3 Planificación de procesos y procesadores

La forma en que se reparte el uso de la CPU entre los procesos tiene un enorme impacto en el rendimiento de un sistema multiprogramado, por lo que siempre se ha prestado una gran atención a las políticas de planificación que se implementan y se han elaborado multitud de conceptos relacionados con ello. El estudio de estas políticas es el objeto de este capítulo. Se presta también atención a la planificación en multiprocesadores, que añade una dimensión espacial al problema, y a la planificación de tiempo real.

Contenido

3.1 Introducción

En el capítulo anterior se ha estudiado cómo se representan los procesos y se implementan las transiciones de estado entre ellos y los cambios de contexto. Uno de los objetivos de un sistema operativo multiprogramado es proporcionar una utilización eficiente de los recursos del proceso, permitiendo a los procesos un uso de ellos que evite situaciones de inanición. Todo esto es lo que persigue una **política de planificación** adecuada, que determina los criterios de elección del siguiente proceso a usar la CPU. Evaluar la calidad de una política de planificación es complejo y presenta diferentes perspectivas, dependiendo de los intereses de las aplicaciones, lo que lleva a definir previamente un conjunto de **parámetros de rendimiento**. El rendimiento de una determinada política de planificación dependerá también del **comportamiento de los programas**, por lo que la elección de una u otra política deberá tener en cuenta el tipo de procesos que ejecuta el sistema, fundamentalmente si están **orientados a cálculo** o son **interactivos**. Algunas aplicaciones, como las de **tiempo real**, imponen unos requisitos muy particulares en el uso del procesador, lo que hace difícil su convivencia con las aplicaciones habituales en los sistemas de propósito general (de **tiempo compartido**), conduciendo a políticas de planificación de tiempo real específicas.

Hoy en día son cada vez más habituales las plataformas **multiprocesador**. Aunque la política de planificación es independiente, en general, del número de unidades de proceso, los sistemas multiprocesador requieren políticas complementarias que tienen como objetivo un compromiso entre el equilibro de la carga de los procesadores, que conduce a una mejor utilización de estos, y el aprovechamiento de la localidad de los procesos, que impulsa a mantener a cada proceso en un mismo procesador durante su ejecución.

El trabajo de planificación reside en gran parte en una función *scheduler* del núcleo del sistema operativo, pero otras partes del sistema pueden colaborar en esta tarea, normalmente modificando los parámetros que utiliza el scheduler para decidir qué proceso planificar. En general la planificación puede repartirse en tres niveles [STA05]:

- En la llamada al sistema de *ejecutar programa*. Cuando se crea un proceso se puede decidir alguno de los criterios para su planificación, como por ejemplo la prioridad inicial y el quantum. A esta planificación se la denomina de **largo plazo**.

- En la función scheduler. Cada vez que un proceso abandona la CPU, toma la decisión de qué proceso planificar en función de la política de planificación establecida y del valor de los parámetros de planificación. A esta planificación se la denomina de **corto plazo**.

- Otras partes del sistema operativo pueden intervenir en la planificación, bien periódicamente (como en algunos sistemas UNIX que estudiaremos), bien de forma indirecta, como es el caso del *swapper* de memoria[1]: al sacar un proceso de memoria por problemas de espacio, hace que este no sea inmediatamente planificable. A este tipo de planificación se la denomina de **medio plazo**.

La Figura 3.1 muestra qué transiciones de estado están involucradas en cada plazo de la planificación.

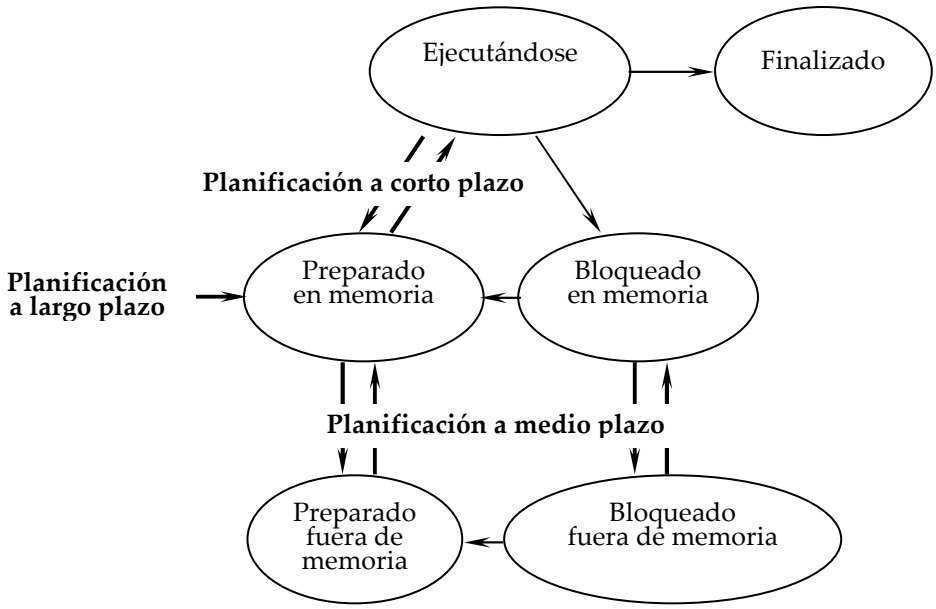

Figura 3.1. Plazos de la planificación.

