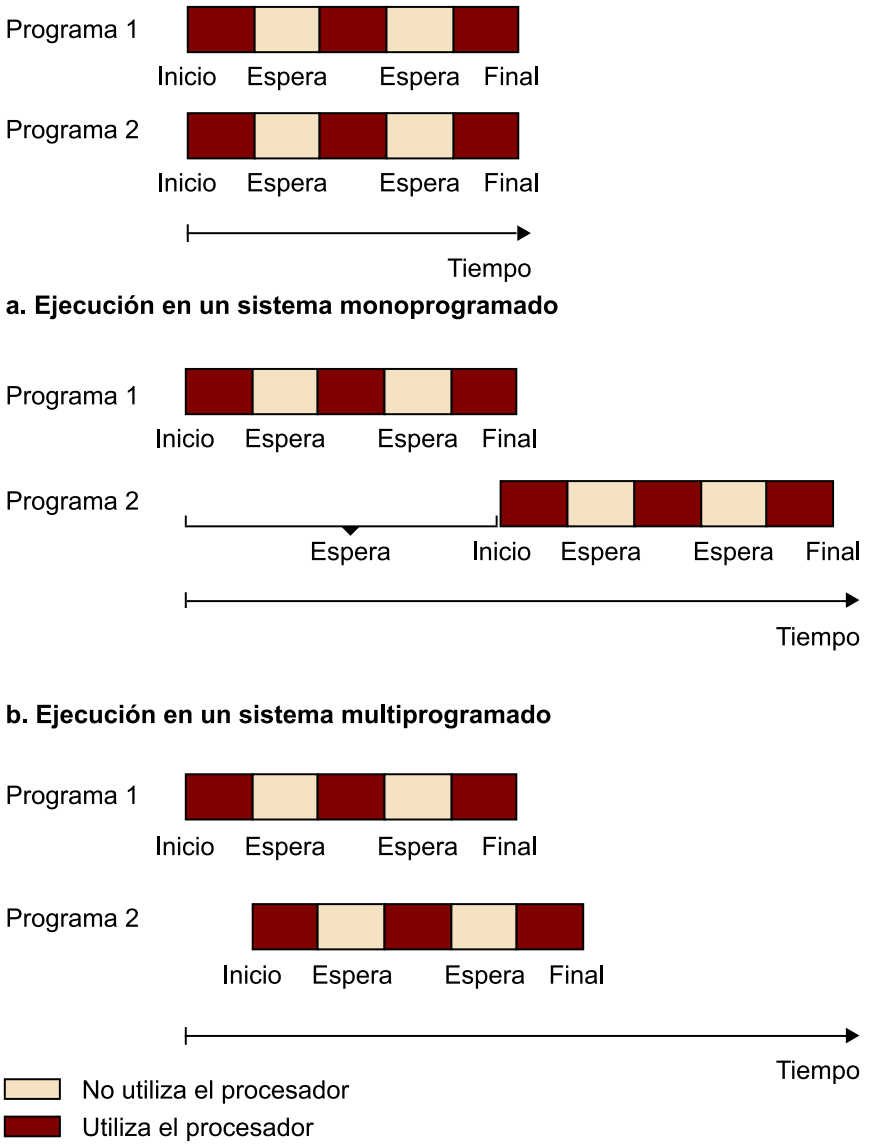
Los **sistemas operativos monoprogramados** sólo permiten la ejecución de un programa, de forma que hasta que no finaliza la ejecución o el programa es abortado por el usuario, el SO no puede ejecutar ningún otro. En estos SO, el programa en ejecución suele tener un control absoluto sobre los recursos de la máquina, ya que los utiliza de manera exclusiva; al acabar la ejecución del programa, la máquina queda disponible para el programa que deba ejecutarse a continuación.

Actualmente, casi todos los SO son multiprogramados. Los **sistemas operativos multiprogramados** permiten ejecutar diferentes programas de forma concurrente, compartiendo los diferentes recursos de la máquina. Esto permite un mejor aprovechamiento de estos recursos.

Ejecución en un SO monoprogramado y en un SO multiprogramado

La figura 1 muestra las diferencias que existen entre ejecutar dos programas en un SO monoprogramado y ejecutarlos en un SO multiprogramado. Hemos representado ambos programas teniendo en cuenta los momentos en que usan el procesador (zonas oscuras) y los momentos en que esperan algún suceso (zonas blancas). Durante estos momentos de espera (por ejemplo, los instantes antes de que el usuario pulse una tecla) el programa no utiliza el procesador.

Figura 1. Representación de los programas



Si ejecutamos los programas en un SO que sólo ofrece monoprogramación (caso **a**), el segundo programa no podrá empezar a ejecutarse hasta que no haya finalizado el primero. En cambio, si los ejecutamos en un SO que ofrece multiprogramación (caso **b**), podemos solapar la ejecución de ambos programas, de forma que mientras el primero no utiliza el procesador, el segundo sí. De esta forma, el SO puede aprovechar mejor determinados recursos de la máquina, como el procesador. Así pues, si el SO que tenemos instalado es multiprogramado, podremos ejecutar de forma íntegra ambos programas aproximadamente en la mitad del tiempo que invertiríamos en utilizar un SO monoprogramado.

En un SO multiprogramado, los diferentes programas que se encuentran en ejecución comparten los recursos de la máquina. El SO debe garantizar que los diferentes programas activos se podrán ejecutar sin que interfieran de manera no deseada, para que el resultado de ambas ejecuciones sea correcto. Por ejemplo, si un programa en ejecución almacena una variable en una posición de memoria, ningún otro programa en ejecución debería poder modificarla.

Ved también

Consultad los mecanismos que permiten ejecutar diferentes procesos en exclusión mutua en el módulo didáctico "Comunicación y sincronización" de la asignatura *Sistemas operativos*.

Por lo tanto, un SO multiprogramado debería ser capaz de aislar cada programa en ejecución del resto de los programas que se encuentran en ejecución en la máquina, y garantizar que sólo pueda acceder a los recursos (memoria, ficheros, etc.) el programa que los tiene asignados.

1.2. El concepto de proceso

No existe una definición unánimemente aceptada de *proceso*. Además, en algunos SO se habla de *trabajo* (*job*) o de *tarea* (*task*) para referirse a este mismo concepto. De todas formas, según la definición más ampliamente aceptada, un proceso es, sencillamente, un programa en ejecución.

Ved también

Consultad la definición del término proceso en el apartado 1 del módulo "La gestión de procesos" de la asignatura *Sistemas operativos*.

Para entender esta definición hay que diferenciar los programas guardados en ficheros ejecutables de los programas en ejecución:

a) En el primer caso hablamos de una entidad pasiva.

b) En el segundo, hablamos de una entidad activa que se crea a partir de un fichero ejecutable que necesita memoria[1], dispositivos para realizar operaciones de entrada/salida y un procesador para ejecutar el código, y que debe ser protegida del resto de los procesos que se encuentran en ejecución en la máquina.

(1)Los programas en ejecución necesitan memoria para guardar su código, sus datos y su pila.

Podemos considerar el fichero ejecutable como un molde de hacer procesos, puesto que un mismo fichero ejecutable puede generar procesos diferentes.

Ejemplo

Si diferentes usuarios pretenden compilar un programa escrito en lenguaje C, cada uno creará un proceso diferente a partir del fichero ejecutable correspondiente al compilador de C.

A continuación mostramos otras definiciones posibles del término *proceso*:

- Espíritu animado de un procedimiento.
- Centro de control de un procedimiento en ejecución.
- Entidad a la que se asignan los recursos de un computador.
- Instanciación de un programa.

(2)En inglés, *thread*.

1.2.1. Hilos de ejecución

En este punto veremos una extensión del concepto de proceso: el hilo de ejecución[2]. A veces, los hilos de ejecución también reciben el nombre *de procesos ligeros* (*lightweight processes*), subprocesos o subtareas.

Cuando escribimos un algoritmo estamos acostumbrados a partir de una hipótesis implícita: la secuencialidad. Es decir, damos por hecho que durante la ejecución de un programa, éste sólo podrá realizar una determinada acción

en cada periodo de tiempo (por ejemplo, no podrá ejecutar simultáneamente dos instrucciones de lenguaje máquina). Podemos decir que tenemos un único hilo de ejecución definido por un contador de programa (PC).

De todas formas, algunos algoritmos pueden incluir partes que sean independientes, y que, por lo tanto, se podrían llevar a cabo simultáneamente o en paralelo, hecho que se podría aprovechar para sacar más partido de la máquina.

Por ejemplo, supongamos que queremos calcular el mínimo, el máximo y la media de los valores almacenados en un vector. Un posible algoritmo secuencial para resolver este problema sería el que encontramos a continuación.

```
vector_enteros v;
enteros max, min, med;

Programa() {
    InicializarVector(v);
    max = Maximo(v);
    min = Minimo(v);
    med = Media(v);
    EscribirResultados(max, min, med);
}
```

Este programa inicializa un vector y a continuación calcula el máximo de sus elementos, posteriormente el mínimo y, en último lugar, la media. Para terminar, escribe estos tres valores. De forma implícita partimos del supuesto de que tenemos un único contador de programa que se refiere al código del programa y que indica cuál es la instrucción que se ejecuta en cada momento. Es decir, la ejecución de este programa crea un proceso con un único hilo de ejecución.

Analizando el algoritmo anterior podemos constatar que el cálculo del máximo, del mínimo y de la media de los elementos del vector son tareas independientes. Por lo tanto, podríamos pensar en un tipo diferente de solución, como puede ser la que propone el programa en pseudocódigo que mostramos a continuación.

```
vector_enteros v;
enteros max, min, med;

Programa() {
    IncializarVector(v);
    CrearHilo(Hilo1);
    CrearHilo(Hilo2);
    CrearHilo(Hilo3);
    EsperarFinal_Hilos();
```

```
    EscribirResultados(max, min, med);
}
Hilo1() {
    max = Maximo(v);
    FinFlujo();
}
Hilo3() {
    med = Media(v);
    FinFlujo( );
}
Hilo2() {
    min = Minimo(v);
    FinFlujo( );
}
```

En esta solución, el hilo principal del programa crea tres hilos de ejecución (mediante la llamada al sistema *CrearHilo*), y cada uno se encarga de realizar una parte del cálculo. Cada hilo tiene un contador de programa (PC) propio e, independientemente de los demás, ejecuta la rutina que le corresponde. Cuando las tres partes están calculadas (los tres hilos han ejecutado *FinFlujo*), el hilo principal se encarga de escribir los resultados.

Si la máquina tuviera suficientes procesadores físicos, el SO podría hacer que cada hilo se ejecutara en un procesador físico diferente. De esta forma, el programa tardaría menos tiempo en ejecutarse, puesto que los tres cálculos se realizarían de forma simultánea.

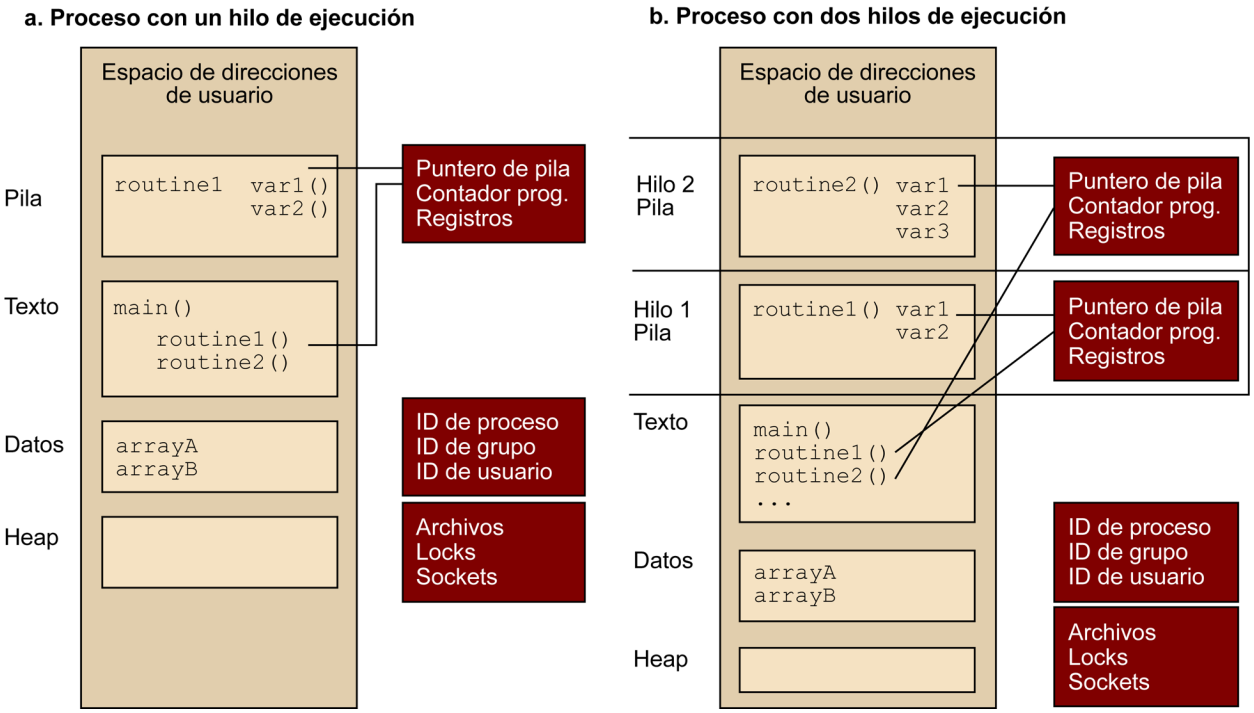
En cambio, si la máquina sólo dispusiera de un procesador físico, el SO no podría aprovechar el paralelismo del programa, y todo el programa se ejecutaría de forma secuencial en el único procesador disponible. Sin embargo, si algún día cambiara la configuración de la máquina y se instalara un procesador adicional, el tiempo de ejecución del programa experimentaría una reducción sin que el usuario tuviera que reescribir ni volver a compilar el programa.

La **representación interna de un proceso** depende el número de hilos de ejecución que tenga el proceso. Desde el punto de vista del SO, todos los hilos de un proceso comparten todos los recursos asignados al proceso, por ejemplo, el espacio lógico. Internamente, cada hilo de ejecución de un proceso se representa por un PC, que indica qué instrucción de lenguaje máquina ejecuta, y por una pila donde están guardadas las variables locales en el hilo y los bloques de activación de las rutinas activas en el hilo, así como los registros y el estado del mismo.

Representación interna de dos procesos

La figura 2 muestra la representación interna de dos procesos, uno con un único hilo de ejecución y otro con cuatro:

Figura 2. Representación interna de dos procesos



El hecho de escribir los algoritmos identificando los hilos de ejecución tiene la ventaja de que permite al SO aprovechar mejor los recursos de la máquina, pero también presenta algunos inconvenientes:

a) Uno de los problemas es que la identificación de las partes del programa que son independientes debe ser efectuada por el programador, y esto no siempre es tan sencillo como en el ejemplo en que calculábamos el máximo, el mínimo y la media de los valores de un vector. Existen grupos de investigación que trabajan en este tema; buscan mecanismos automáticos que permitan detectar qué partes de un programa se pueden ejecutar de forma concurrente y cuáles deben ser ejecutadas secuencialmente. No obstante, por lo general la determinación de las partes que se pueden ejecutar en paralelo es responsabilidad del programador.

b) Se nos plantea otro problema cuando los hilos de ejecución interaccionan entre sí (por ejemplo, cuando un hilo no puede consultar una variable hasta que otro hilo no la haya modificado).

Es importante diferenciar, por un lado, un proceso con diferentes hilos de ejecución y por otro, diferentes procesos con un único hilo de ejecución. A pesar de que el SO puede ejecutar los diferentes hilos concurrentemente en ambos casos, en el primero los hilos comparten recursos (por ejemplo, el espacio lógico), mientras que en el segundo los hilos no comparten ningún recurso.

1.2.2. Uso de subprocesos (subtareas, hilos de ejecución o procesos ligeros)

La descomposición de las aplicaciones en múltiples subprocesos permite la posibilidad de su ejecución paralela; además, nos ofrece la posibilidad de compartir un espacio común de direcciones y datos por distintos flujos de ejecución. Esta posibilidad de compartición provoca que aplicaciones con estos requisitos puedan utilizar este nuevo modelo de solución de manera eficiente.

La mayor facilidad a la hora de su creación y eliminación (2 órdenes de magnitud más rápidos que los procesos convencionales) hace de los subprocesos, en entornos donde el número de estos cambia de manera dinámica, una alternativa muy adecuada para este tipo de situaciones.

Debido a la poca información que el sistema operativo debe guardar en relación con los subprocesos, el cambio de contexto se realiza con rapidez. Por ello en entornos donde las operaciones de cómputo de diferentes subprocesos alternan su ejecución con operaciones de entrada/salida, el solapamiento de estas operaciones beneficiará de manera significativa al tiempo de ejecución de las aplicaciones involucradas.

No obstante, la característica más relevante de los subprocesos para el programador de aplicaciones consiste, sin duda, en mantener el modelo de procesos secuenciales utilizando llamadas al sistema bloqueantes. Supongamos como ejemplo el diseño de un servidor concurrente, diseñado a partir del modelo de proceso. En este sistema, cuando llega una solicitud al servidor, se examina por parte del proceso que atiende las peticiones y éste dispara un nuevo proceso para su ejecución concurrente. No obstante, si la nueva petición genera una petición al disco, el servidor se bloqueará y perderemos prestaciones; si no se quiere bloquear el sistema, el diseñador estará obligado a generar llamadas al disco no bloqueantes, y deberá por tanto manejar las interrupciones involucradas en el proceso, lo que complicará la programación de manera notable. Las llamadas al sistema de naturaleza bloqueante facilitan la programación, y el paralelismo que permiten los subprocesos mejora el rendimiento.

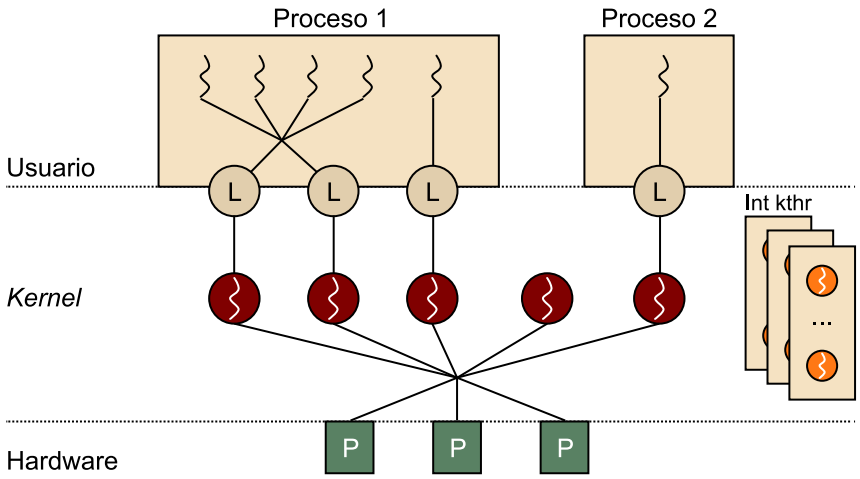
Implementación de subprocesos

Hay diferentes maneras de implementar subprocesos: en el espacio de usuario, en el de núcleo y formas híbridas. El aspecto más notable que se debe tener en cuenta reside en saber si el núcleo tiene o no conciencia de los subprocesos, a efectos de poder planificarlos de manera adecuada. Si el núcleo no sabe que existen los subprocesos, deberá ser el usuario quien se encargue de su planificación y manejo mediante código en el espacio de usuario. Si el núcleo conoce

la existencia de los subprocesos, la tabla de subprocesos es administrada por el sistema y podrá planificar los subprocesos intercalando, si lo desea, subprocesos de aplicaciones diferentes.

Frente a los rápidos cambios de contexto, característica importante de las implementaciones en el espacio de usuario y la posibilidad de definir su propio algoritmo de planificación para cada aplicación, el problema de las llamadas bloqueantes es un problema importante en este tipo de implementación.

Figura 3. Relaciones *m*:*n* y 1:1 entre procesos de usuario y núcleo



La figura 3 muestra la relación de las diferentes entidades que configuran el sistema. En el caso del proceso 1, se muestra la asociación de múltiples subprocesos en el nivel de usuario con múltiples subprocesos en modo núcleo; en el proceso 2, la asociación de un subproceso en el nivel de usuario con un subproceso en el nivel de núcleo.

La combinación de los subprocesos en el nivel usuario y en el de núcleo permite explotar la concurrencia de manera más eficiente en las aplicaciones.

Por defecto, en este módulo presuponemos que trabajamos con procesos con un único hilo de ejecución, a pesar de que los conceptos que presentaremos son fácilmente extensibles a procesos con más de un hilo de ejecución. Cuando los comentarios se refieran a procesos con diferentes hilos de ejecución, se indicará explícitamente.

1.3. Estados de los procesos

Cualquier proceso tiene un ciclo de vida. El SO crea los procesos como respuesta a una petición formulada mediante la llamada al sistema correspondiente (*crear_proceso*), y los destruye cuando el programa que ejecuta invoca la llamada al sistema que la hace finalizar (*destruir_proceso*). Entre ambos sucesos se produce la ejecución del proceso.

Ved también

Consultad el ciclo de vida de un proceso en el apartado 2 del módulo "La gestión de procesos" de la asignatura *Sistemas operativos*.

Hay que tener presente que en un SO multiprogramado podemos tener más procesos que procesadores y, por lo tanto, pueden existir procesos que estén sin ejecutarse, esperando que el SO les asigne algún procesador. Asimismo, los procesos pueden pedir servicios al SO que no se pueden satisfacer inmediatamente[3] y, por lo normal, el proceso no se continuará ejecutando hasta que el servicio no se haya llevado a cabo. En función de la semántica de las llamadas al sistema, el proceso se puede continuar ejecutando aunque la petición no se haya servido; en este caso hablamos de **llamadas al sistema asíncronas** o no bloqueadoras. Estas llamadas modifican sustancialmente la manera de escribir los programas. En este módulo supondremos que todas las peticiones de entrada/salida son bloqueadoras.

(3)Normalmente, por ejemplo, la lectura de un bloque de disco no se puede satisfacer de forma inmediata.

Desde el punto de vista del SO, es muy importante saber exactamente qué está haciendo cada proceso en cada momento para poder planificar correctamente la utilización de los procesadores. El **estado de un proceso** se define como una descripción de su actividad en un momento dado.

A continuación mostramos una lista de algunos de los estados posibles de los procesos; más adelante añadiremos algún estado suplementario:

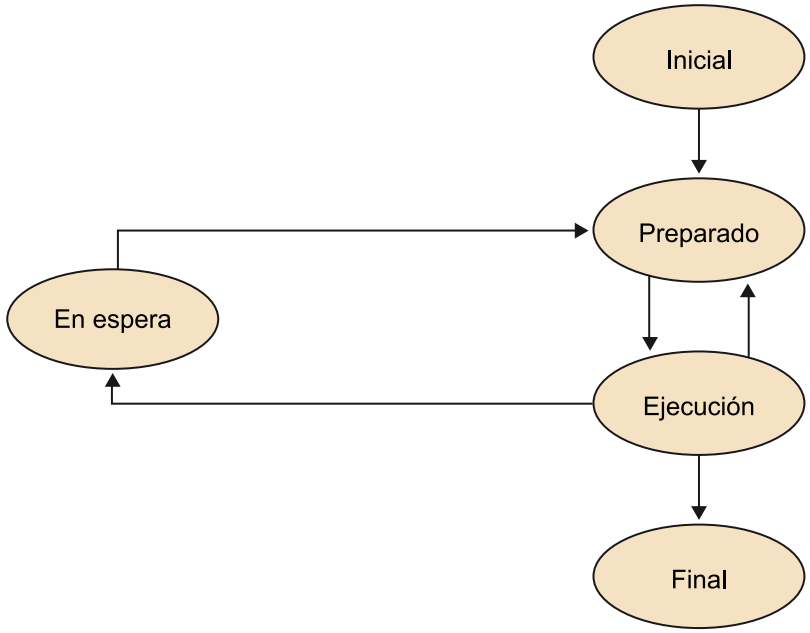
Ved también

Analizad la presentación de algún estado suplementario en el subapartado 4.3 de este módulo.

- ***Inicial***: el proceso está en curso de creación.
- ***Preparado*** (*ready*): el proceso está en condiciones de ejecutar la siguiente instrucción del programa, pero no tiene ningún procesador asignado.
- ***Ejecución*** (*run*): el proceso tiene asignado un procesador y ejecuta instrucciones del programa.
- ***En espera*** (*wait*): el proceso espera algún suceso para poder continuar ejecutándose. Este estado también recibe el nombre de *bloqueo*.
- ***Final***: el proceso está en curso de destrucción.

A medida que se ejecutan, los procesos van cambiando de estado. La figura 4 muestra las transiciones entre los diferentes estados:

Figura 4. Grafo de estados de un proceso



Nos podemos plantear qué motivos provocan las transiciones entre los diferentes estados de los procesos:

Ved también

En el anexo 1 podéis ver el diagrama de estados completo del sistema operativo UNIX System V. Para entender los nuevos estados que aparecen, es necesario que leáis los apartados 4 y 5 de este módulo.

- ***Inicial → preparado***: el SO ha finalizado las tareas asociadas a la creación del proceso (llamada al sistema *crear_proceso*), y éste ya está en disposición de empezar a ejecutarse.

- ***Preparado → ejecución***: el SO ha asignado el procesador a un proceso que estaba en condiciones de ser ejecutado.

- ***Ejecución → preparado***: el SO ha decidido desasignar el procesador al proceso que lo estaba utilizando.

- ***Ejecución → en espera***: el proceso ha pedido algún servicio al SO que no puede ser satisfecho en aquel momento (por ejemplo, una lectura de teclado).

- ***En espera → preparado***: el servicio que el proceso había pedido un instante antes ya ha sido completado. El proceso puede volver a competir por el uso del procesador.

- ***Ejecución → final***: el proceso invoca la llamada al sistema que provoca la finalización (*destruir_proceso*).

Si el SO tiene procesos con diferentes hilos de ejecución, entonces cada hilo tendrá su propio estado. Por tanto, dentro del mismo proceso podremos tener hilos en estado *preparado*, hilos en estado de espera de alguna entrada/salida e hilos en ejecución. Cada hilo de un proceso tiene su propio estado. El estado del proceso será el estado global de sus hilos. En los casos de implementacio-

nes en modo núcleo, puede existir más de un hilo de ejecución en el mismo instante (si se disponen de varios núcleos o CPU), y más de un hilo en estado de bloqueado al mismo tiempo, lo cual no puede darse en los hilos que se implementan en el nivel de usuario.

1.4. El bloque de control del proceso (PCB)

Para gestionar los recursos del sistema y permitir que sean compartidos entre los procesos, el SO agrupa toda la información que necesita conocer de los procesos en una estructura de datos denominada **bloque de control del proceso** (PCB[4]). Todo proceso tiene asignado un PCB durante su ciclo de vida.

[4]PCB es la sigla que corresponde al término inglés *Process Control Block*.

Los datos que acostumbran a estar presentes en los PCB son:

- **Identificadores.** Los identificadores pueden ser de proceso, de usuario, propietario, etc.
- **Estado del proceso**. El PCB toma un valor de los que figuran en la lista de estados posibles de un proceso.
- **Registros del procesador**. El PCB contiene todos los registros del procesador (PC, SP, acumuladores, registros de propósito general, palabra de estado, etc.). Entre estos registros el PC es fundamental, puesto que indica cuál es la siguiente instrucción de código que hay que ejecutar. También es necesario guardar los demás registros porque definen el entorno del proceso.
- **Información de planificación de la CPU**. El PCB recoge datos referidos a la prioridad del proceso y otros datos necesarios para determinar cuándo el proceso volverá a disponer del procesador.
- **Información de gestión de memoria**. Son datos relativos al proceso específicos del sistema de gestión de memoria utilizado por el SO.
- **Información de entrada/salida**. Son datos referentes a los dispositivos asignados al proceso, los ficheros abiertos, las operaciones pendientes, etc.
- **Información de utilización de recursos**. Se puede computar el tiempo de procesador consumido por el proceso, el número de operaciones de entrada/salida efectuadas, etc. De esta forma, se puede evitar que el proceso supere los límites de utilización de recursos que tiene asignados; se puede facturar a los usuarios el uso que llevan a cabo del sistema, y se puede planificar mejor el uso que se hace de la máquina.

Ved también

Podéis informaros acerca del tipo de información que se guardará en el PCB leyendo el apartado 5 de este módulo didáctico.

- **Dominio de protección**. El PCB guarda información de los dominios a los que pertenece el proceso y de los derechos que le corresponden.

Cuando un proceso no se ejecuta (estados *preparado* y *en espera*), el PCB contiene toda la información necesaria para que el SO pueda continuar la ejecución del proceso en el punto donde se ha detenido.

Si el SO ofrece la posibilidad de disponer de procesos con diferentes hilos de ejecución, es necesario guardar una serie de datos referentes a cada hilo. Concretamente, se trata del estado, el PC y el contenido de los registros del procesador.

2. Cambios de contexto

El objetivo de los SO multiprogramados es aprovechar al máximo el procesador intentando que siempre esté ejecutando algún proceso de usuario. En estos SO, el procesador en algún momento debe dejar de ejecutar un programa para pasar a ejecutar otro. Esto comporta la obligación de guardar toda la información asociada al proceso que tenemos en ejecución (el contexto) en su PCB, y restaurar el contexto del proceso que se ejecutará a continuación. La información necesaria del segundo proceso se obtiene del PCB correspondiente. Este conjunto de acciones recibe el nombre de **cambio de contexto**.

Si analizamos el diagrama de estados de un proceso, podemos ver que las transiciones de estado que deberían producir un cambio de contexto son las siguientes:

Ved también

Consultad el grafo de un proceso en el subapartado 1.3 de este módulo.

- ***Ejecución → en espera***: el proceso se ha bloqueado y, por tanto, el procesador queda libre para ejecutar el código correspondiente a otro proceso.

- ***Ejecución → final***: el proceso ha finalizado; el procesador está disponible.

Está claro que en estos dos casos el SO debería provocar un cambio de contexto. Es necesario que se produzcan cambios de contexto en otras situaciones para garantizar una compartición racional del procesador entre los diferentes procesos en ejecución. Pensad qué podría pasar si durante su ciclo de vida un proceso realizara pocas peticiones de entrada/salida, y entre dos de estas peticiones utilizase el procesador durante mucho tiempo.

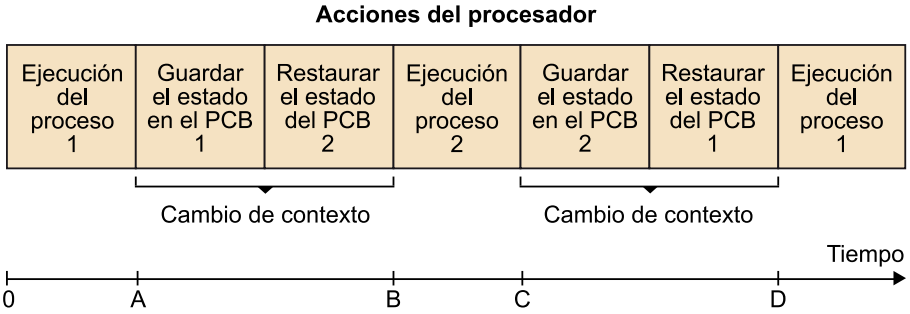
Ved también

Consultad las situaciones que hacen necesario un cambio de contexto en el apartado 5 de este módulo didáctico.

La figura 5 muestra lo que ejecuta un procesador cuando se produce un cambio de contexto a medida que pasa el tiempo[5]:

[(5)]Los tiempos de ejecución de las rutinas de guardar el estado y de restaurar el estado representados en la figura 5 no son reales.

Figura 5. Cambios de proceso en ejecución en un procesador



Ved también

Podéis consultar el coste temporal que supone tener que escoger el proceso que se pasará a ejecutar en el apartado 5 de este módulo didáctico.

Inicialmente, el procesador ejecuta el código correspondiente al proceso 1. En el momento A el SO decide que debe haber un cambio de contexto y, por tanto, guarda en el PCB del proceso 1 toda la información relativa al proceso. A continuación, el SO determina que el nuevo proceso que debe ejecutarse es el proceso 2. Por tanto, restaura el estado del procesador teniendo en cuenta los datos del PCB 2 y el procesador pasa a ejecutar el código del proceso 2. En este esquema no hemos tenido en cuenta el coste de escoger el nuevo proceso que hay que ejecutar.

La rutina encargada del cambio de contexto es la rutina del SO que se ejecuta más veces, pero, directamente, no realiza ninguna tarea propia de los procesos de usuario. Es decir, el tiempo transcurrido entre el momento A y el momento B (y entre el C y el D) es inútil desde el punto de vista del usuario. Estos dos motivos hacen que la implementación del cambio de contexto deba ser lo más eficiente posible, y por esta razón se suele escribir directamente en lenguaje máquina. Por otro lado, para reducir el tiempo de ejecución de esta rutina, algunas arquitecturas proporcionan instrucciones de lenguaje máquina que simplifican su programación.

A medida que los SO son más complejos, la penalización introducida por la ejecución de esta rutina va aumentando. Un procedimiento para disminuir el impacto del coste de los cambios de contexto es utilizar procesos con diferentes hilos de ejecución. Pasar de ejecutar un hilo de un proceso a ejecutar otro hilo del mismo proceso tiene un coste menor que realizar un cambio de contexto entre dos procesos diferentes, puesto que dos hilos del mismo proceso comparten gran parte del contexto (espacio de direcciones, dispositivos de entrada/salida, etc.).

3. El grado de multiprogramación

En cualquier SO multiprogramado podemos hablar del **grado de multiprogramación**, que se define como la cantidad de procesos que se encuentran en el sistema en un momento determinado. Los SO multiprogramados intentan utilizar la multiprogramación para aprovechar el procesador durante más tiempo.

Una medida que nos indica cómo se aprovecha un procesador está determinada por el porcentaje de tiempo que este procesador está ejecutando código relacionado directamente con los programas de los usuarios. Denominaremos a esta medida **grado de aprovechamiento de la CPU**:

a) En los SO monoprogramados, cuando el programa en ejecución debía esperar la finalización de una entrada/salida, el procesador permanecía sin realizar ninguna tarea útil para el usuario. Esto podía provocar un grado de aprovechamiento bajo del procesador.

b) En cambio, en los SO multiprogramados podemos tener diferentes procesos activos. Supongamos que tenemos dos procesos activos. Cuando un proceso deba abandonar el procesador porque tenga que esperar una entrada/salida, el SO podrá asignar el procesador al otro proceso. De esta forma puede sacar mayor partido del procesador, puesto que lo utilizará para ejecutar el código correspondiente a dos procesos. Además, desde el punto de vista de los usuarios, el tiempo de retorno de los procesos (el tiempo que transcurre desde que el usuario indica que quiere ejecutar el programa hasta que finaliza la ejecución) puede ser prácticamente el mismo que en el sistema monoprogramado.