3.2 Evaluación del rendimiento

La selección de una determinada política de planificación de procesos se basa en un conjunto de **parámetros de rendimiento** cuya importancia relativa depende de algunas características particulares del sistema (por ejemplo, interactivo o batch, existencia de procesos de tiempo real…), lo que determina los compromisos que hay que establecer en la selección de las política y mecanismos de la gestión de procesos. En general, todo mecanismo introduce una sobrecoste en tiempo de ejecución que contrarresta parte de los beneficios obtenidos por la introducción del mecanismo.

Los parámetros de rendimiento que utilizaremos como criterios para analizar la calidad de una política de planificación se basan en los definidos en el Capítulo 1, particularizados para su aplicación al recurso procesador:

[1] Se estudiará con la gestión de memoria.

Eficiencia

Se refiere a la eficiencia temporal. Se expresa como el porcentaje de tiempo en que la CPU se mantiene ocupada haciendo *trabajo útil*. Por trabajo útil se entiende la ejecución de código de los programas (y de los servicios solicitados por éstos). Cabe esperar que un sistema multiprogramado sea mucho más eficiente que uno monoprogramado, ya que en éstos la CPU está ociosa cuando en programa espera por una operación de E/S pudiendo haber programas esperando a ejecutarse, por lo que ese tiempo contará como *tiempo perdido*. Sin embargo, un sistema multiprogramado tiene que ejecutar las tareas propias de la gestión de procesos (scheduler y del dispatcher), y éstas deben computarse también como tiempo perdido.

$$Ef_t = \frac{t_{útil}}{T} = \frac{t_{útil}}{t_{útil} + t_{perdido}}$$

A veces resultará interesante referirse a la **pérdida de eficiencia** como:

$$1 - Ef_t = \frac{t_{perdido}}{T}$$

Productividad (*throughput*)

En lo que respecta a la gestión de procesos, mide el número de programas que se ejecutan por unidad de tiempo. Incluye otras muchas características que afectan el rendimiento del sistema, como por ejemplo la velocidad del procesador, que habrá que compensar si se comparan máquinas con distinto hardware.

Tiempo de finalización

Considera el rendimiento del sistema desde el punto de vista del programa que se ejecuta. Globalmente, se puede expresar como el tiempo desde que se solicita la ejecución de un programa hasta que ésta finaliza. Es una medida válida para sistemas batch. Lo denotaremos t_f

Tiempo de espera

Mide exclusivamente los tiempos totales de espera de un proceso en la cola de preparados, t_w, eliminando la dependencia de la duración del propio programa. Depende en cierta medida, sin embargo, del número de veces que éste se bloquea.

Debe tenerse en cuenta la siguiente relación entre tiempos: si t_{CPU} es el tiempo que pasa el proceso en la CPU y t_{bloq} es el tiempo total que el proceso está bloqueado, entonces

$$t_f = t_w + t_{CPU} + t_{bloq}$$

Tasa de CPU

La relación entre el tiempo de CPU del programa y su tiempo de espera expresa la **tasa de CPU**, que indica el grado de disfrute del procesador que ha tenido el proceso:

$$TasaCPU = \frac{t_{CPU}}{t_{CPU} + t_w}$$

Latencia (tiempo de respuesta)

Mide el tiempo desde que un proceso entra en el estado de preparado (porque se crea o porque se desbloquea) hasta que entra en ejecución. Sólo tiene sentido en sistemas interactivos. Lo denotaremos t_r.

Equidad (predecibilidad)

Los parámetros anteriores (en particular el tiempo de terminación, la latencia y el tiempo de espera) se miden con parámetros estadísticos. La media parece ser el parámetro más indicativo. Sin embargo, otros parámetros de la distribución estadística de una medida del rendimiento, como por ejemplo la varianza de la latencia, son de gran interés. Una varianza pequeña en la latencia es indicadora de un comportamiento homogéneo, lo que es deseable por dos motivos: por una parte el comportamiento de los programas es más predecible; por otra parte indica un comportamiento más ecuánime y menos discriminatorio. En cambio, varianzas muy grandes pueden indicar riesgo de inanición para algunos programas.

3.3 Comportamiento de los programas

Para desarrollar mecanismos efectivos de planificación de procesos es necesario conocer previamente el comportamiento típico de los programas en ejecución. Este comportamiento se refiere a dos aspectos fundamentales:

(1) El número de transiciones entre estados que se producen durante la ejecución.

(2) El tiempo relativo que un proceso gasta en cada estado.

Medidas empíricas han mostrado que estos parámetros son muy variables de unos programas a otros. En concreto, para el parámetro de mayor interés, que es el tiempo de servicio de un proceso en la CPU, es decir, el tiempo consecutivo de CPU que un

proceso necesita (**intervalo de CPU**), se ha observado la distribución de probabilidades que se muestra en la Figura 3.2 (distribución *heavy-tailed*).

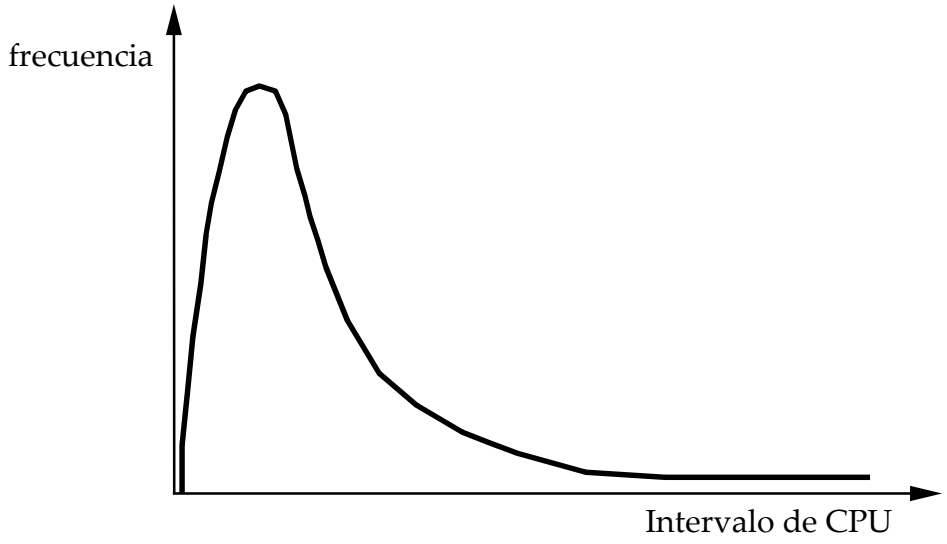

Figura 3.2. Histograma de intervalos de CPU de una muestra típica de procesos.