Ved también

Consultad la representación de un proceso en un sistema monoprogramado en el subapartado 1.1 de este módulo didáctico.

c) A medida que tengamos más procesos activos (cuando aumentemos el grado de multiprogramación), el SO tendrá más procesos por escoger y conseguirá que el procesador esté ocupado durante una fracción de tiempo superior. De todas formas, este grado de aprovechamiento nunca llegará al 100%, porque el procesador deberá ejecutar rutinas propias del SO, como la de cambio de contexto. Desde el punto de vista de los usuarios, el tiempo de retorno se empezará a ver afectado, ya que, probablemente, los procesos estarán cierto tiempo en el estado *preparado* esperando recibir la asignación del procesador, a causa de la competencia de los otros procesos.

d) Si incrementamos aún más el grado de multiprogramación, los usuarios verán cómo aumenta cada vez más el tiempo de retorno de sus programas, pero ya no mejoraremos el grado de aprovechamiento del procesador. Además, este grado de aprovechamiento finalmente puede caer en picado, pero para entenderlo hay que conocer el concepto de *paginación bajo demanda*. Para hacerse una idea del problema hay que pensar que todos los procesos en ejecución ocupan recursos del sistema (por ejemplo, memoria). Si tenemos muchos procesos en ejecución, el SO tendrá cada vez más dificultades para encontrar recursos disponibles y, como consecuencia de ello, pasará más tiempo ejecutando el código interno del SO. Esta situación se denomina *thrashing*[6], y provoca una fuerte bajada del rendimiento del sistema.

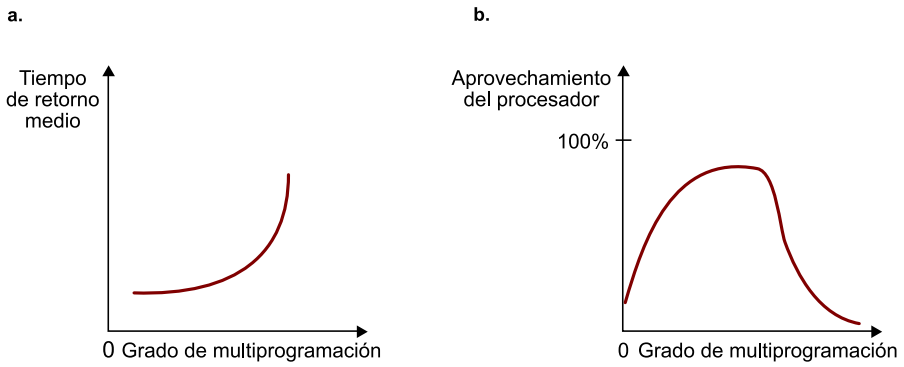
Ved también

Consultad la paginación bajo demanda en el subapartado 2.1 del módulo "La memoria virtual" de esta asignatura.

[6]La traducción literal de *thrashing* es 'hiperpaginación'.

En la figura 6 podemos ver la relación que existe entre el grado de multiprogramación y el tiempo de retorno medio de los programas en ejecución (gráfico **a**), así como la relación entre el grado de multiprogramación y el porcentaje de aprovechamiento del procesador (gráfico **b**):

Figura 6. Relación entre el grado de multiprogramación y: (a) el tiempo de retorno medio; y (b) el aprovechamiento del procesador



4. Los planificadores del procesador

La **planificación del procesador** es el conjunto de políticas y mecanismos que implementa el SO para decidir qué proceso debe utilizar el procesador en cada instante.

Los planificadores deben intentar conseguir los siguientes objetivos:

- **Disponibilidad**: un proceso no puede esperar indefinidamente para disponer del procesador.
- **Reparto**: hay que intentar maximizar el número de procesos que recibe el procesador por unidad de tiempo.
- **Equilibrio**: los planificadores deben intentar favorecer los procesos que utilizan recursos poco utilizados.
- **Aprovechamiento**: hay que sacar el máximo partido posible del procesador intentando que siempre haya procesos disponibles para ser ejecutados.
- **Eficiencia**: el procesador no puede estar demasiado tiempo ejecutando código del planificador, ya que durante este lapso no ejecuta código de usuario. De todas formas, dedicar más tiempo a la ejecución del planificador nos puede ayudar a sacar más partido del procesador, puesto que podremos escoger los procesos que permitan obtener el máximo rendimiento del computador.
- **Interactividad**: hay que favorecer los procesos interactivos porque hay usuarios "impacientes" tras estos procesos y porque el tiempo de usuario es más caro que el tiempo de máquina.

Estos objetivos no se pueden alcanzar simultáneamente, puesto que algunos están contrapuestos. Por tanto, la gestión del planificador constituye un problema complejo. Para alcanzar al máximo estos objetivos, la planificación se divide en diferentes niveles (a largo, medio y corto plazo). Veremos que cada uno presenta problemas diferentes, y que son más sencillos de resolver de forma separada.

4.1. Planificadores a largo plazo

Para entender la función de estos planificadores pensaremos en sistemas de trabajo no interactivos (por lotes, o *batch*). Los programas ejecutados en estos sistemas acostumbran a ser de baja prioridad y realizan un uso intensivo de los recursos del sistema (tiempo de procesador, memoria, operaciones de entrada/salida, etc.).

Cuando un usuario pretende ejecutar un programa en uno de estos sistemas, lo coloca en la cola de programas que están esperando para ejecutarse (cola por lotes). También hace una estimación de la cantidad de recursos que consumirá el programa (el tiempo máximo de CPU que se puede dedicar a su ejecución, la cantidad máxima de memoria asignable al proceso, el uso que hará de los dispositivos de entrada/salida, etc.).

La función del planificador a largo plazo es escoger la tarea siguiente de la cola por lotes que se podrá empezar a ejecutar, teniendo en cuenta las condiciones actuales de carga del sistema y las estimaciones de uso de los recursos indicadas por el usuario, para obtener el máximo rendimiento posible del sistema.

Podemos decir que el planificador a largo plazo es el encargado de controlar el grado de multiprogramación del sistema, es decir, la cantidad de procesos que se ejecutan a la vez en el sistema.

Planificación del proceso siguiente

Si los procesos que hay en ejecución en la máquina utilizan principalmente el procesador, el planificador puede decidir que el proceso que se empezará a ejecutar a continuación sea un proceso que utilice sobre todo la entrada/salida.

La frecuencia de llamada de este planificador es baja, sólo se invoca cuando un proceso finaliza su ejecución y cuando la utilización del procesador es pobre. Por tanto, este planificador puede implementar algoritmos de selección complejos que tengan en cuenta las estimaciones de uso de los recursos, las estadísticas del comportamiento de los programas en sus últimas ejecuciones, etc. Si observamos el diagrama de transiciones entre los estados de los procesos, este planificador es el encargado de hacer entrar nuevos procesos al sistema; permite el paso de procesos desde el estado *inicial* al estado *preparado*.

Ved también

Podéis ver las transiciones entre los estados en la figura 4, que representa el grafo de estados de un proceso, en el subapartado 1.3 de este módulo didáctico.

Este tipo de planificadores son prácticamente inexistentes en los sistemas orientados al trabajo interactivo. Cuando uno de estos sistemas recibe la petición de crear un nuevo proceso, lo pone directamente en el estado *preparado*. En este caso, la única limitación para crear procesos nuevos está determinada por el tamaño de la tabla del PCB.

4.2. Planificadores a corto plazo

El planificador a corto plazo es el encargado de escoger el siguiente proceso que pasará al estado *ejecución* entre los están en el estado *preparado*. Este planificador es invocado cada vez que se produce un cambio de contexto.

Si analizamos el grafo de estados de los procesos, vemos claramente que este planificador sería invocado cuando algún proceso experimentase una de las siguientes transiciones: ***preparado → final*** o ***ejecución → en espera***.

A diferencia del planificador a largo plazo, éste es invocado con mucha frecuencia (varias veces por segundo); por tanto, conviene que sea rápido al elegir el proceso siguiente.

Tiempo de decisión

Si el planificador de corto plazo tarda, por ejemplo, un milisegundo en decidir cuál es el proceso siguiente que debe utilizar la CPU, y este planificador se invoca una vez cada 10 ms, tendremos que el 10% del tiempo del procesador está ocupado por el planificador de bajo nivel.

4.3. Planificadores a medio plazo

En algunos SO existe un nivel intermedio de planificación. Básicamente se trata de sistemas de tiempo compartido en los que la carga del sistema puede experimentar variaciones importantes en poco tiempo, hecho que puede poner en peligro el rendimiento global del sistema[7]. En este caso, puede ser útil eliminar temporalmente algún proceso de la memoria y colocarlo en un dispositivo secundario, como puede ser el disco. De esta manera tenemos menos procesos compitiendo por utilizar la CPU y damos más oportunidades al resto de los procesos. Más adelante, cuando las condiciones del sistema sean más favorables, este proceso volverá a la memoria y continuará su ejecución en el punto en el que se detuvo.

Ved también

Consultad las causas que provocan una caída súbita del rendimiento del sistema en el apartado 3 de este módulo didáctico.

(7)En inglés, *thrashing*.

Un criterio que puede justificar la elección de un proceso para eliminarlo temporalmente de la memoria física sería tener en cuenta si se está ejecutando de forma interactiva o por lotes[8]. Puesto que los procesos interactivos son más sensibles al tiempo de reposo que los procesos por lotes, estos últimos serían candidatos a ser escogidos por el planificador de medio plazo.

(8)En inglés, *batch*.

Otro criterio es considerar el tiempo que hace que un proceso está en el estado *en espera*. Por ejemplo, si un usuario abandona su terminal con un intérprete de órdenes en marcha, el proceso correspondiente estará en el estado *en espera*, puesto que habrá realizado una petición de lectura al SO para obtener la siguiente instrucción, que no puede ser satisfecha (no hay ningún usuario que teclee la instrucción). Es presumible que este proceso continuará en el estado *en espera* durante cierto tiempo. Por tanto, los procesos que han estado bastante tiempo en dicho estado también son candidatos a ser escogidos por el planificador a medio plazo.

El hecho de detener la ejecución de un proceso y llevarlo a un dispositivo secundario recibe el nombre de *intercambio en el disco*[9], y la acción inversa es el *intercambio en la memoria física*[10].

[9] En inglés, *swapping out*.

[10] En inglés, *swapping in*.

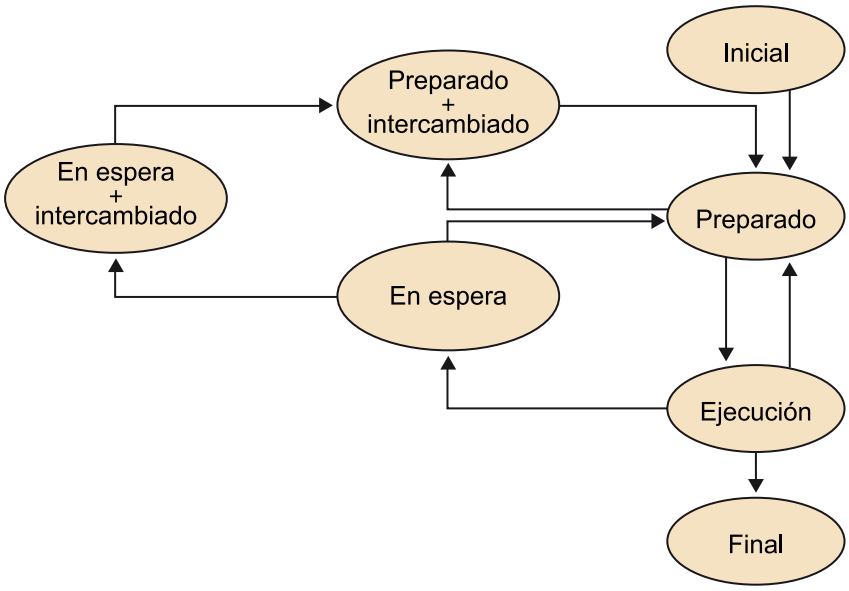
Podemos decir que el planificador a medio plazo también controla el grado de multiprogramación del computador, puesto que retira temporalmente ciertos procesos de la competencia del procesador.

Este planificador introduce un par de estados nuevos en el diagrama de estados de los procesos:

- ***En espera + intercambiado***: un proceso que espera algún suceso y ha sido intercambiado en el disco.
- ***Preparado + intercambiado***: un proceso que está preparado para ejecutarse, pero que no está en la memoria física. Las transiciones que llegan en este estado se originan en el estado *preparado* y en el estado *en espera + intercambiado*.

La figura 7 muestra cómo queda el **grafo de estados de los procesos** una vez añadidos estos dos nuevos estados:

Figura 7. Grafo de estado de un proceso considerando el intercambio en el disco



Las nuevas transiciones posibles entre estados son las siguientes:

- ***En espera → en espera + intercambiado***: un proceso que estaba bloqueado ha sido intercambiadso en el disco.

- ***En espera + intercambiado → preparado + intercambiado***: el servicio que esperaba el proceso ha finalizado. En el PCB del proceso (que continúa en la memoria física) se refleja este cambio de estado.

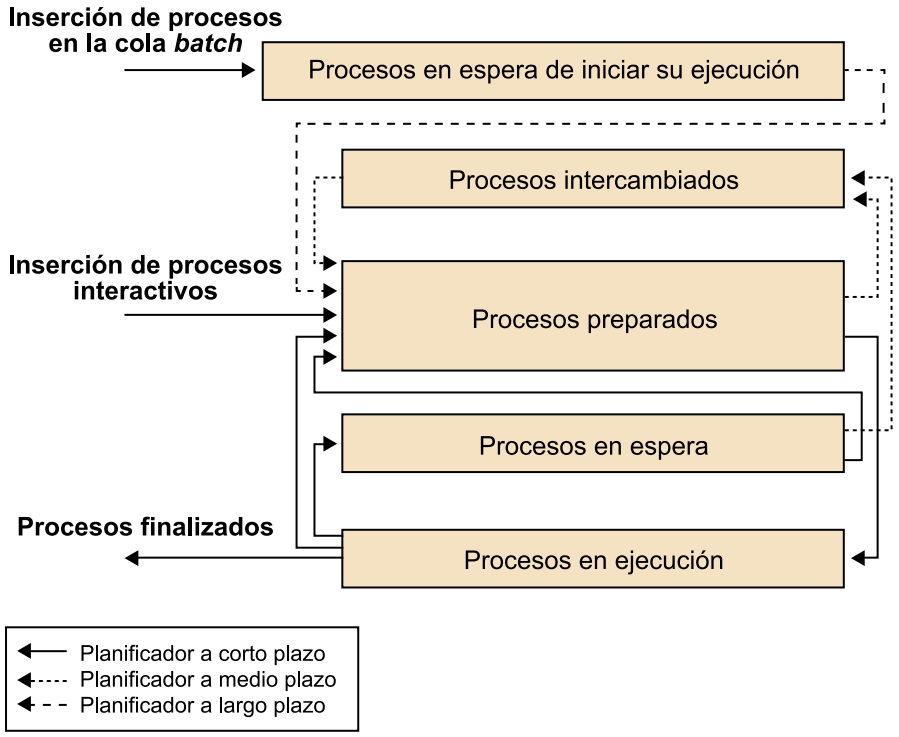
- ***Preparado + intercambiado → preparado***: un proceso que había sido intercambiado es devuelto a la memoria. El proceso está en condiciones de continuar su ejecución.

- ***Preparado → preparado + intercambiado***: un proceso que estaba preparado es intercambiado en el disco para reducir la cantidad de procesos que compiten por el procesador.

4.4. Interacción entre los tres niveles de planificación

La figura 8 muestra cómo interaccionan los diferentes planificadores. Las líneas gruesas representan la inserción y la eliminación de procesos del sistema:

Figura 8. Interacción entre los diferentes niveles de planificación



a) La entrada de los procesos por lotes es controlada por el planificador de largo plazo, mientras que los procesos interactivos entran libremente.

b) El planificador a corto plazo selecciona los procesos que pueden usar el procesador de los que se encuentran en la cola de procesos preparados. Además, retira los que han utilizado el procesador y los lleva a la cola de procesos preparados o a la cola de procesos en espera.

c) Finalmente, el planificador de medio plazo selecciona procesos que no están en ejecución y los lleva al disco si las necesidades del sistema lo exigen.

5. Algoritmos de planificación del procesador

Un algoritmo de planificación debe escoger un proceso entre todos los que se encuentran en una cola teniendo en cuenta una serie de criterios.

En este apartado definiremos los parámetros que permiten evaluar los algoritmos de planificación y presentaremos los algoritmos más típicos para cada nivel de planificación, comentando los puntos fuertes y los puntos débiles de cada uno.

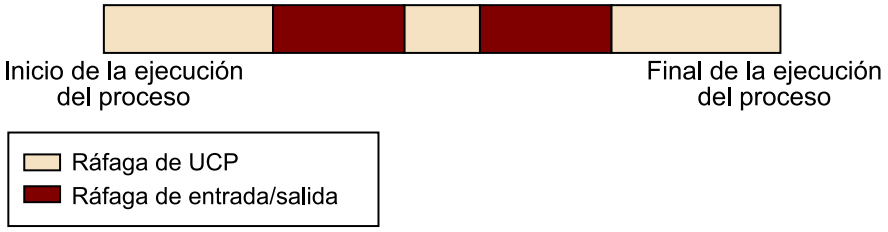
5.1. Ráfagas de CPU y de entrada/salida

Los SO multiprogramados aprovechan una propiedad presente en todos los programas: su ejecución se puede descomponer en una sucesión de pares de intervalos de tiempo. En el primer intervalo se utiliza el procesador (**ráfaga de CPU**), y en el segundo el programa espera la finalización de una entrada/salida (**ráfaga de entrada/salida**).

La ejecución de todo programa acaba con un intervalo en el que se utiliza el procesador y se invoca la llamada al sistema que provoca la finalización del proceso.

Podemos ver la representación esquemática de la separación en ráfagas de la ejecución de un proceso en la figura 9.

Figura 9. Descomposición en ráfagas del tiempo de ejecución



Los SO multiprogramados tienen en cuenta esta característica de los programas e intentan mantener el procesador ocupado durante el máximo tiempo posible haciendo que ejecute el código correspondiente a procesos diferentes. Esto también permite aprovechar más los dispositivos de entrada/salida.

Se han medido estas ráfagas de CPU y se ha observado que la mayoría son cortas y que sólo una minoría son bastante largas.

5.2. Evaluación de los algoritmos de planificación

El rendimiento de un planificador se puede evaluar estudiando los siguientes parámetros:

- **Uso del procesador**: indica el porcentaje de tiempo que el procesador ejecuta código directamente relacionado con algún proceso de usuario. Todo el tiempo que el procesador pasa ejecutando la rutina de cambio de contexto o el algoritmo de planificación es tiempo desaprovechado para ejecutar procesos de usuario.

- **Productividad**[11]: mide el número de procesos que finalizan su ejecución por unidad de tiempo. Esta medida varía sustancialmente al pasar de sistemas destinados al trabajo interactivo (procesos de duración corta) a sistemas destinados a cálculos científicos (procesos de duración larga).

(11) En inglés, *throughput*.

- **Tiempo de retorno**[12]: se define como el tiempo que transcurre entre el momento en que el usuario indica que pretende ejecutar un programa y la finalización de la ejecución de dicho programa. Desde el punto de vista del usuario, éste es el tiempo que ha tardado el programa en ejecutarse.

(12) En inglés, *turnaround time*.

- **Tiempo de espera**: corresponde al tiempo que un proceso ha pasado al estado *preparado*. Este tiempo está determinado por los algoritmos de planificación utilizados y por el grado de multiprogramación.

- **Tiempo de respuesta**: mide el tiempo que transcurre desde que se realiza una petición a un dispositivo de entrada/salida hasta que se obtiene la respuesta.

Dado uno de estos parámetros, nos interesa optimizarlo. Queremos maximizar el uso del procesador y su productividad; también convendría minimizar el tiempo de retorno, el tiempo de espera y el tiempo de respuesta. Normalmente es más interesante optimizar el valor medio de estos criterios y, en algunos casos, optimizar algún valor máximo o mínimo. Por ejemplo, para ofrecer un buen servicio a los usuarios se podría minimizar el tiempo de espera máximo.

En los SO orientados al trabajo interactivo es interesante ofrecer un tiempo de respuesta estable, de manera que el usuario pueda predecir el tiempo de respuesta de la máquina y éste no sufra variaciones importantes. Por tanto, en estos sistemas también se intenta minimizar la varianza del tiempo de respuesta.

5.3. Algoritmos de planificación a corto plazo

Los algoritmos de planificación a corto plazo asignan el procesador a uno de los procesos que está en el estado *preparado*.

Dispatcher

El planificador a corto plazo también recibe el nombre de *dispatcher*.

En función de la manera como finaliza esta asignación, podemos clasificar los tipos de algoritmo de planificación a corto plazo en los dos grupos siguientes:

1) No apropiativos: algoritmos en los que la asignación finaliza por voluntad del proceso, es decir, el proceso utiliza el procesador hasta que acaba la ráfaga de CPU que está ejecutando, independientemente de su duración y del resto de los procesos que hay en el computador.

2) Apropiativos: el algoritmo de planificación puede desasignar el procesador antes que finalice la ráfaga de CPU. En este caso tenemos dos nuevas posibilidades:

a) Apropiación inmediata: el algoritmo de planificación se invoca siempre que un proceso se desbloquea; por tanto, se producirá un cambio de contexto si el proceso que se desbloquea tiene más derecho a ejecutarse que el proceso que en aquel momento está en el estado *ejecución*.

b) Apropiación diferida: el algoritmo de planificación se invoca cuando finaliza la ráfaga de CPU del proceso y también a intervalos de tiempo regulares, independientemente de la duración de las ráfagas de CPU de los procesos. En estas invocaciones adicionales al planificador, éste puede volver a decidir qué proceso debería utilizar el procesador.

Podemos ver que los algoritmos no apropiativos otorgan demasiada libertad a los procesos, puesto que no dejan libre al procesador hasta que la ráfaga de CPU finaliza. Esto puede provocar problemas si las ráfagas de CPU son muy largas, o si existen procesos que necesitan el procesador urgentemente.

Por otro lado, los algoritmos de apropiación diferida dan la oportunidad de cambiar el proceso en ejecución periódicamente.

A continuación veremos los algoritmos de planificación típicos que utiliza el planificador a corto plazo del SO.

5.3.1. De orden de llegada (*first come, first served*)

El algoritmo de orden de llegada ordena los procesos en la cola de procesos preparados por orden de llegada. Pasará al estado *ejecución* el proceso que haya permanecido más tiempo en el estado *preparado*.

Este algoritmo es no apropiativo, es decir, sólo se invoca cuando finaliza la ráfaga de CPU del proceso que está utilizando el procesador. Esto comporta que el algoritmo pueda ofrecer rendimientos pobres, muy variables y que penalice los trabajos pequeños.

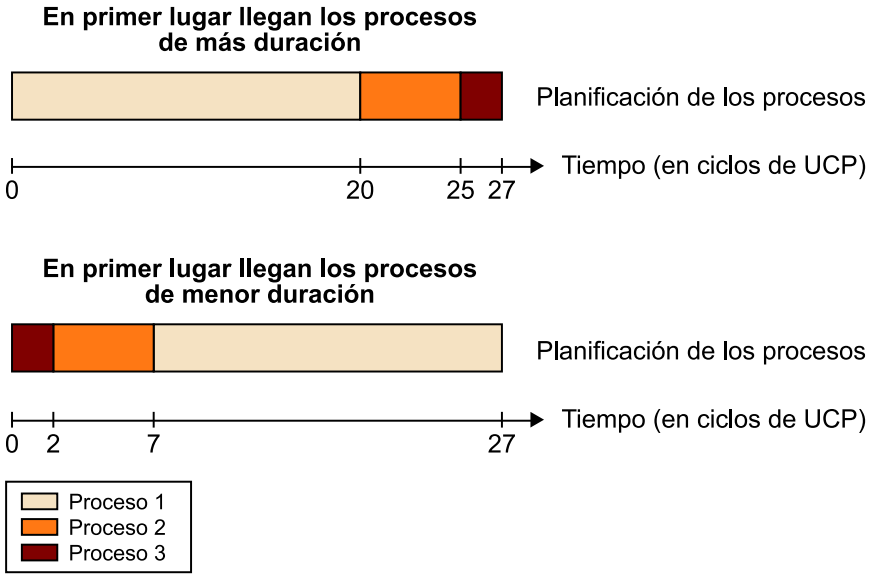
Planificación de tres procesos según el algoritmo de orden de llegada

Considerad que tenemos tres procesos compuestos por una única ráfaga de CPU, con una duración de 20, 5 y 2 ciclos, respectivamente. A continuación veremos cómo sería la planificación de estos trabajos partiendo de los dos casos siguientes:

- En primer lugar han llegado los trabajos de mayor duración.
- En primer lugar han llegado los trabajos de menor duración.

La figura 10 ilustra la planificación de los procesos en ambos casos:

Figura 10. Planificación de los procesos según el algoritmo de orden de llegada



Si comparamos los tiempos de retorno medio en ambos casos observamos una diferencia notable. En el primer caso, los tiempos de retorno son de 20, 25 y 27 ciclos respectivamente; por tanto, el tiempo de retorno medio es de 24 ciclos. En el segundo caso, los tiempos de retorno son de 27, 7 y 2 ciclos respectivamente; esto da un tiempo de retorno medio de 12 ciclos. Podemos apreciar que el rendimiento es muy variable y que puede penalizar los trabajos de corta duración.

Nota

En ninguno de los ejemplos de este módulo tomaremos en consideración el coste temporal de la ejecución de los algoritmos de planificación y de la rutina de cambio de contexto.

El algoritmo que acabamos de describir es muy desaconsejable en sistemas de tiempo compartido, en los que es importante que los procesos dispongan del procesador a intervalos regulares para que los usuarios tengan la sensación de disponer de la máquina en exclusiva. Este algoritmo puede provocar una diferencia de tiempo de retorno muy grande en función de los tipos de procesos que se encuentran en la máquina. La ventaja del algoritmo es que resulta muy sencillo de implementar.

5.3.2. De menor tiempo primero (*shortest job first*)

El algoritmo de menor tiempo primero intenta evitar la penalización que podía imponer el algoritmo de orden de llegada a los trabajos cortos. El algoritmo asocia a cada proceso la duración de la ráfaga siguiente de CPU. El procesador

se asigna al proceso de la cola de estados preparados que tiene la ráfaga de CPU de menor duración. En caso de empate entre dos procesos, se aplica el algoritmo de orden de llegada.

El problema de este mecanismo es la dificultad de saber cuál será la longitud de la ráfaga siguiente de CPU de un proceso. Normalmente se utiliza una estimación que tiene en cuenta la duración de las ráfagas anteriores de CPU de este proceso. Una **estimación posible de la duración de la próxima ráfaga de CPU** (e_{n+1}) es la siguiente:

$$e_{n+1} = \alpha \cdot r_n + (1 - \alpha) \cdot e_n,$$

donde r_n es la duración real de la última ráfaga de CPU, e_n es la duración estimada de la última ráfaga de CPU, y α es un factor comprendido entre 0 y 1.

Cuando el procesador es asignado a un proceso, la asignación dura hasta que la ráfaga de CPU finaliza. Por tanto, este algoritmo es no apropiativo.

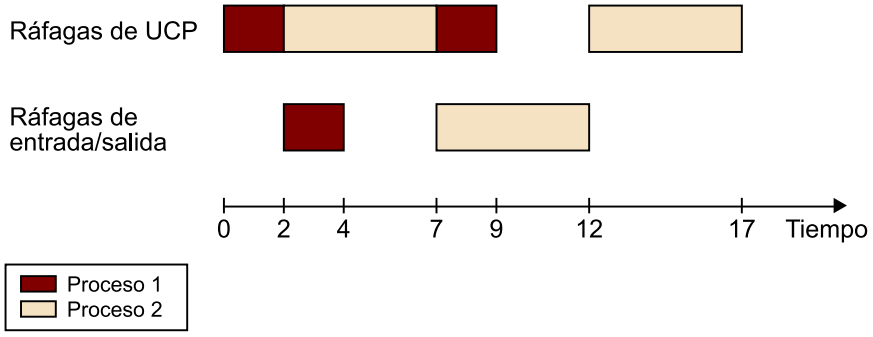
Ved también

Veremos la versión apropiativa del algoritmo de menor tiempo primero en el subapartado 5.3.3 de este módulo didáctico.

Planificación de dos procesos según el algoritmo de menor tiempo primero

Supongamos, por ejemplo, que debemos planificar dos procesos. El primero consta de una ráfaga de CPU, una ráfaga de E/S y una ráfaga final de CPU, todas de 2 ciclos. El segundo proceso también consta de tres ráfagas, pero de 5 ciclos cada una. La figura 11 muestra cómo se planificaría cada proceso según este algoritmo. Supondremos que nuestro mecanismo de predicción de las duraciones de las ráfagas de CPU es perfecto, y que inicialmente, ambos procesos son los únicos que están en la cola de procesos preparados.

Figura 11. Planificación de los procesos según el algoritmo de menor tiempo primero



Este algoritmo ofrece tiempos de retorno mejores que el de orden de llegada, pero tampoco es aconsejable para sistemas de tiempo compartido, ya que no es apropiativo. Además, la predicción de la duración de la ráfaga siguiente no es fácil.

Observemos otro problema en este algoritmo: puede producir inanición. Si la estimación de la duración de la ráfaga siguiente de CPU de un proceso da un resultado muy grande, el proceso puede permanecer indefinidamente a la cola de procesos preparados; esto sucederá en el caso de que a esta cola lleguen continuamente procesos con ráfagas de CPU más cortas.

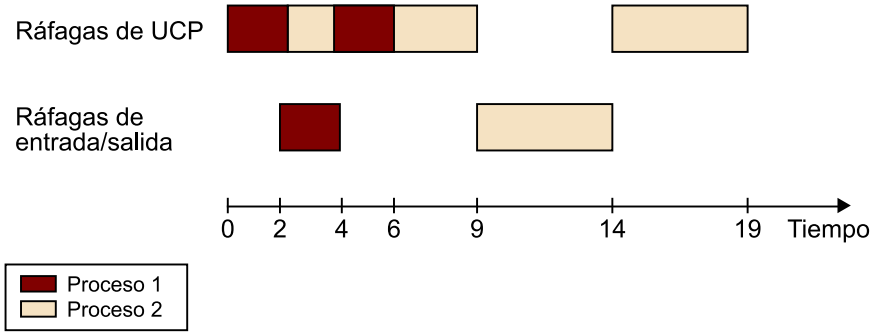
5.3.3. De menor tiempo pendiente (*shortest remaing time*)

El algoritmo de menor tiempo pendiente sería la versión apropiativa del algoritmo de menor tiempo primero. La apropiación aparece porque el planificador es invocado cada vez que hay que insertar un trabajo en la cola de procesos preparados. Entonces, si la duración estimada de la ráfaga siguiente del proceso que pasamos a *preparado* es inferior a la duración de la ráfaga del proceso que tenemos en ejecución, se producirá un cambio de contexto. Por tanto, este algoritmo aplica la apropiación inmediata.

Planificación de dos procesos según el algoritmo de menor tiempopendiente

Volvamos a suponer los dos procesos que teníamos en el ejemplo que hemos visto al explicar el algoritmo de menor tiempo primero. En este caso, el planificador desbanca del procesador el proceso 2 en el instante 4. Esto sucede porque el proceso 1 ha pasado al estado *preparado*, y su ráfaga siguiente de CPU es de 2 ciclos. Por otro lado, el tiempo pendiente de la ráfaga del proceso 2 es de 3 ciclos. Por tanto, el planificador cambia la asignación del procesador.

Figura 12. Planificación de los procesos según el algoritmo de menor tiempo pendiente



Este algoritmo conserva el problema de la estimación de la longitud de las ráfagas. Una implementación de este algoritmo también debería valorar el coste del cambio de contexto antes de decidir apropiarse el procesador, ya que tal vez se aprovechará más el procesador dejando acabar la ráfaga actual que provocando un cambio de contexto para ejecutar otro proceso. Este algoritmo también presenta el problema de la inanición.

5.3.4. Prioritario

El algoritmo prioritario asocia una prioridad a cada proceso, y el procesador se asigna al proceso de la cola de procesos preparados que tiene la prioridad más alta. En caso de empate, se utiliza el algoritmo de orden de llegada para decidir qué proceso se asigna al procesador. El algoritmo de menor tiempo primero se puede considerar como un algoritmo prioritario, en el que la prioridad es la inversa del tamaño de la ráfaga siguiente.

Las prioridades se pueden definir interna o externamente:

- Las **prioridades definidas internamente** tienen en cuenta características del proceso mensurables por parte del SO; por ejemplo, el tiempo de pro-

cesador que se ha consumido, la relación entre las duraciones de la ráfagas de CPU y las de entrada/salida, etc.

- Las **prioridades definidas externamente** tienen en consideración aspectos ajenos al SO, como el tipo de usuario responsable del proceso.

El algoritmo prioritario también se puede aplicar no apropiativamente o de manera apropiativa inmediata:

a) En la **versión no apropiativa**, un proceso que utiliza el procesador no lo abandona hasta que se acaba la ráfaga de CPU, sin tener en cuenta si entran procesos más prioritarios en la cola de procesos preparados.

b) En la **versión apropiativa inmediata** se comprueba si la prioridad de los procesos que deben insertarse en la cola de procesos preparados es más alta que la prioridad del proceso en ejecución. En caso afirmativo, se produce un cambio de contexto aunque no haya finalizado la ráfaga de CPU.

Cualquier algoritmo de planificación prioritario tiene el problema de la inanición. Un proceso de prioridad baja puede permanecer indefinidamente en la cola de estados preparados si no dejan de llegar procesos más prioritarios.

Una posible solución para este problema es el envejecimiento[13]. Consiste en incrementar periódicamente la prioridad de los procesos que hace tiempo se encuentran en la cola de preparados. De esta forma llegará un momento en que todo proceso será el más prioritario de los que están en la cola de estados preparados.

(13) En inglés, *aging*.

5.3.5. De reparto de tiempo (*round robin*)

El algoritmo de reparto de tiempo está pensado para sistemas de tiempo compartido. En estos sistemas es importante asignar el procesador a diferentes procesos de manera regular, independientemente de la duración de sus ráfagas de CPU. De esta forma, los usuarios pueden trabajar de manera más agradable. Por tanto, hay que garantizar que se pueda producir un cambio de contexto independientemente de la duración de las ráfagas de CPU de los procesos.

Este algoritmo está basado en el algoritmo de orden de llegada, pero aplicando la apropiación diferida. Para poder hacerlo, el algoritmo define una unidad de tiempo llamada *cuota*, y garantiza que el algoritmo de planificación se activará, como mínimo, una vez cada cuota de tiempo.