Se obtiene como consecuencia que, aunque la mayoría de los procesos necesitan la CPU para pequeños intervalos de tiempo, bloqueándose con frecuencia (programas interactivos u **orientados a entrada/salida**), existe un número de procesos que tienden a utilizar la CPU durante un tiempo prácticamente indefinido (programas **orientados a cálculo**). Este hecho va a condicionar el diseño de la planificación de procesos.

3.4 Políticas de planificación

A continuación describiremos las políticas generales de planificación de procesos, haciendo referencia a los mecanismos que los implementan. Las evaluaremos de acuerdo a los parámetros de rendimiento introducidos previamente.

Los algoritmos de planificación que se exponen en esta sección no se limitan a un plazo de planificación concreto. Aunque en general asumiremos planificación a corto plazo (entrada de procesos a la CPU), algunos algoritmos pueden implementarse también a largo plazo o distribuirse en más de un tipo de planificación (por ejemplo, a corto y medio plazo).

Para describir una política de planificación hay que considerar los siguientes aspectos:

- Cómo se selecciona el proceso que entrará a ejecución. De entre los procesos que están en estado preparado, se elige uno de acuerdo a criterios como, por ejemplo,

prioridades, tiempo que lleva en la cola de preparados, tasa de CPU que le ha correspondido...

- Cuándo se lleva a cabo la planificación. Este aspecto afecta fundamentalmente a la planificación a corto plazo. Hay dos alternativas básicas: si únicamente se planifica cuando un proceso abandona la CPU porque acaba o se bloquea (**políticas no expulsoras**), o si se puede forzar al proceso que está usando la CPU a abandonarla para planificar otro proceso (**políticas expulsoras**).

- La existencia o no de más de una política de planificación, ajustadas a los diferentes tipos de procesos, y cómo se combinan. En los sistemas de propósito general coexisten procesos de tipos diferentes que pueden requerir políticas específicas, por lo que se suelen definir varias políticas particulares, cada una adecuada a un tipo de procesos, y una política global aplicable al conjunto de tipos de procesos (**planificación multinivel**).

- Además, en multiprocesadores, en qué procesador se ejecuta el proceso, lo que estudiaremos en la Sección 3.5.

Abordaremos primero los criterios de selección de un proceso y luego consideraremos los aspectos de expulsión y la planificación multinivel.

3.4.1 Selección de un proceso para entrar en la CPU

Podemos considerar las siguientes políticas, comenzando por la más sencilla y terminando por las más generales, basadas en prioridades.

3.4.1.1 FCFS

Se elige el proceso a entrar en la CPU según el tiempo que lleva esperando; es decir, en el orden de entrada al estado de preparados.

La implementación de esta política es sencilla: la cola de procesos preparados se gestiona con disciplina FIFO.

Presenta el problema del **efecto convoy**. La existencia de programas muy largos

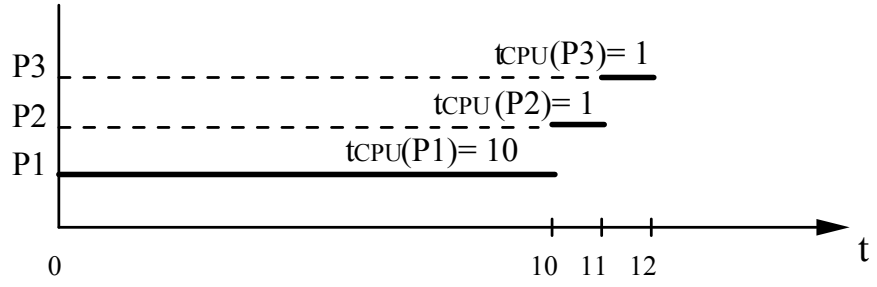

Figura 3.3. Planificación FCFS. Efecto convoy.

dispara el tiempo de finalización, la latencia y el tiempo de espera medios. En el ejemplo de la Figura 3.3 se muestra el orden de ejecución de acuerdo a FCFS de tres procesos que llegan en el orden P1, P2, P3. La latencia media, T_r, es (0+10+11)/3= 7 unidades de tiempo, frente a una T_r mínima de 1 que se obtendría si los procesos se planificasen en orden P2, P3, P1.

3.4.1.2 SJF (el más corto primero)

Se selecciona para entrar a ejecutarse el proceso de menor duración. Esta planificación es (teóricamente) óptima para los tiempos medios de respuesta, finalización y espera.

Es necesario estimar la duración de un proceso. Esta estimación dependerá del plazo de la planificación.

- Largo plazo. El usuario proporciona la estimación.
- Corto plazo. El scheduler puede predecir la duración del próximo intervalo de CPU en función de las duraciones de los intervalos de CPU anteriores. Así, la duración prevista para el instante $n+1$, τ_{n+1}, se calcula a partir de una media ponderada de la duración del intervalo n, t_n, y predicciones anteriores:

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$$

  donde $0 \leq \alpha \leq 1$.