El algoritmo asigna el procesador a los procesos de la cola de procesos preparados siguiendo el orden de llegada, y la asignación dura hasta que se produce uno de los dos sucesos siguientes:

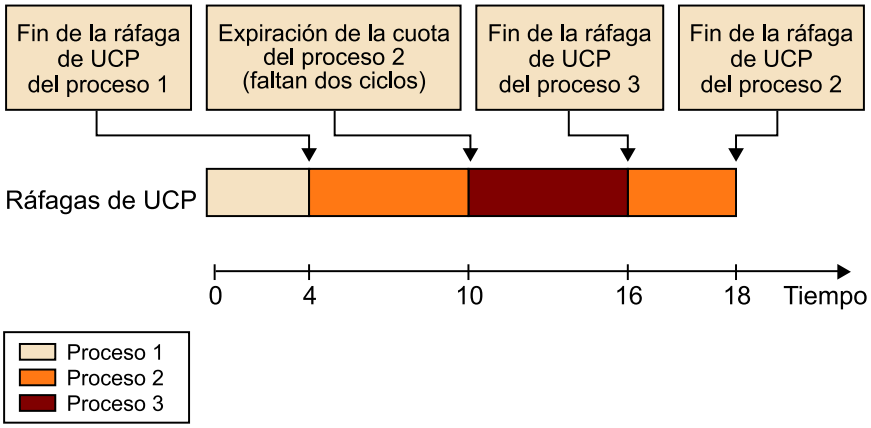
1) El proceso finaliza la ráfaga de CPU. En este primer caso, el proceso ha abandonado voluntariamente la CPU, ya que espera el resultado de una operación de entrada/salida.

2) El proceso ya ha estado un cuanto de tiempo ejecutando la ráfaga de CPU. En este caso, la ejecución del planificador es provocada desde la rutina de atención al temporizador. El algoritmo desasigna el procesador del proceso en ejecución (se apropia el procesador), coloca el proceso en ejecución al final de la cola de procesos preparados y selecciona el primer proceso de la cola de preparados para que utilice el procesador.

Planificación de tres procesos según el algoritmo de reparto de tiempo

Supongamos, por ejemplo, que tenemos 3 procesos, cada uno con una ráfaga única de CPU de 4, 8 y 6 ciclos de duración, respectivamente; que el tamaño del cuanto es de 6 ciclos, y que tenemos los tres procesos en la cola de preparados en el orden indicado. La figura 13 muestra el cronograma correspondiente a la ejecución de estos tres procesos según el algoritmo de reparto de tiempo.

Figura 13. Planificación de los procesos según el algoritmo de reparto de tiempo



Inicialmente el algoritmo selecciona el proceso 1 porque ha sido el primero en llegar. Puesto que su ráfaga es inferior a 6 ciclos, antes de expirar la cuota, el proceso 1 se bloquea. Por tanto, se produce un cambio de contexto y se selecciona el proceso 2. Puesto que la ráfaga de este proceso es más larga que la cuota, el temporizador provoca un cambio de contexto antes de finalizar la ráfaga, y el algoritmo de planificación selecciona el proceso 3. Puesto que la ráfaga del proceso 3 tiene el mismo tamaño que la cuota, la ráfaga se ejecuta entera. Finalmente, el proceso 2 ejecuta los 2 ciclos que le quedaban pendientes de su ráfaga.

Un factor crítico de este algoritmo es la selección de la duración de la cuota de tiempo. Valores grandes de la cuota pueden convertir este algoritmo en el algoritmo de orden de llegada puro, mientras que los valores pequeños pueden hacer que el rendimiento del sistema se vea afectado por el coste de ejecutar la rutina de cambio de contexto. Pensemos que si la cuota tiene un valor pequeño, la mayoría de las ráfagas de CPU serán más largas que el tamaño de la cuota y se verán interrumpidas por la ejecución de la rutina del temporizador y el consiguiente cambio de contexto.

Duración del cuanto de tiempo

Supongamos, por ejemplo, que todas las ráfagas de CPU son de 5 ms.

a) Para un tamaño de cuota igual a 10 ms, el algoritmo permitirá la ejecución completa de las ráfagas antes de seleccionar otro proceso.

b) Para un tamaño de cuota igual a 3 ms, para ejecutar una ráfaga entera será necesario asignar dos veces el procesador al proceso. La primera asignación podría ejecutar los tres primeros milisegundos de la ráfaga, y la segunda ejecución ejecutaría los dos milisegundos restantes. A diferencia del caso anterior, a la mitad de la ráfaga se ha producido un cambio de contexto.

c) Para un tamaño de cuota igual a 1 ms, tendríamos cuatro cambios de contexto durante la ejecución de la racha.

El **valor óptimo de cuanto** es un término medio entre un valor lo bastante pequeño para que todos los procesos vayan recibiendo el procesador de forma regular, y lo bastante grande para evitar que las ráfagas de CPU se vean interrumpidas por la finalización de la cuota. Además, debe ser largo comparado con el tiempo de ejecución de la rutina de cambio de contexto. Pensemos que, como mínimo, se dará una invocación a la rutina de cambio de contexto al finalizar cada cuota de unidades de tiempo. Por tanto, la fracción de tiempo provechosa para los programas de usuario está limitada por el cociente [cuota / (cuota + tiempo de ejecución del cambio de contexto)].

Aprovechamiento de los programas

Si el tiempo de ejecución de la rutina de cambio de contexto fuera el 10% del tiempo de cuota (q), calculamos los tiempos en que podríamos ejecutar código útil de la manera siguiente:

Tiempo de aprovechamiento = q / (q + 0,1 · q).

El resultado es aproximadamente el 91% del tiempo.

5.3.6. Colas multinivel (*multilevel queues*)

En la mayoría de los SO encontramos procesos de características muy diferentes, como pueden ser:

a) Procesos interactivos, que necesitan un tiempo de respuesta pequeño.

b) Procesos correspondientes a tareas propias del SO. Estos procesos deben ser ejecutados de manera muy prioritaria.

c) Procesos por lotes (*batch*), para los que el tiempo de respuesta no es un factor crítico.

El algoritmo de planificación de colas multinivel divide la cola de procesos preparados en diferentes colas exclusivas y asocia cada proceso a una de estas colas durante todo su ciclo de vida. En cada cola se aplica una política de planificación diferente. Por ejemplo, si tenemos una cola para los procesos interactivos y otra para los procesos por lotes, parece razonable utilizar una política de reparto de tiempo para la cola de procesos interactivos y una política de

orden de llegada para los procesos por lotes. De esta forma, los procesos interactivos recibirán regularmente el procesador y en los procesos por lotes minimizaremos la penalización impuesta por la ejecución de la rutina de cambio de contexto.

Si tenemos varias colas, se hace necesario aplicar algún tipo de planificación entre éstas. Podemos asignar, por ejemplo, una prioridad a cada cola y no permitir la ejecución de un proceso de una cola si hay algún proceso preparado en alguna cola más prioritaria. Otra posibilidad consiste en repartir el tiempo del procesador entre las diferentes colas: por ejemplo, el 90% del tiempo para la correspondiente a procesos interactivos y el 10% para la de procesos por lotes o, incluso, se pueden variar dinámicamente estos porcentajes en función de la carga de ambas colas.

5.3.7. Colas multinivel realimentadas (*multilevel feedback queues*)

Podemos aplicar un perfeccionamiento en el algoritmo de las colas multinivel que acabamos de explicar. Se trata de permitir que los procesos cambien de cola a medida que pasa el tiempo en función de su comportamiento. Si un proceso tiene ráfagas de CPU muy largas, podemos bajar su prioridad, y podemos pasar los procesos que han estado mucho tiempo en una cola de prioridad baja a una cola de prioridad mayor para evitar la inanición.

En este tipo de algoritmos hay que definir una serie de aspectos: la cantidad de colas, el tipo de planificación de cada cola, la planificación entre colas, los criterios para decidir a qué cola hay que insertar inicialmente cada proceso y los criterios para determinar cuándo debe pasarse un proceso a una cola menos prioritaria y cuándo conviene pasar un proceso a una cola más prioritaria.

Ved también

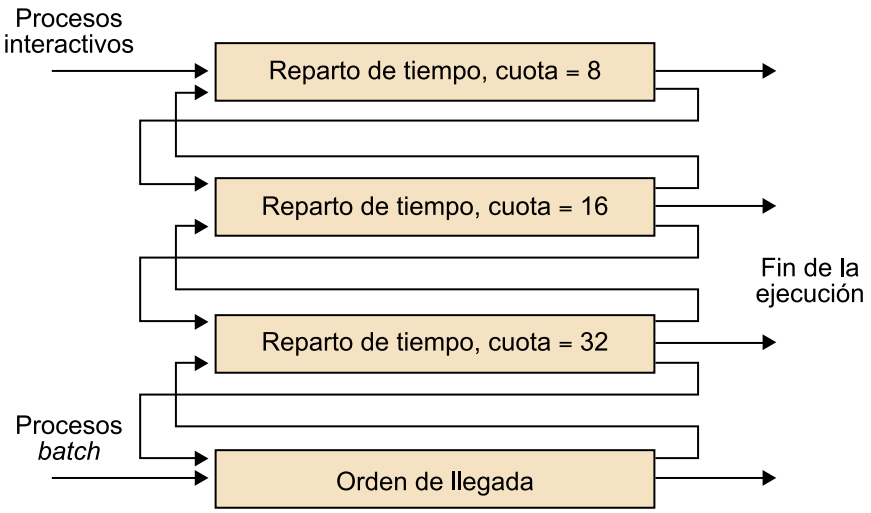
En el anexo 2 de esta asignatura encontraréis explicado con detalle el algoritmo de planificación que utiliza UNIX System V, que está basado en la planificación de colas multinivel realimentadas.

Planificación con cuatro colas multinivel realimentadas

Mostramos una posible organización de un planificador con colas multinivel realimentadas. Consta de cuatro colas. Las tres primeras se planifican utilizando el algoritmo de reparto de tiempo, y la cuarta utiliza el algoritmo de orden de llegada. La diferencia entre las tres primeras colas será el tamaño de cuota utilizado (8, 16 y 32 unidades de tiempo, respectivamente). La gestión de las colas y de los nuevos procesos será la siguiente:

- La cola con cuota = 8 tendrá preferencia sobre todas las otras: mientras haya algún proceso preparado en esta cola, no se ejecutará ningún proceso de las otras. La cola con cuota = 16 será prioritaria respecto a la de cuota = 32 y la de orden de llegada, y la de cuota = 32 será prioritaria sólo sobre la de orden de llegada.
- Los procesos interactivos se insertarán en la cola prioritaria con cuanto = 8 y los procesos por lotes irán a la cola gestionada por orden de llegada.
- Un proceso irá a la cola menos prioritaria siguiente si la duración media de sus últimas ráfagas de CPU es más larga que la cuota de la cola donde está en aquel momento.
- Un proceso irá a la siguiente cola prioritaria si la duración media de sus últimas ráfagas de CPU es inferior al tamaño de la cuota de la cola más prioritaria siguiente. Además, cada cierto tiempo podemos llevar a la cola más prioritaria siguiente todos los procesos que hace tiempo que no han recibido el procesador. Este procedimiento permite adaptar de forma dinámica la planificación del proceso a su comportamiento.

Figura 14. Planificación con cuatro colas multinivel realimentadas



5.4. Algoritmos de planificación a largo plazo

El planificador a largo plazo es el encargado de iniciar la ejecución de los procesos. Este planificador sólo tiene que seleccionar una vez cada proceso para darle la entrada al sistema, a diferencia del planificador a corto plazo, que podía seleccionar cada proceso varias veces.

Los algoritmos que aplica el planificador a largo plazo son algunos de los que hemos visto al hablar del planificador a corto plazo:

a) De orden de llegada: el algoritmo selecciona el proceso que ha estado más tiempo en espera para entrar en el sistema.

b) De menor tiempo primero: teniendo en cuenta las estimaciones del usuario, el planificador da entrada a los procesos que tienen una menor duración estimada. En los sistemas por lotes normalmente el usuario tiene que asignar una duración máxima a la ejecución de cada trabajo; mediante esta información, el sistema puede dar preferencia a los procesos más cortos.

c) Prioritario: el planificador puede asignar una prioridad a los procesos teniendo en cuenta las estimaciones del usuario, el uso actual de la máquina, la información estadística de ejecuciones anteriores del mismo programa, etc. Por ejemplo, si la mayoría de los procesos que hay en la máquina tienen ráfagas de CPU largas, este planificador puede intentar elegir uno que tenga ráfagas de CPU cortas.

Dado que la frecuencia de invocación de este planificador es muy inferior a la del planificador a corto plazo, este planificador podría implementar algoritmos de selección más complejos.

5.5. Algoritmos de planificación a medio plazo

Este planificador acostumbra a ser invocado de forma periódica, aunque menos a menudo que el planificador de corto plazo. Si detecta que el grado de multiprogramación es demasiado alto, decide llevar hacia el disco (intercambiar en el disco) alguno de los procesos del sistema. Así tiene menos procesos compitiendo por el procesador.

Los criterios para seleccionar el proceso que se envía al disco pueden tener en cuenta:

- El tiempo que ha pasado desde la última vez que este proceso fue intercambiado en el disco: intenta no penalizar siempre el mismo proceso.

- La prioridad del proceso: no vale la pena intercambiar en el disco procesos muy prioritarios, ya que, con toda probabilidad, pronto será preciso volverlos a llevar a la memoria física.

- El hecho de ser un proceso por lotes: estos procesos son menos sensibles al tiempo de retorno y, por tanto, son candidatos a ser intercambiados en el disco.

- El tiempo que ha transcurrido en el estado *espera*: si hace mucho tiempo que un proceso está bloqueado, será candidato a ser intercambiado, ya que es probable que continúe bloqueado durante cierto tiempo.

En el caso de que el aprovechamiento de la máquina empiece a ser bajo porque el grado de multiprogramación ya es demasiado pequeño, este planificador llevará hacia la memoria física los procesos que hay intercambiados en el disco para que puedan continuar ejecutándose.

5.6. Planificación en multiprocesadores

Un multiprocesador (sistema fuertemente acoplado) es un sistema de cómputo en el que varias CPU comparten el acceso a una memoria RAM común y en el que la propiedad de consistencia estricta en el acceso a memoria está garantizada por construcción.

Además de ver el mismo espacio de direcciones, hay algunos multiprocesadores que presentan otras propiedades, que tienen que ver con la distribución de la memoria. Así, por ejemplo, hay sistemas en los que el acceso a la memoria es uniforme desde el punto de vista de tiempo (sistemas UMA), frente a otros sistemas que carecen de esta propiedad (sistemas NUMA); además, los tiempos de acceso al subsistema de memoria difieren de manera significativa en función de la localización del acceso a su memoria local o a la memoria

remota. Dejando de lado los aspectos hardware involucrados, nuestro interés se centrará en los aspectos software del multiprocesador y, en particular, en el SO y su relación con la planificación de CPU.

5.6.1. Sistemas operativos para multiprocesador

Dos son los modelos más comúnmente utilizados a la hora de organizar el SO para entornos multiprocesador:

- **Modelo asimétrico de tipo maestro/esclavo**: solo una CPU procesa todas las llamadas al sistema; el resto de las unidades de cómputo pueden ejecutar código de proceso en modo no privilegiado.

- **Modelo simétrico**[14] (SMP): cualquier CPU puede ejecutar código privilegiado del SO y, además, con una adecuada distribución del núcleo por estructuras de datos, se puede dividir el núcleo en regiones críticas independientes protegidas por *mutex*, que se pueden ejecutar en paralelo.

(14)En inglés, *symmetric multiprocessing*, de ahí la sigla, SMP.

5.6.2. Planificación basada en una o múltiples colas

Frente a la planificación convencional de un único procesador, en la que la problemática residía fundamentalmente en asignar el tiempo del procesador entre el conjunto de procesos asociados a la cola de preparados, los entornos multiprocesador añaden a la ya conocida problemática temporal una nueva, que consiste en la asignación de procesos a los nodos de procesamiento. Esta nueva necesidad de planificación espacial genera la necesidad de incluir nuevos ajustes que tengan en cuenta características: de afinidad natural y estricta, de los nodos de procesamiento (*hyperthreading*, NUMA) y aspectos relacionados con la contención de recursos. En el anexo 3 de este módulo didáctico se presenta la planificación de procesos en Linux 2.6; donde se reflejan algunas de las características más relevantes que se pueden encontrar en los planificadores de CPU actuales.

Debido a la compartición de recursos por los nodos de procesamiento (arquitecturas multiprocesador con variadas jerarquías de memoria respecto a la L1, L2, L3 y Mp), la utilización de estructuras de datos centralizadas o distribuidas en el planificador ocasiona que se genere más o menos sobrecarga[15] en el uso del planificador.

(15)En inglés, *overhead*.

1) Planificadores basados en una única cola

Los primeros planificadores para multiprocesadores disponían de una única cola de procesos listos, o una única estructura de datos para todo el sistema. El funcionamiento del planificador se reduce a atender las peticiones de los nodos de procesamiento cuando quedan libres y necesitan procesos para ejecutar.

El sistema se balancea automáticamente, aunque puede generar sobrecargas en servidores donde el número de accesos a la cola de procesos del planificador se realiza con mucha frecuencia. La necesidad de introducir primitivas de exclusión mutua (*mutex*) para manejar de manera ordenada la cola de procesos listos puede producir efectos no deseados en las prestaciones del planificador.

En relación con el manejo de la afinidad, se tiene en cuenta a la hora de reasignar el proceso a la misma CPU de la que fue expulsado por la realización de una operación de E/S. Este tipo de afinidad se denomina afinidad natural, frente a la afinidad estricta, que ofrece a los procesos la posibilidad de seleccionar los procesadores en los que se ejecutarán.

Para los multiprocesadores NUMA, el planificador debe tener conciencia de la topología del sistema a efectos de extender el modelo de afinidad al nodo de cómputo o al procesador físico en su caso.