Al trabajar con estimaciones en lugar de duraciones reales, el éxito de la planificación dependerá de lo adecuado que sea el modelo de estimación.

3.4.1.3 Prioridades

A cada proceso se le asocia una prioridad, a partir de las cual se establece su orden para la planificación. Para igual prioridad, suele tomarse orden FCFS.

Habitualmente, la prioridad de un proceso se almacena como un entero positivo dentro de su PCB, que se inserta en la cola de preparados de acuerdo a ella para facilitar la labor del scheduler (en planificación de corto plazo).

La prioridad se establece inicialmente en función de determinados parámetros, como el propietario del proceso, el tipo del proceso (por ejemplo, procesos del sistema, de tiempo real, batch), su tamaño, los recursos que requiere, etc. La prioridad inicial puede usarse para planificar a largo plazo.

Si la prioridad inicial no cambia durante la ejecución del proceso y ésta se usa para planificar a corto plazo, se obtiene una política de **prioridades estáticas**. Un problema de las prioridades estáticas es que algunos parámetros, fundamentalmente los referidos al consumo de recursos, y en particular al tiempo de CPU y duración de sus intervalos, no se conocen a priori, por lo que la prioridad puede no ser indicativa de comportamiento del programa. Otro problema es el riesgo de inanición para los procesos de prioridad baja, lo que además puede tener consecuencias para el propio sistema operativo: si un proceso de prioridad alta realiza espera activa para entrar en una sección crítica ocupada por un proceso de prioridad menor, la sección crítica nunca se liberará[2].

En general, resulta más adecuada una planificación por **prioridades dinámicas**. Un proceso se crea con una prioridad base inicial, y su prioridad dinámica se recalcula periódicamente, fundamentalmente de acuerdo al gasto de CPU del proceso. La prioridad de los procesos que más tiempo de CPU han gastado disminuye en relación a la prioridad de los procesos que han gastado menos[3].

Existen fundamentalmente dos formas de ajustar la prioridad:

(a) Periódicamente, como por ejemplo UNIX System V, UNIX 3.2BSD y Linux, que ajustan conjuntamente las prioridades de todos los procesos cada cierto tiempo.

(b) Aperiódicamente, como por ejemplo Windows NT y UNIX SVR4, que aprovecha, la transición de estado de un proceso para recalcular la prioridad

3.4.2 Expulsión

Los algoritmos vistos hasta ahora se aplican en el momento en que se decide planificar un proceso, cuando la CPU queda libre. En principio, esto ocurre cuando el proceso que la ocupa se bloquea o termina. Si no hay más oportunidades para planificar, las políticas de planificación resultantes se denominan **no expulsoras**. Estas políticas permiten el abuso de CPU por parte de procesos orientados a cálculo, en detrimento de los orientados a entrada/salida. Piénsese por ejemplo en un proceso que ejecute un bucle de espera activa sin que se cumpla nunca la condición de salida. La posibilidad alternativa es forzar la expulsión del proceso que ocupa la CPU en determinadas situaciones, llevándolo a la cola de preparados y provocando una nueva planificación (Figura 3.4). Estas políticas de planificación **expulsoras** tienden a

---

[2] Este fenómeno se conoce como *inversión de prioridad*.

[3] Obsérvese que SJF a corto plazo es un caso particular de planificación por prioridades dinámicas.

incrementar la equidad en el reparto de la CPU y contribuyen a evitar la inanición de los procesos.

Fundamentalmente, hay dos formas de provocar la expulsión del proceso, que no son excluyentes:

(a) **expulsión por evento** (transición de un proceso a preparados,[4] ya sea por desbloqueo o por creación de uno nuevo), que permite la planificación de **procesos de tiempo real**.

(b) **expulsión por tiempo**, que conduce a sistemas de **tiempo compartido**.

Las políticas expulsoras, como las no expulsoras, pueden aplicar cualquiera de los algoritmos de selección del proceso descritos anteriormente.

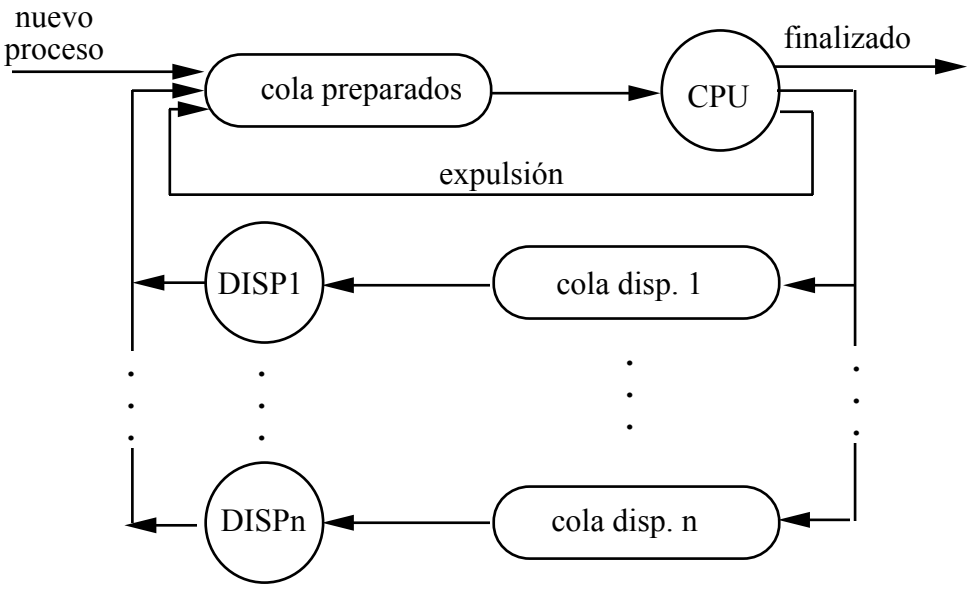

Figura 3.4. Modelo de colas en un sistema expulsor

3.4.2.1 Expulsión por evento

La expulsión del proceso que ocupa la CPU cuando se produce la transición de otro proceso a preparados (ya sea cuando se crea el proceso o cuando se cumple la condición de desbloqueo), otorga éste (y al resto de procesos preparados) la oportunidad de entrar a la CPU, lo que beneficia a los tiempos de respuesta. Esta forma de expulsión resulta en general muy conveniente en sistemas de propósito

[4] A menudo, cuando se habla de planificación expulsora en un sistema operativo, se refiere exclusivamente a esta forma de expulsión.

general, y es un mecanismo necesario cuando se tienen procesos de tiempo real[5]. En este caso se requiere además un mecanismo de prioridades que asigne la máxima prioridad a los procesos de tiempo real, permitiéndoles tiempos de respuesta acotados

Una política particular de expulsión por eventos es la que resulta de aplicar expulsión al algoritmo SJF. El algoritmo así obtenido, ***SRTF*** (Primero el de Menor Tiempo Restante), presenta una planificación sobre la base de los tiempos de CPU que les restan a los procesos para finalizar.

3.4.2.2 Turno circular o *Round-Robin (RR)*

La expulsión se produce por tiempo. En el algoritmo de turno circular puro, la elección entre los procesos preparados se asume por FCFS (así lo consideraremos a lo largo de esta sección), pero puede combinarse también con otras políticas, como prioridades.

Asociado a cada proceso se define un ***quantum*** de tiempo, q, o cantidad máxima de tiempo de CPU que se permite a un proceso consumir ininterrumpidamente. El quantum puede ser un parámetro común para todos los procesos o propio de cada uno, y entonces se almacena en el PCB del proceso. La rutina de atención al reloj es la encargada de comprobar si el proceso en ejecución ha alcanzado su quantum, y entonces promover su expulsión y la consiguiente planificación.

Los sistemas con planificación a corto plazo Round-Robin se denominan **sistemas de tiempo compartido**, pues el tiempo de procesador se ve como un recurso que se comparte por turno entre los procesos.

El valor de q determina el comportamiento de estos sistemas[6]. Una propiedad interesante de esta política es que permite acotar el tiempo de respuesta de los procesos para un grado de multiprogramación dado N, eludiendo el efecto convoy. Considerando aquí q como el mismo para todos los procesos:

$$t_r \leq (N-1)q$$

[5] Procesos de tiempo real son aquéllos que tienen establecidas determinadas restricciones en sus tiempos de respuesta o finalización, y por lo tanto su ejecución no puede verse afectada en sistemas multiprogramados por circunstancias como la carga del sistema. Más adelante dedicaremos un apartado a la planificación específica de tiempo real.

[6] Obsérvese que para un q muy grande ($q \rightarrow \infty$), RR tiende a comportarse como FCFS no expulsora.

De aquí resulta obvio que q debería ser lo más pequeño posible, con el fin de minimizar la latencia. Para $q \to 0$, $t_r \to 0$, lo que optimiza el rendimiento medido bajo este criterio. En el límite, cada proceso ve el sistema como si dispusiese de un procesador ideal dedicado exclusivamente a la ejecución de ese proceso con velocidad $1/N$ de la del procesador real de la máquina. Este concepto se conoce como **procesador compartido.**

Pero la latencia no es el único criterio de rendimiento. Para calcular la eficiencia, hay que considerar la sobrecarga introducida en el cambio de contexto, (tiempo de ejecución del scheduler y el dispatcher, t_{cs}, que definíamos como trabajo no útil)[7]. Para una carga suficientemente grande de procesos orientados a cálculo, el número de cambios de contexto por unidad de tiempo tenderá a $1/q$, por lo que el tiempo perdido por los cambios de contexto[8] será t_{cs}/q. En el cálculo de la eficiencia temporal, de acuerdo a lo definido previamente, el trabajo útil por unidad de tiempo, que determina precisamente la eficiencia del sistema, será:

$$Ef_t = 1 - t_{cs}/q$$

Como puede observarse, cuando q tiende a t_{cs}, la eficiencia tiende a cero[9], ya que el número de cambios de contexto es muy grande, y por lo tanto también el tiempo gastado en realizarlos. En consecuencia, otro criterio en la implementación de un sistema de tiempo compartido es establecer un quantum lo suficientemente grande con respecto al tiempo de cambio de contexto:

$$q >> t_{cs}$$

Para encontrar un compromiso entre este criterio y el de procesador compartido hay que basarse en el comportamiento de los programas, que matiza la expresión de la eficiencia anterior. En efecto, con una distribución típica de la duración de los intervalos de CPU, es posible elegir un quantum no demasiado grande que permita a los procesos agotar la mayoría de sus intervalos de CPU (zona sombreada de la Figura 3.5), manteniendo reducido el número de cambios de contexto adicionales provocados por el tiempo compartido.

---

[7] El cálculo de los retardos introducidos por los cambios de contexto requeriría un análisis mucho más profundo, ya que, como consecuencia de la pérdida de localidad, un cambio de contexto tiende a incrementar a corto plazo el número de fallos en el acceso al TLB y a memoria virtual.