2) Planificadores basados en múltiples colas

Frente al modelo anteriormente comentado de cola única, se presenta la posibilidad de asociar a cada procesador una cola de procesos listos. Con este esquema, cada procesador se planifica de manera independiente como si se tratara de un sistema de una única CPU. No existe congestión en el acceso a las estructuras de datos del planificador, dado que cada planificador solo maneja su propia estructura. Desde la perspectiva deseable de manejar la afinidad de manera adecuada, en principio se cumple la condición, dado que cada cola mantiene sus propios procesos. No obstante, este esquema impide por construcción el balanceo automático de carga que se generaba en los sistemas de una única cola. Será deseable, por tanto, dotar a los sistemas de planificación que incorporen este modelo de mecanismos de balanceo de carga que tiendan a equilibrar la carga global del sistema, para no dejar nodos de procesamiento inactivos.

Resumen

En este módulo nos hemos centrado en los **sistemas operativos multiprogramados**. Estos sistemas intentan aprovechar mejor la máquina, ya que permiten la ejecución concurrente de varios programas; hemos llamado a cada uno de estos programas en ejecución **proceso**.

Estos procesos plantean el problema de que se tienen que ejecutar sobre un número reducido de procesadores. Por tanto, el SO tiene que ser capaz de cambiar el proceso que está en ejecución en un determinado procesador. Hemos visto que esta tarea es llevada a cabo por una rutina del SO llamada **cambio de contexto**.

Por otro lado, surge una cuestión de decisión: cuando el SO quiere cambiar el proceso que está en ejecución en un procesador, hay que decidir cuál será el nuevo proceso que se ejecutará. Este problema recibe el nombre genérico de **planificación de procesador** y, para resolverlo, se descompone en tres niveles: a **largo plazo**, a **medio plazo** y a **corto plazo**. Hemos estudiado cada uno de estos niveles presentando una serie de algoritmos de planificación para cada nivel y evaluando sus características.

Actividades

1. Dado el grafo de estados de un proceso, ¿para qué casos creéis que sería útil una transición directa de los estados *preparado* y *en espera* hacia el estado *final*?

2. Continuando con el grado de estados de un proceso, ¿qué utilidad tiene una transición directa entre el estado *en espera* y el estado *ejecución*? ¿A qué tipo de planificación corresponde?

3. ¿Tiene sentido hablar de algoritmos de planificación a largo plazo apropiativos?

4. ¿Creéis que en un sistema de planificación con colas multinivel realimentadas es razonable utilizar una planificación de orden de llegada en la cola menos prioritaria?

5. Si tenéis acceso a un computador multiprogramado, buscad información sobre el tiempo de retorno y el tiempo de uso del procesador por parte de los procesos. (Si tenéis acceso a una máquina UNIX, las peticiones *time* y *ps* nos dan esta información; consultad el manual de la máquina).

Ejercicios de autoevaluación

1. Analizad el grafo de estados de un proceso. ¿Tendría sentido una transición desde el estado *preparado* hacia el estado *en espera*?

2. Imaginad un ordenador que se dedica de forma exclusiva a la ejecución de un programa con un único hilo de ejecución. ¿Cuál de estas dos opciones creéis que se tendría que utilizar para obtener la máxima reducción del tiempo de ejecución del programa?
a) Instalar un segundo procesador (idéntico al que hay).
b) Duplicar la frecuencia del procesador.

3. Reescribimos el programa que teníamos en el ejercicio 2 considerando que hay dos hilos de ejecución y obtenemos que durante el 80% del tiempo de ejecución del programa original se pueden aprovechar ambos procesadores de forma simultánea.
a) ¿En qué factor tendríais que incrementar la frecuencia de la máquina que tiene un único procesador para obtener el mismo tiempo de ejecución que el de la máquina que tiene dos?
b) Responded la misma pregunta para un porcentaje arbitrario *P*.
Podéis suponer que al incrementar la frecuencia del procesador en un factor *K*, se reducirá el tiempo de ejecución del programa en este mismo factor.

4. Los procesos descritos en la siguiente tabla muestran la duración de las ráfagas de CPU de los procesos y el momento en que cada proceso llega al estado preparado:

	Duración de la ráfaga	Tiempo de llegada
Proceso 1	5	0
Proceso 2	7	1
Proceso 3	4	2
Proceso 4	1	3

Planificad con estos datos la ejecución de los procesos mediante los siguientes algoritmos de planificación: de orden de llegada, de menor tiempo primero, de menor tiempo pendiente y de reparto de tiempo con cuota de 1, 2, 4 y 8 ciclos. Calculad también el tiempo de retorno para cada caso.

Solucionario

Ejercicios de autoevaluación

1. No. Un proceso va al estado *en espera* cuando pide un servicio que no puede ser satisfecho de forma inmediata; pero si el proceso está en estado *preparado* no puede pedir ningún servicio, ya que no está utilizando el procesador.

2. La segunda opción permitiría reducir el tiempo de ejecución del programa, ya que el procesador es más rápido. En cambio, la primera opción es inútil, dado que el programa nunca sacará provecho del segundo procesador, porque está escrito para ser ejecutado en un único procesador.

3. a) Llamamos T al tiempo real que una máquina con un único procesador tarda en ejecutar el programa original. El tiempo que una máquina con dos procesadores que trabajan en la misma frecuencia tardaría en ejecutarlo sería el siguiente:

$$(0{,}8/2) \cdot T + 0{,}2 \cdot T = 0{,}6\ T.$$

Conocido este resultado, el incremento de la frecuencia que se tendría que aplicar a la máquina con un único procesador sería:

$$T/K = 0{,}6 \cdot T \Rightarrow K = 1{,}667.$$

b) El tiempo que tardaría una máquina con dos procesadores que funcionan en la misma frecuencia que el original (supongamos que $p = P/100$) es:

$$p/2 \cdot T + (1 - p) \cdot T = (1 - p/2) \cdot T.$$

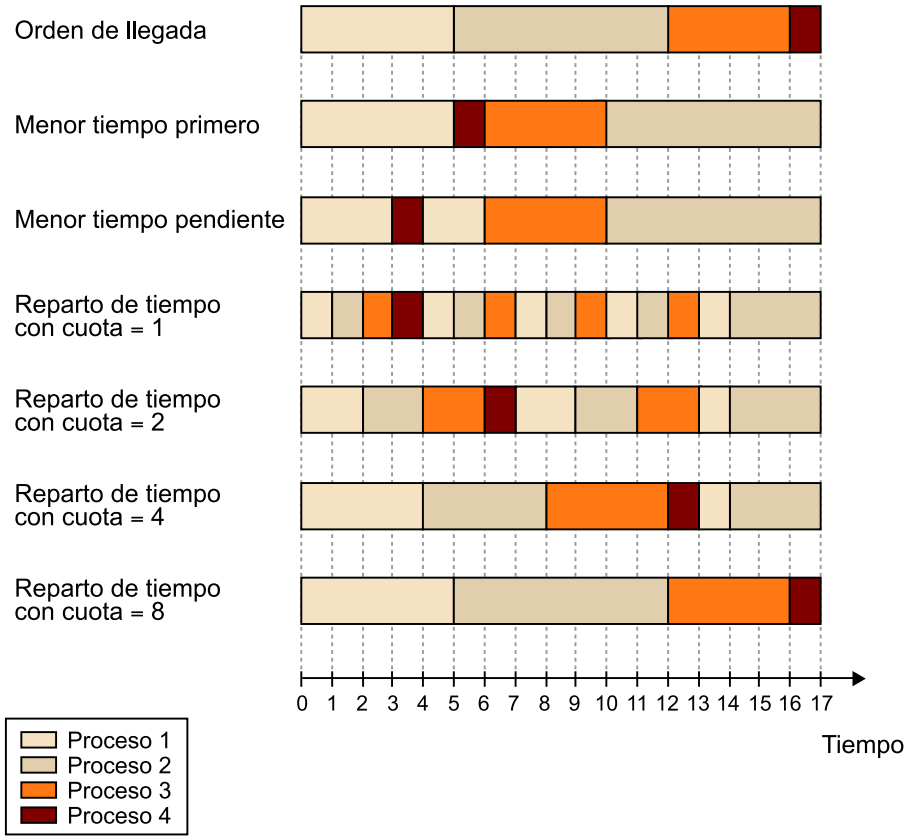
El incremento de frecuencia que necesitaría una máquina con un procesador para obtener este mismo resultado sería:

$$T/K = 1/(1 - p/2) \cdot T = 2/(2 - p) \cdot T \Rightarrow K = 2/(2 - p).$$

Por tanto, para $P = 0\%$, el factor multiplicativo es 1, y para $P = 100\%$ es 2.

4. La planificación de los procesos con los diferentes algoritmos es la siguiente:

Figura 15



A continuación mostramos el tiempo de retorno para cada proceso con cada algoritmo:

Tiempo de retorno para cada proceso con cada algoritmo							
	Orden de llegada	**Menor tiempo primero**	**Menor tiempo pendiente**	**Reparto del tiempo cuota igual a 1**	**Reparto del tiempo cuota igual a 2**	**Reparto del tiempo cuota igual a 4**	**Repartodel tiempo cuota igual a 8**
Proceso 1	5	5	6	14	14	14	5
Proceso 2	11	16	16	16	16	16	11
Proceso 3	14	8	8	11	11	10	14
Proceso 4	14	3	1	1	4	10	14

Glosario

apropiatividad Característica de los sistemas de planificación de recursos que permite retirar en cualquier momento la asignación de un recurso.

bloque de control del proceso Estructura de datos donde el SO almacena toda la información relativa a los procesos.
Sigla: PCB

cambio de contexto Conjunto de acciones gracias a las cuales el procesador deja de ejecutar código correspondiente a un hilo de ejecución de un proceso para pasar a ejecutar código perteneciente a otro hilo de ejecución (del mismo proceso o de otro).

cola multinivel Algoritmo de planificación que tiene como función asociar a cada proceso el tipo de planificación más adecuado a sus características. Esta asociación se hace en el momento de crear el proceso, y no puede variar.

cola multinivel realimentada Algoritmo de planificación que intenta asociar a cada proceso el tipo de planificación más adecuado a sus características. Esta asociación se puede modificar a medida que se ejecuta el proceso.

cuota Unidad de tiempo utilizada por los sistemas de planificación de reparto de tiempo que indica el tiempo máximo durante el cual un procesador puede estar asignado a un hilo de ejecución.

envejecimiento Mecanismo propio de los sistemas de planificación de recursos basado en prioridades que consiste en incrementar la prioridad de los procesos que hace tiempo que esperan la asignación de un recurso para evitar el problema de la inanición.

estado Descriptor asociado a cada hilo de ejecución que indica su grado de actividad actual con respecto al procesador.

grado de multiprogramación Número de procesos que hay en ejecución en un computador.

hilo de ejecución Cada una de las partes de un proceso que pueden ser ejecutadas de forma concurrente.

inanición Circunstancia que se puede dar en los sistemas de planificación basados en prioridades, en la que un proceso de prioridad baja espera de forma indefinida la asignación de un recurso porque siempre hay procesos más prioritarios.

menor tiempo pendiente Algoritmo apropiativo de planificación del procesador que asigna el procesador al hilo de ejecución en estado preparado que le falta menos tiempo para acabar la ráfaga de CPU actual.

menor tiempo primero Algoritmo de planificación del procesador que asigna el procesador al hilo de ejecución en estado preparado que tiene la siguiente ráfaga de CPU más corta.

no apropiatividad Característica de los sistemas de planificación de recursos según la cual la desasignación de un recurso es responsabilidad exclusiva de quien lo ha pedido.

orden de llegada Algoritmo de planificación del procesador que asigna el procesador a aquel proceso que ha estado más tiempo en la cola de procesos preparados.

PCB Ved bloque de control del proceso.

planificación Política que decide cuál será la siguiente entidad que podrá hacer uso de un recurso. En función del tiempo que dure la asignación, los planificadores serán apropiativos o no apropiativos.

proceso Programa en ejecución.

reparto de tiempo Algoritmo de planificación del procesador que limita el tiempo máximo durante el cual el procesador puede estar asociado a un hilo de ejecución. Este algoritmo se apropia el procesador si un hilo no lo ha abandonado antes de este tiempo máximo.

thrashing Situación en la que el tiempo de retorno de los procesos aumenta considerablemente porque el grado de multiprogramación es demasiado elevado.

tiempo de retorno Cantidad de tiempo que pasa desde que un usuario indica al SO que quiere ejecutar un programa hasta que finaliza la ejecución de un programa.

Bibliografía

Carretero, J. (2007). *Sistemas operativos: una visión aplicada* (2.ª ed.). McGraw Hill.

Silberschatz, A. (2006). *Fundamentos de sistemas operativos* (7.ª ed.). McGraw Hill.

Stallings, W. (2005). *Sistemas operativos* (5.ª ed.). Pearson-Prentice Hall.

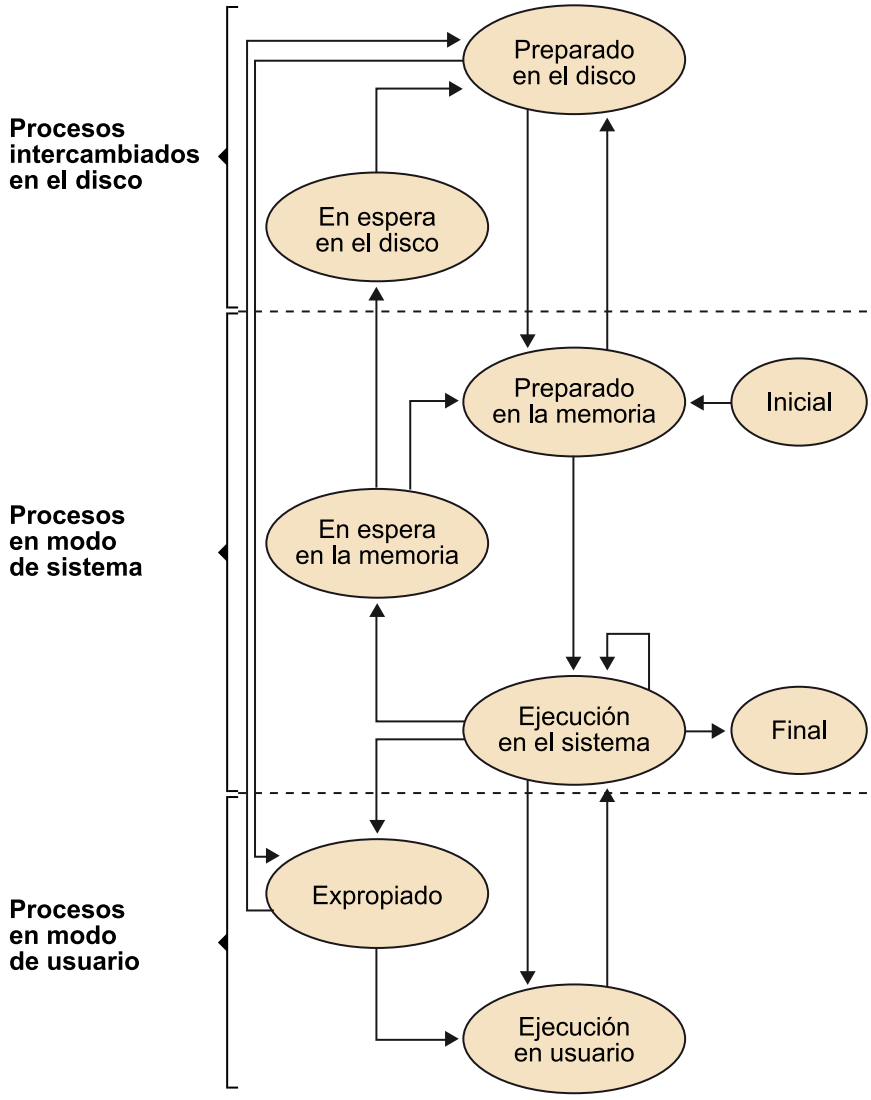
Tanenbaum, A. (2003). *Sistemas operativos modernos* (2.ª ed.). Pearson-Prentice Hall.

Anexos

Anexo 1. Diagrama de estados en UNIX System V

Como ejemplo práctico, veremos el diagrama de estados de los procesos en UNIX System V. Esta versión de UNIX se ejecutaba en una máquina con un único procesador y los procesos tenían un único hilo de ejecución.

Figura 16



En este diagrama encontramos los siguientes estados:

- ***Inicial***: se crea el proceso.

- ***Preparado en la memoria***: el proceso está residente en la memoria y está preparado para ejecutar código en modo sistema.

- ***Ejecución en sistema***:el proceso ejecuta código en modo sistema.

- ***Ejecución en usuario***: el proceso ejecuta código en modo usuario.

- ***En espera en la memoria***: el proceso está residente en la memoria, pero no está preparado para ejecutar código en modo sistema, ya que espera que se complete algún servicio pedido.

- ***En espera en el disco***: el proceso ha sido intercambiado en el disco y se espera que se complete algún servicio pedido.

- ***Preparado en el disco***:el proceso ha sido intercambiado en el disco, pero ya está en condiciones de continuar ejecutando código en modo sistema.

- ***Expropiado***: este estado es muy parecido al estado *preparadoen la memoria*. La diferencia es que los procesos en estado *expropiado* están a punto de volver a modo usuario, ya han hecho todo lo que tenían que hacer en modo sistema.

- ***Final***: el proceso se destruye. Este estado está relacionado con la sincronización que ofrecen las llamadas al sistema `wait` y `exit` y con los procesos *zombies*.