[8] Suponiendo que los procesos son orientados a cálculo.

[9] Obsérvese que para $q<t_{cs}$ esta fórmula carece de sentido, ya que se obtienen valores negativos para la eficiencia. En la práctica, el sistema sólo ejecutaría cambios de contexto.

Como la evolución tecnológica está proporcionando incrementos enormes en la velocidad de ejecución de los procesadores, y el tiempo de cambio de contexto puede considerarse como que se mantiene constante, la condición de $q >> t_{cs}$ se consigue cada vez con mayor amplitud, lo que permite a los sistemas operativos acercarse al ideal de procesador compartido.

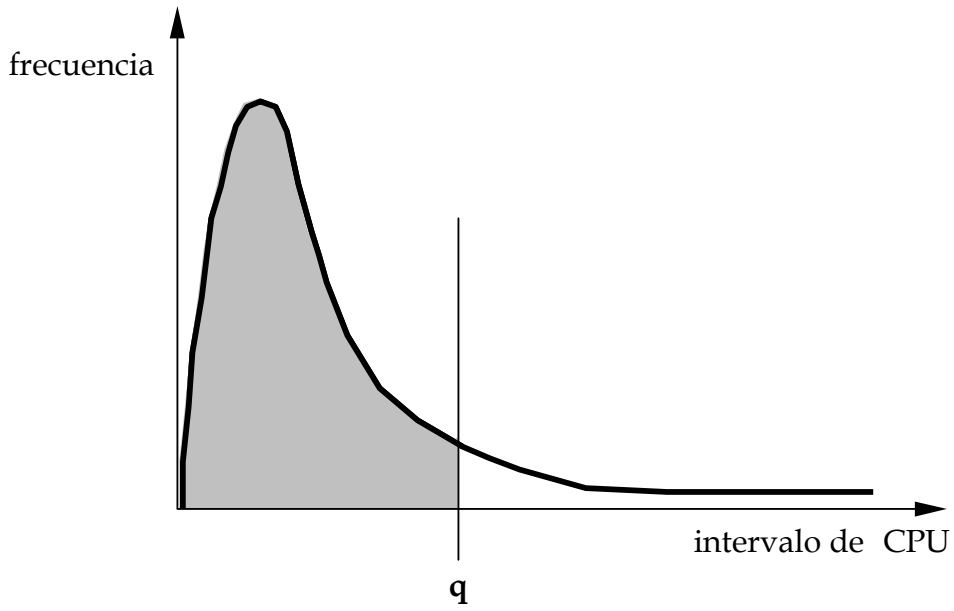

Figura 3.5. Elección de q

3.4.3 Planificación multinivel

En los sistemas operativos de propósito general coexisten procesos con diferentes necesidades. Los usuarios lanzarán habitualmente procesos interactivos, para los que una planificación de tiempo compartido es adecuada, pero quizás haya que ejecutar también procesos de tiempo real, que no pueden estar sujetos a una expulsión por tiempo. Si fuera posible identificar en un sistema clases diferenciadas de procesos (por ejemplo tiempo real, interactivos, batch…), tendría interés establecer una cola de preparados para cada clase de procesos, según se muestra en la Figura 3.6. Por ejemplo, en UNIX SVR4 se distinguen procesos de tiempo real, procesos del sistema y procesos de tiempo compartido.

Este esquema requiere dos niveles de planificación:

(1) Planificación dentro de cada cola. Cada cola puede utilizar su propia política de planificación, de acuerdo a la clase de procesos que acoge.

(2) Planificación entre colas. En general se hará por prioridades, bien de forma absoluta (sólo se planifica una cola si todas las de prioridad mayor están vacías, como ocurre en UNIX), o distribuyendo la planificación de forma que

una cola de mayor prioridad reciba mayor número de planificaciones que una de menor prioridad.

Si no es posible determinar estáticamente clases de procesos, la planificación multinivel también tiene interés si se permite a los procesos cambiar de cola (planificación multinivel con **colas realimentadas**). En la planificación de colas realimentadas es preciso definir adicionalmente los criterios de paso de una cola a otra. Por ejemplo, se podría asociar prioridad creciente y quantum decreciente a las clases de procesos de la Figura 3.6. Los procesos comenzarían en una clase determinada y pasarían a la inferior o a la superior dependiendo de si agotan el quantum o se bloquean antes, respectivamente. Los procesos orientados a CPU tenderían a caer a las clases inferiores y se ejecutarían con poca frecuencia pero con un quantum mayor, al contrarío de los interactivos. Esto tiene efectos positivos sobre el rendimiento, equilibrando el tiempo de respuesta y la pérdida de eficiencia ocasionada por las expulsiones.

Como ejemplo, en UNIX los procesos de tiempo compartido se gestionan mediante colas realimentadas. Se asocia una cola a cada nivel de prioridad y el paso entre colas se gestiona a medio plazo en función del gasto de CPU de cada proceso.

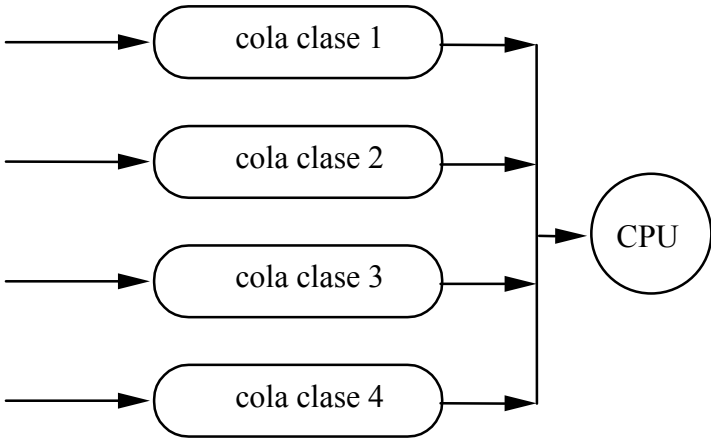

Figura 3.6. Planificación multinivel

3.5 Planificación en multiprocesadores

Consideraremos un multiprocesador como una máquina con un conjunto de procesadores que comparten un mismo espacio de direcciones de memoria física. Por esta razón, también se les llama **multiprocesadores de memoria compartida**[10] o

[10] *Shared-memory Multi-Processor*, *SMP*. No debe confundirse con el término *Symmetric Multi-Processing*, que utiliza el mismo acrónimo.

fuertemente acoplados, a diferencia de los multicomputadores, donde los elementos de proceso no comparten memoria física, y que no vamos a tratar aquí. Sin embargo hay que recordar que en un multiprocesador moderno existen niveles de memoria cache (normalmente dos, interna y externa) privados para cada procesador, lo que, como veremos, tiene relevancia para la elección de una política de planificación.

Las aplicaciones modernas están constituidas por *threads* o flujos de ejecución que se comunican por memoria compartida. Los sistemas operativos tradicionales planifican procesos. Para obtener un buen rendimiento es deseable que los sistemas operativos de hoy en día, en particular los sistemas multiprocesador, reconozcan el thread como entidad planificable. En lo que sigue utilizaremos en general el término proceso para referirnos a la entidad planificable, salvo cuando sea pertinente distinguir entre procesos y threads.

La planificación de procesos en un sistema multiprocesador presenta dos componentes:

- **Planificación temporal**, que define la política de planificación en cada procesador individual, exactamente igual que si de un monoprocesador se tratase, salvo por el hecho de que en los multiprocesadores es menos relevante para el rendimiento la política que se elija, y tanto menos cuanto más procesadores. En la planificación temporal se decide *qué procesos* se ejecutan.

- **Planificación espacial**, que define cómo se asignan los procesadores a los diferentes procesos. En la planificación espacial se decide *en qué procesadores* se ejecutan los procesos.

3.5.1 Criterios para la planificación en multiprocesadores

En la planificación espacial es necesario considerar dos hechos relevantes en la ejecución de las aplicaciones multiprogramadas o multithread:

(1) A medida que proceso se ejecuta en un procesador determinado va dejando una *huella* (estado del proceso en la memoria cache del procesador donde se ejecuta), por lo que parece adecuado mantener a un proceso en el mismo procesador (criterio de **afinidad al procesador**). Sin embargo, las políticas que potencian la afinidad al procesador tienden a provocar un desequilibrio en la carga de los procesadores, que se traduce en una utilización menos eficiente de los recursos. El criterio opuesto (**reparto de la carga**), provoca sobrecargas en el bus de memoria al invalidar frecuentemente el contenido de las memorias cache.

(2) En las aplicaciones concurrentes de hoy en día, de tipo *multithread*, los threads habitualmente interaccionan de manera muy estrecha (utilizando memoria

compartida como mecanismo de comunicación), por lo que la carencia de recursos para planificar alguno de los threads puede provocar que otros threads de la aplicación que sí disponen de procesador no progresen y consuman ciclos de CPU esperando por condiciones que deberían satisfacer los threads no planificados, lo que conduce a pobres resultados en cuanto a latencias y tiempos de finalización. Este hecho invita a implementar políticas de asignación de procesadores que permitan a una aplicación disponer de los recursos necesarios para que la ejecución de la aplicación progrese adecuadamente. El criterio de proporcionar **recursos exclusivos** para la aplicación, en vez de compartidos, tiene como contrapartida la posibilidad de que gran parte de ellos queden ociosos o con un uso poco eficiente, debido a que el sistema operativo no conoce a priori el perfil de necesidades de la aplicación[11].

Las políticas de planificación en multiprocesadores atienden a combinaciones de estos criterios.

3.5.2 Políticas de planificación en multiprocesadores

Estas políticas pueden clasificarse en dos grandes grupos no disjuntos, en función de si el conjunto de procesadores se multiplexa entre las aplicaciones en el tiempo (políticas de **tiempo compartido**) o en el espacio (políticas de **espacio compartido**).

3.5.2.1 Políticas de tiempo compartido

Una primera opción es mantener los criterios de la planificación de los monoprocesadores y no gestionar espacialmente el uso de los procesadores. En el caso más simple se puede utilizar una **cola global única** de procesos preparados, que van asignándose a los procesadores libres. Esta política optimiza el reparto de la carga en detrimento de la afinidad al procesador, siendo la alternativa utilizar una **cola local** para cada procesador, de forma que la asignación al procesador se hace inicialmente y luego el proceso siempre se planifica en el mismo procesador, potenciando la afinidad.

---

[11] De hecho, la tendencia es dejar a la aplicación gran parte de la responsabilidad de la planificación. La idea es que el sistema operativo asigne recursos de proceso en función de las necesidades de la aplicación, mientras que la librería de threads (en el espacio de usuario) planifica el uso de los recursos entre los threads de la aplicación de acuerdo a sus propios criterios, frecuentemente especificados en el código mediante llamadas explícitas, como cesión del procesador a otro thread, modificación de la prioridad, etc. La utilización de mecanismos de colaboración entre el sistema operativo y la librería contribuye a optimizar la utilización de los recursos.