Con respecto al diagrama de estados presentado al hablar de los planificadores a medio plazo, vemos que el estado *ejecución* se ha separado en dos estados (*ejecución en usuario* y *ejecución en sistema*) y, además, ha aparecido un nuevo estado llamado *expropiado*.

El estado *ejecución* ha sido dividido para reflejar si un proceso está ejecutando código en modo usuario o en modo sistema.

El estado *expropiado* aparece a raíz de una decisión de diseño de este SO. Para simplificarlo, los diseñadores decidieron que un proceso, cuando ejecutaba código en modo sistema, se apropiaba del procesador, es decir, ningún otro proceso lo puede desbancar hasta que éste no se bloquee o vuelva a modo usuario. El estado *expropiado* es para los procesos que han perdido el procesador cuando volvían de modo sistema a modo usuario.

Sobre las transiciones, ya hemos explicado muchas de ellas. Sólo comentaremos las que aporten alguna novedad:

- ***Ejecución a sistema → ejecución a sistema***: esta transición se produce cuando llega una interrupción mientras el procesador está en modo sistema. Se ejecuta la rutina de atención a la interrupción, y cuando ésta finaliza, se continúa lo que se estaba ejecutando al recibir la interrupción. En

algunos puntos del código del SO, las interrupciones están inhibidas para evitar problemas de coherencia en las estructuras de datos del sistema.

- ***Ejecución a usuario → ejecución a sistema***: esta transición puede ser provocada por la ejecución de la instrucción `trap` o por la llegada de una interrupción. En ambos casos, el SO guarda el contexto del proceso de usuario para poder continuar su ejecución más adelante.

- ***Ejecución a sistema → expropiado***: esta transición indica que un proceso ha perdido el procesador al cambiar el modo sistema a modo usuario. Esta situación se puede producir cuando un proceso ha acabado de ejecutar una llamada al sistema o cuando ha acabado de ejecutar la rutina de atención a una interrupción que ha sido recibida durante la ejecución del proceso.

Las transiciones del diagrama de estados en las que el SO podrá hacer un cambio de contexto son las siguientes:

- ***Ejecución a sistema → final***: el proceso en ejecución ha invocado la llamada al sistema `exit`. Por tanto, cuando el proceso sea eliminado, el SO no tendrá nada más que hacer.

- ***Ejecución a sistema → en espera en memoria***: el proceso se ha bloqueado y, por tanto, hay que dar el procesador a otro proceso.

- ***Ejecución a sistema → expropiado***: el proceso ha acabado de ejecutar la llamada al sistema que había invocado, pero hay otro proceso más prioritario.

- ***Ejecución a sistema → expropiado***: durante la ejecución del proceso, el procesador ha recibido una interrupción (por ejemplo, la del temporizador). Al finalizar la atención a la interrupción, hay otro proceso más prioritario.

Anexo 2. La planificación en UNIX System V

La planificación utilizada por UNIX System V se podría definir como de reparto de tiempo con colas multinivel realimentadas, pero con alguna peculiaridad adicional. Para entender las siguientes explicaciones hay que tener presente el diagrama de estados de UNIX System V que hemos mostrado en el anexo 1. Tenemos $N + M$ colas de prioridad; N corresponden a prioridades en modo sistema y M corresponden a prioridades en modo usuario. Sólo las M colas de modo usuario son realimentadas. Cualquier cola de modo sistema es más prioritaria que una cola de modo usuario. El planificador escogerá un proceso de una cola si todas las colas más prioritarias que ésta están vacías.

Las colas correspondientes a prioridades en modo sistema se planifican según el orden de llegada, y las de modo usuario, según una política de reparto de tiempo. Hay que notar que en esta versión de UNIX, cuando un proceso ejecuta código en modo sistema no puede ser desbancado del procesador, es decir, la asignación del procesador en modo sistema es no apropiativa. Esta característica simplifica bastante la implementación del núcleo del SO y, por otro lado, no comporta demasiados problemas de rendimiento porque la duración de las ráfagas de CPU en modo sistema no son muy grandes.

UNIX System V asigna de forma dinámica una prioridad a cada proceso. Los momentos en que se calcula esta prioridad son los siguientes:

a) Cuando un proceso se bloquea: la prioridad asignada sólo depende del motivo que provoca el bloqueo. Se asignan prioridades más altas a los procesos que se bloquean por los motivos más críticos; por ejemplo, la prioridad máxima corresponde a un bloqueo causado por el intercambiador de páginas, y la mínima corresponde a la espera de la finalización de un proceso hijo (sincronización entre las llamadas *wait* y *exit*). Todas estas prioridades corresponden a prioridades de modo sistema, ya que todos los bloqueos se producen al ejecutar alguna llamada al sistema.

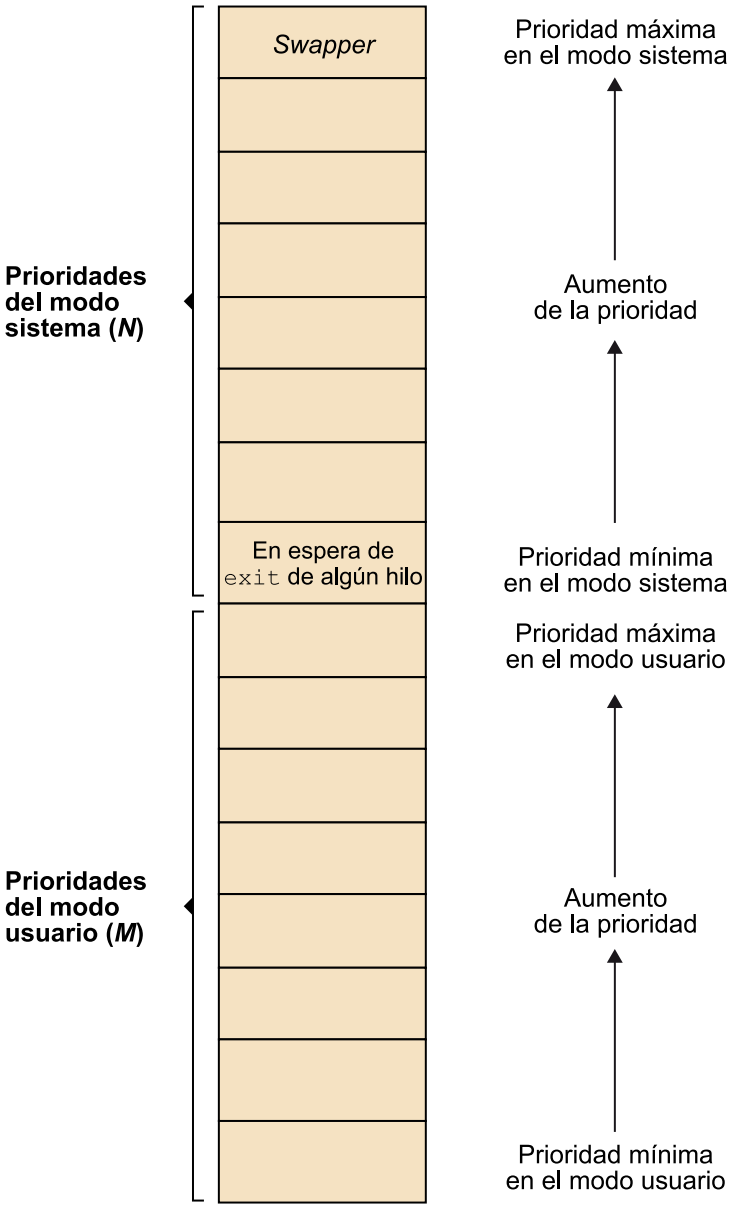
b) Cuando un proceso vuelve a modo usuario: el sistema evalúa una prioridad de usuario teniendo en cuenta la utilización de los recursos que hace este proceso. De esta forma, puede penalizar procesos que utilizan demasiados recursos. En este momento, si hay algún otro proceso más prioritario, el proceso puede ser desbancado del procesador y pasar al estado *expropiado*.

c) Cada segundo (aproximadamente): se vuelve a calcular la prioridad de todos los procesos que están en el estado *expropiado* para evitar que algún proceso monopolice el procesador. Este nuevo cálculo se hace teniendo en cuenta el uso que cada proceso ha hecho del procesador durante los últimos *K* segundos. Así, los procesos de usuario que hace tiempo que no han recibido el procesador ven su prioridad aumentada. Es posible que este cálculo no se haga exactamente cada segundo, dado que el SO puede ejecutar código dentro de alguna región crítica en la que las interrupciones estén inhibidas.

Cuando la rutina de cambio de contexto tiene que elegir el proceso siguiente que debe utilizar el procesador, UNIX elige el proceso más prioritario que esté en el estado *preparado* en la memoria o en el estado *expropiado*. En caso de haber varios procesos con la misma prioridad, se elige el que ha estado más tiempo esperando. Si no hay ningún proceso disponible, se ejecuta el proceso *null*, que mantiene ocupado el procesador hasta que se produce una interrupción. Por tanto, siempre que haya algún proceso preparado con prioridad de sistema, éste recibirá el procesador.

Sólo podrán cambiar de prioridad los procesos que tienen una prioridad de usuario y que están en el estado *expropiado*.

Figura 17



Este algoritmo de planificación está diseñado para un SO de tiempo compartido, y no es adecuado para un SO de tiempo real. En estos sistemas, algunos acontecimientos deben tener una respuesta en un intervalo de tiempo definido, pero esto no siempre se puede garantizar por el hecho de que la planificación del procesador en modo sistema es no apropiativa.

El usuario tiene la posibilidad de influir en esta planificación utilizando la llamada al sistema *nice*, que influye en la forma como se hacen los cálculos sucesivos de la prioridad de un proceso. Ahora bien, para un usuario, esta llamada sólo permite disminuir la prioridad de sus procesos.

Anexo 3. La planificación de procesos en Linux 2.6

Los avances hardware para aumentar el rendimiento de los sistemas, como son los sistemas con multiprocesador (SMP y NUMA), los procesadores con múltiples núcleos o *multicores* (CMP) y los procesadores *multithreading* (SMT), no solo han incidido de manera importante en el número de unidades de procesamiento y sus características, sino también en otros módulos de la arquitectura de los sistemas de cómputo, como el subsistema de memoria, tanto en su tecnología de fabricación como en su organización y funcionamiento dentro del sistema. Estos avances, sin duda, condicionan el diseño del sistema operativo y especialmente el diseño y funcionamiento del planificador de CPU.

La inclusión de múltiples núcleos en el nivel del procesador extiende la problemática de la planificación monoprocesador de corto plazo, hasta ahora ocupada de repartir el tiempo de CPU entre el conjunto de procesos asignados, en un problema bidimensional de planificación espacial y temporal.

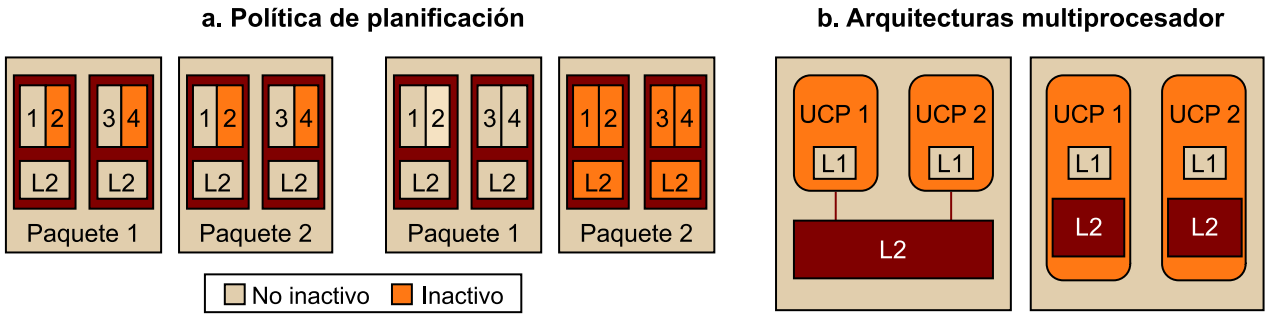
Nuevos conceptos adquieren especial relevancia:

- **Afinidad**: aplicada por algunos sistemas en multiprocesadores, puede ser de gran relevancia en los sistemas de múltiples núcleos, a efectos de solicitar al planificador seguir utilizando la misma caché, una vez desalojado el subproceso (*thread*) por necesidades de entrada/salida de este.

- **Simbiosis**: en cuanto a la mezcla del tipo de subprocesos (CPU, limitados por entrada/salida) a efectos de uso de recursos compartidos. Si dos subprocesos intentan acceder a la vez a información contenida en la caché L2, como en la arquitectura que se muestra en la figura 18 (caso *b*), se producirá contención en el uso del recurso compartido y uno de los dos deberá esperar. De ahí la necesidad de asociar al planificador la funcionalidad de mezclar de manera adecuada los subprocesos en ejecución con el fin de minimizar la posible contención en el uso de recursos compartidos.

- **Consumo de energía y número de procesos por unidad de tiempo (*throughput*)**: serán dos índices de prestaciones que se deberán decidir en cada situación por el planificador. Cuando el número de núcleos por procesador y número de procesadores aumenta de manera significativa en los sistemas de cómputo, la posibilidad de asignar los subprocesos a núcleos siguiendo esquemas de tipo turno rotatorio (*round-robin*) o *fille-up*, como se muestra en la figura siguiente (caso *a*), abre nuevas vías a la hora de construir planificadores que tengan en cuenta criterios de planificación diferentes de los tradicionales.

Figura 18



La actualización de Linux 2.6 proporciona un salto importante en la gestión de la planificación de procesos. Los planificadores CTS (*constant time scheduler*, versión 2.6.8) y CFS (*completely fair scheduler*, versión núcleo 2.6.23), desarrollados por Ingo Molnar, han supuesto una mejora sustancial en prestaciones, comparable a los planificadores existentes en otros sistemas Unix. A continuación se muestran algunas de las características más relevantes de ambos planificadores.

1) El planificador CTS

A grandes rasgos, el planificador CTS (*constant time scheduler*) es también llamado O(1) por tener complejidad constante a la hora de seleccionar el siguiente proceso de la cola de preparados en tiempo constante, con independencia de la carga. Supone ya un acercamiento notable a otros planificadores existentes, donde la eliminación del cálculo de prioridades para los procesos en modo usuario después de cada segundo de uso de CPU generaba una dependencia en prestaciones del planificador demasiado fuerte, en función del número de procesos en la cola.

El planificador de Linux 2.6.8 se estructura en un sistema de 140 niveles de prioridad, donde se asocian rangos de prioridades a efectos de catalogar diferentes tipos de tareas: por lotes (*batch*), interactivas, de tiempo real, etc. Los niveles 0 al 99 están reservados para tareas tiempo real y se planifican siguiendo una política de planificación FIFO, en la que las condiciones de apropiatividad vienen definidas por la interrupción de una tarea más prioritaria, la realización por parte del proceso de una E/S que suponga un bloqueo o la cesión voluntaria de la CPU. Los niveles del 100 al 139 se reservan a trabajos de usuario (no-tiempo real), en los que a las tareas de usuario se les asigna una prioridad estática (que podemos cambiar mediante las llamadas *nice()* y *setpriority()*), y un valor dinámico (Vd), que se adquiere en tiempo de ejecución en función de las operaciones que va realizando el proceso en ejecución, donde Vd = (Max(100, min(sp-bonus + 5,139)), siendo $0 \leq$ bonus ≤ 10. La tabla siguiente muestra la relación existente entre el tiempo de espera del proceso y el valor de bonus asignado.

Tabla. Relación entre el tiempo de espera y el valor de bonus asociado

Tiempo medio de espera	Bonus
0 ≤ N < 100 ms	0
300 ≤ N < 400 ms	3
600 ≤ N < 700 ms	6
900 ≤ N < 1.000 ms	9

Es importante notar que el tamaño de rebanada de tiempo (*timeslice*) va asociado a la prioridad del proceso:

```
If sp < 120 => slice = 20 * (140-static priority)

If sp ≥ 120 => slice = 5 * (140-static priority)
```

En una primera instancia, siempre se planifica la tarea de mayor prioridad, y si existen múltiples tareas en el mismo nivel de prioridad, se planifican entre sí en modo de turno rotatorio.

Hay dos estructuras de datos clave en el planificador de Linux 2.6.8 que permiten cumplir el objetivo en $O(1)$, y su diseño gira en torno a ellas: colas de ejecución y los *arrays* de prioridad.

a) Colas de ejecución y balanceo de carga

Una cola de ejecución almacena información acerca de los procesos que se están ejecutando en una sola CPU, si hay una cola de ejecución por cada CPU en el sistema. La información contenida en las colas de ejecución le permiten al planificador transferir el control de una CPU hacia otra usando un método denominado balanceo de carga. El balanceo de la carga es una manera de asegurar que los recursos de una CPU no serán desperdiciados mientras otra CPU se encuentre sobrecargada. Si el planificador encuentra que una cola de ejecución tiene muchos más procesos en ella que otra, uno o más procesos pueden ser movidos desde la cola más larga hacia la más corta. El balanceador de carga es invocado cada vez que se vacía una cola de ejecución; si no hay colas de ejecución vacías, es invocado periódicamente (del orden de 200 ms). El temporizador periódico le permite al sistema mantener un balance de carga razonable a través de varias CPU sin tener que dedicar demasiado tiempo a mover procesos de una CPU hacia otra. Balancear la carga cuando una cola de ejecución se vacía permite al planificador asegurar que el recurso CPU nunca va a desperdiciarse. La estructura de datos de las colas de ejecución es la estructura más básica del planificador de Linux 2.6. Es el principio sobre el cual se construye el algoritmo entero.

Esencialmente, una cola de ejecución no pierde de vista todas las tareas ejecutables asignadas a una CPU en particular. Como tal, una cola de ejecución se crea y se mantiene para cada CPU en un sistema. Cada cola de ejecución contiene dos *arrays* de prioridad. Todas las tareas en una CPU comienzan en un *array* de prioridad, el activo, y si se ejecutan fuera de su plazo de CPU, son movidas al *array* de prioridad expirada. Durante el movimiento se calcula un nuevo plazo de CPU. Cuando no hay más tareas ejecutables en el *array* de prioridad activa, simplemente se intercambia con el *array* de prioridad expirada (que exige simplemente actualizar dos punteros). El trabajo de las colas de ejecución consiste en no perder de vista la información de los hilos especiales de la CPU (hilo ocioso, hilo de migración) y manejar sus dos *arrays* de prioridad.

b) *Arrays* de prioridad

Esta estructura de datos es la base para la mayor parte del comportamiento ventajoso del planificador de Linux 2.6, en particular su tiempo de funcionamiento O(1) (constante). El planificador de Linux 2.6 siempre planifica la tarea de prioridad más alta en un sistema, y si existen múltiples tareas en el mismo nivel de prioridad, son planificadas mediante turno rotatorio entre ellas. Los *arrays* de prioridad hacen que encontrar la tarea de mayor prioridad de un sistema sea una operación de tiempo constante, y también posibilitan el comportamiento de turno rotatorio con niveles de prioridad en tiempo constante. Además, el uso de dos *arrays* de prioridad al unísono (los *arrays* de prioridad activo y expirado) hace las transiciones entre las épocas de rebanada de tiempo una operación de tiempo-constante. Una época es el tiempo entre que todas las tareas ejecutables comienzan con una nueva rebanada de tiempo, y que lo terminan de usar.

Los *arrays* de prioridad son un *array* de listas enlazadas, una para cada nivel de prioridad (140 niveles de prioridad). Cuando una tarea es agregada a un *array* de prioridad, es añadida a la lista para su nivel de prioridad. Un mapa de bits de tamaño *MAX_PRIO* + 1 (realmente puede ser un poco más grande puesto que debe ser implementado en bloques de tamaño de palabras) tiene un bit activo para cada nivel de prioridad que contenga tareas activas. Para encontrar la tarea de mayor prioridad en un *array* de prioridad, uno solamente ha de encontrar el primer bit activo en el mapa de bits. Múltiples tareas con la misma prioridad son planificadas mediante turno rotatorio; después de su ejecución, las tareas son puestas al final de su lista de nivel de prioridad. Dado que encontrar el primer bit en un mapa de bits de longitud finita y encontrar el primer elemento en una lista son operaciones con un límite superior finito en cuanto al tiempo que la operación puede tardar, esta parte del algoritmo de planificación se realiza en un tiempo constante *O*(1).

Cuando una tarea se ejecuta sobrepasado su rebanada de tiempo, se quita del *array* de prioridad activo y se pone en el *array* de prioridad expirado. Durante este movimiento, se calcula una rebanada de tiempo nueva. Cuando no hay más tareas ejecutables en el *array* de prioridad activo, los punteros a los *arrays*

de prioridad activo y expirado simplemente se intercambian. Debido a que se recalculan las rebanadas de tiempo cuando son desalojados, no hay un punto en el que todas las tareas a la vez necesiten el cálculo de nuevas rebanadas de tiempo; es decir, se realizan muchas pequeñas operaciones de tiempo constante en vez de iterar sobre las tareas; sin embargo, hay muchas tareas y cálculos de rebanadas de tiempo para ellas (que sería un indeseable algoritmo $O(n)$). El intercambio de los punteros de los *arrays* activo y expirado es una operación de tiempo constante, que evita la operación de tiempo $O(n)$ de mover n tareas de un *array* o cola a otro, ya que todas las operaciones involucradas en el mantenimiento de un sistema de *arrays* de prioridad activo y expirado ocurren en un tiempo constante $O(1)$.

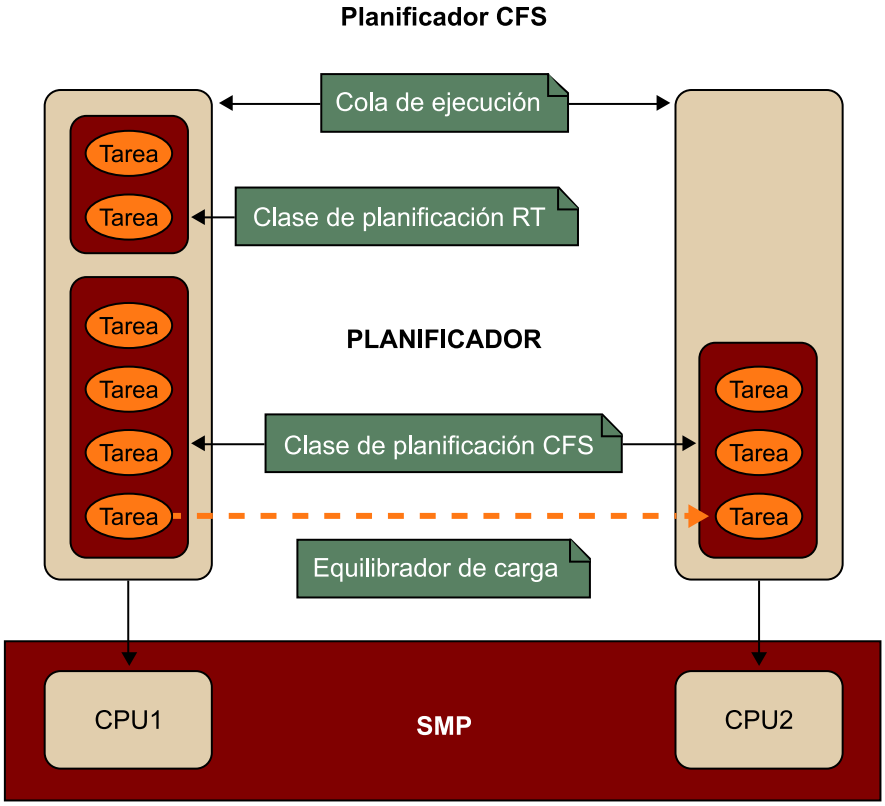
En la misma línea, se mejoran las prestaciones de los procesos interactivos, e introducen mejoras en la asignación del tiempo de CPU, hacia modelos de equitatividad, impidiendo la posposición indefinida de algunos procesos, frente a la asignación indiscriminada del tiempo de CPU por parte del planificador hacia otros procesos.

Siguiendo la tendencia de otros sistemas operativos de tipo generalista, se potencia la apropiatividad del núcleo, haciendo más predecible el núcleo a efectos de mejorar la ejecución de aplicaciones de tiempo real débil, donde los índices de rendimiento se basan en cumplir los requerimientos de tiempo de las aplicaciones (*deadlines*, tiempo de respuesta acotado, calidad de servicio, etc.).

2) El planificador CFS

CFS (*completely fair scheduler*) supone una ruptura en relación con las versiones anteriores al planificador Linux 2.6.23. Se abandona la idea de asociar a cada valor de prioridad una rebanada de tiempo, se desestima el tiempo de seguimiento de los períodos de inactividad de los procesos a efectos de cambio en la prioridad dinámica y se elimina la identificación por tipo de proceso entre otras características.

Figura 19



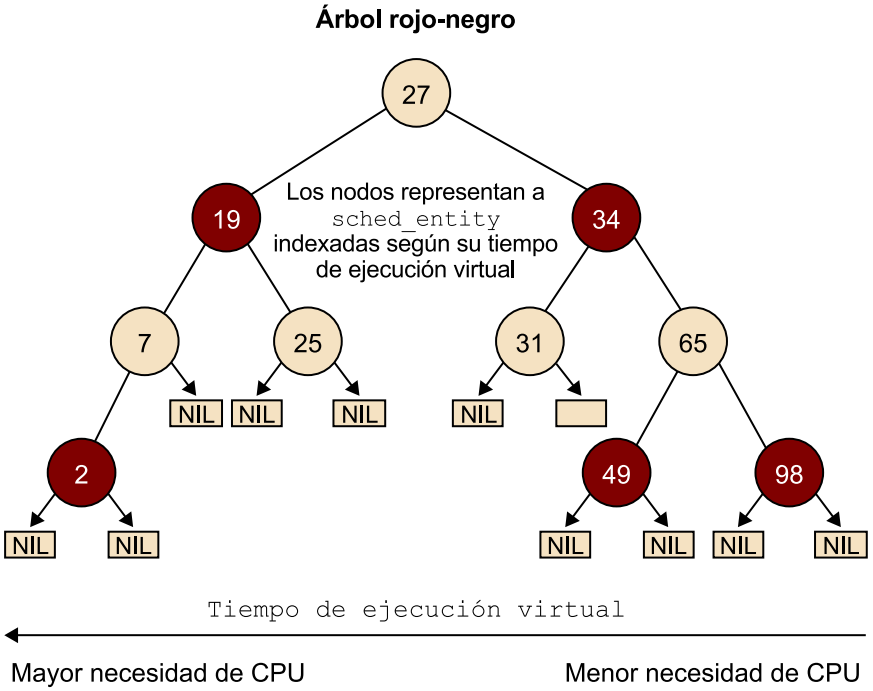
El planificador CFS de Linux contiene una cola de ejecución (RQ) que se crea para cada procesador, donde la clase *scheduler* se introduce en el núcleo 2.6.23 y se trata de una jerarquía de módulos extensibles. Estos módulos de programación se encargan de encapsular detalles de la política que se va a desarrollar y se llaman desde el núcleo del planificador. Hay dos clases de políticas implementadas en el núcleo 2.6.32, que a continuación se cometan brevemente:

- Completamente equitativa (CFS *schedule class*).
- Tiempo real (RT *schedule class*).

Como objetivo de diseño, el CFS trata de implementar un modelo preciso de multitarea donde se puedan ejecutar varios procesos al mismo tiempo, dando a cada uno de ellos una potencia de procesamiento igualitaria. Puede que no se disponga de una CPU donde se permita simultanear la ejecución de procesos. No obstante, se puede medir la cantidad de tiempo de ejecución que cada tarea ha tenido y tratar de garantizar que todas ellas tengan la parte de tiempo que les corresponde.

Esto se lleva a cabo manteniendo una variable para cada tarea (*vruntime*) que registra los valores de uso de CPU en nanosegundos. Así, por ejemplo, un valor de *vruntime* inferior indica que la tarea ha tenido menos tiempo para calcular y, por lo tanto, tiene más necesidad del procesador. Con objeto de cumplir el objetivo marcado, en lugar de una cola, CFS utiliza un árbol rojo-negro para almacenar, ordenar y planificar las tareas, como se observa en la figura 20.

Figura 20



La clave para cada nodo es el *vruntime* de la tarea correspondiente. Para seleccionar la siguiente tarea que se va a ejecutar, simplemente se elige el nodo más a la izquierda. No obstante, en las primeras implementaciones, en lugar de seguir el *vruntime*, CFS realizaba el seguimiento del tiempo de espera de una tarea cuando no se estaba ejecutando y lo decrementaba cuando la tarea se ejecutaba. El objetivo era mantener este valor lo más cerca de cero para todas las tareas siempre que fuera posible.

Un árbol rojo-negro (RN) es un árbol de búsqueda binaria, lo que significa que para cada nodo, el subárbol izquierdo solo contiene las claves de menor valor que la clave del nodo, y el subárbol de la derecha contiene las claves de mayor o igual valor a él. No obstante, un árbol RN tiene más restricciones, lo que provoca que sean más eficientes que los árboles binarios ordenados y puedan realizar las operaciones de búsqueda en tiempos de complejidad O (log n).

CFS no utiliza directamente las prioridades, o las colas de prioridad, sino que utiliza la prioridad para modular la acumulación de tiempo en la variable *vruntime*. En esta versión, la prioridad es inversa a su efecto (una tarea de mayor prioridad acumulará *vruntime* más lentamente, ya que necesita más tiempo de CPU). Del mismo modo, una tarea de prioridad baja tendrá su incremento *vruntime* más rápidamente, lo cual hará que sea expropiada antes.

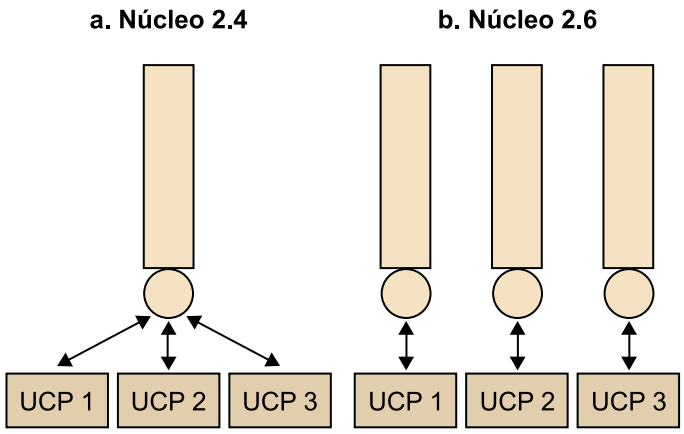
El algoritmo CFS es mucho más simple que el anterior CTS, y no requiere muchas de las variables de CTS. El tiempo cuando se producirá la apropiatividad de la CPU es variable, dependiendo de las prioridades y el tiempo real de funcionamiento, así que no es necesario asignar a las tareas un segmento de tiempo determinado.

CFS, en la versión 2.6.24, extiende su funcionalidad y permite la programación de grupos, agregando otro nivel de equidad. En este modelo, las tareas pueden ser agrupadas juntas, asociadas a un usuario propietario, y CFS puede ser aplicado tanto en el nivel de grupo, como en el nivel de tarea individual. Por ejemplo, para tres grupos, CFS distribuiría a cada grupo un tercio del tiempo de CPU aproximadamente, y luego dentro de cada grupo se dividiría ese tiempo en función de las tareas de cada grupo.

Los aspectos más relevantes del planificador CFS se presentan a continuación:

- El primero de ellos es el relacionado con el soporte a las arquitecturas de multiprocesamiento. Se mejora la escalabilidad del sistema mediante la eliminación de la cola de ejecución global (caso *a*) y la introducción de colas de ejecución independientes por procesador (caso *b*) y se incorpora el soporte a sistemas NUMA, como se muestra en la figura 21.

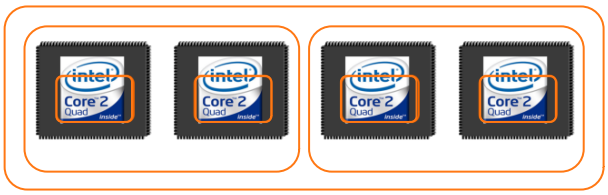
Figura 21



- El segundo motivo, relacionado con el primero, es la mejora en flexibilidad del sistema de planificación mediante un mecanismo de conjuntos multinivel que intenta representar la estructura del hardware del sistema de manera jerárquica, denominado dominio de planificación. En la figura 22 se muestra un multiprocesador jerárquico de tipo NUMA, que consta de cuatro nodos en cada uno de los cuales hay 2 procesadores físicos, cada uno de ellos del tipo doble núcleo (*dual-core*), que podrían implementar mediante *hiperthreading* más procesadores lógicos. Si ahora suponemos procesadores lógicos compartiendo la caché de primer nivel, procesadores físicos compartiendo la caché de segundo nivel y procesadores multinúcleo compartiendo un nodo de un multiprocesador NUMA compartiendo la memoria de ese nodo, la planificación del sistema resultante se complica de manera significativa. Técnicas de coplanificación (*gang scheduling*) y planificación por grupos de procesos extenderían el abanico de complejidad y nueva problemática que se debería asumir en los nuevos entornos.

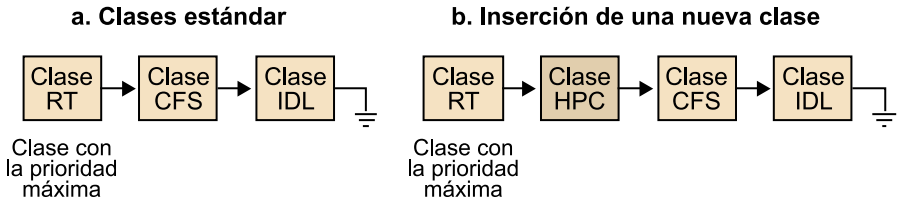
Figura 22

Relación de dominios en arquitecturas NUMA



- A partir de la versión del núcleo 2.6.23, se incorpora lo que se denomina sistema de planificación modular (*modular scheduler core*). Se introducen las denominadas clases de planificación, como se muestra en la figura 23, que encapsulan los detalles de las políticas de planificación, y que son gestionadas por el núcleo del sistema de planificación de manera "casi" trasparente. Esta mejora permite a los desarrolladores incorporar nuevas políticas de planificación de una manera más sencilla al núcleo del sistema, por un lado, minimizando su tiempo de desarrollo y, por otro lado, reduciendo el riesgo de errores en la modificación del código núcleo.

Figura 23



Con la adición del nuevo sistema de planificación, se rediseñan los esquemas existentes de planificación en tiempo real (TR) y se incorpora un nuevo esquema de planificación para las tareas de usuario o no-TR, y el CFS sustituye al planificador CTS debido a su poco grado de equitatividad.

Todavía hay algunos problemas sin solucionar en relación con la falta de respuesta del planificador CFS, en el área de juegos en 3D y en su aplicación eficiente, en entornos de tiempo real débil. No obstante, es difícil diseñar un planificador de tipo generalista que responda de manera adecuada a cualquier tipo de carga de trabajo, de ahí que se prefiera dotar al programador de las llamadas del sistema pertinentes para que sea él quien resuelva en el espacio de usuario los problemas de planificación que se le presenten.