3.5.2.2 Políticas de espacio compartido

Estas políticas combinan planificación temporal y espacial. Se utilizan en entornos de cálculo intensivo, generalmente con aplicaciones multithread. Pueden clasificarse en dos tipos, que no se excluyen entre sí: políticas de planificación en grupos y políticas de particionado.

Planificación en grupos

Si el sistema operativo reconoce el thread como entidad planificable, tiene sentido dedicar recursos suficientes a los threads de una misma aplicación, planificando simultáneamente el conjunto de threads en un grupo de procesadores. El uso de los procesadores se multiplexa en el tiempo para cada conjunto de threads, como en las políticas de tiempo compartido, pero ahora se añade una componente de compartición espacial. De nuevo, la gestión de cada grupo puede hacerse mediante una cola única, primando el reparto, o mediante colas locales, primando la afinidad al procesador.

Este tipo de políticas también reciben el nombre de *gang scheduling* o *co-scheduling*.

Particionado

El conjunto de procesadores se reparte entre la aplicaciones de forma que a cada una se le asigna un subconjunto de procesadores o partición. La asignación puede ser **fija** o **variable**, y en este último caso puede hacerse **estáticamente** (al ejecutar la aplicación) o **dinámicamente**, modificando la asignación en tiempo de ejecución de acuerdo a las necesidades de la aplicación y a la disponibilidad de procesadores. El particionado dinámico requiere un módulo que monitorice la carga del sistema, si se desea optimizar el rendimiento. El particionado fijo, como se recordará, produce fragmentación interna, que en términos de procesadores se entiende como que puede haber procesadores ociosos, contribuyendo a disminuir la eficiencia temporal del sistema. El particionado flexible estático, aunque en menor medida, también presenta este problema.

3.6 Planificación de tiempo real

Procesos de tiempo real (en este contexto es más habitual el término **tareas de tiempo real**) son aquellos cuya corrección no sólo depende del resultado, sino también de si lo producen antes de un plazo determinado de tiempo. En otras palabras, el tiempo de finalización está acotado. Esta restricción implica que, en general, la ejecución de una tarea de tiempo real no puede verse afectada por circunstancias como la carga

del sistema. Ejemplos de tareas de tiempo real abundan en los sistemas de control, que disponen de un límite máximo de tiempo para evaluar una entrada (por ejemplo, un sensor térmico) y producir la respuesta adecuada (por ejemplo, regular una válvula). Son numerosos los campos donde se requiere proceso de tiempo real: control de procesos industriales, fabricación inteligente, robótica, vehículos autónomos, control de tráfico, telecomunicaciones, electrodomésticos, multimedia, predicción meteorológica…

En un sistema de propósito general los procesos de tiempo real conviven con procesos de tiempo compartido o procesos del propio sistema, por lo que es muy difícil garantizar siempre el cumplimiento de los plazos. Además, los sistemas operativos de propósito general de hoy en día no garantizan una fiabilidad absoluta, por lo que, en general, no pueden usarse para sistemas críticos. Sistemas como UNIX SVR4 o Linux poseen una clase de procesos de tiempo real, pero la existencia de trozos de código no expulsable en el núcleo hace que sea difícil acotar los tiempos de respuesta. Por lo tanto, a veces se precisan sistemas de tiempo real con políticas de planificación específicas.

Introduciremos las siguientes definiciones:

Instante de liberación, F_i. Una tarea i no puede comenzar a ejecutarse antes del instante F_i (antes estará bloqueada o no se habrá creado).

Tiempo de ejecución, e_i. Tiempo que necesita la tarea i usando los recursos de la máquina para producir una respuesta.

Plazo, D_i. Instante de tiempo en el que la tarea i debe haber respondido. Es condición necesaria:

$$e_i < D_i - F_i$$

Periodo, P_i. Tiempo entre dos instantes de liberación de la tarea i. Habitualmente, $P_i = D_i - F_i$

Tarea periódica. P_i es constante.

Tarea aperiódica. P_i es variable. Puede estar acotado o no. Si está acotado, una tarea aperiódica se convierte en periódica asignándole como periodo su cota de periodo menor.

Algunas aplicaciones de tiempo real exigen que el plazo de tiempo se cumpla siempre, por las graves consecuencias que su incumplimiento pudiera tener. Esto ocurre por ejemplo cuando un sistema de control como el comentado regula un actuador en una central nuclear o en una aeronave. A estos sistemas se les denomina

sistemas de tiempo real **críticos** (*hard real time*). En otras aplicaciones pueden permitirse incumplimientos esporádicos de los plazos. Por ejemplo, un sistema de descompresión de vídeo puede congelar la imagen durante unos cuantos cuadros si no es capaz de ejecutar el algoritmo de decomprensión a tiempo en un momento dado, sin más consecuencias mientras esto no ocurra con excesiva frecuencia. A estos sistemas se les denomina sistemas de tiempo real **acríticos** (*soft real time*).

En cuanto al cumplimiento de los plazos, en algunos sistemas la ejecución de las tareas tiene un **plazo firme**, en el sentido de que si se rebasa el plazo, aún mínimamente, la respuesta es inútil o incluso contraproducente. En cambio, los sistemas de **plazo flexible** admiten algún retraso en el incumplimiento del plazo. El valor de la respuesta en estos sistemas disminuye progresivamente por encima del plazo. Es habitual que una aplicación de tiempo real admita un plazo flexible con un límite. Por ejemplo, en predicción meteorológica se define un plazo para obtener la predicción con el objetivo de poder informar con tiempo suficiente (por ejemplo 24 horas). Si no se dispone de la predicción en ese plazo, informar de la predicción con retraso sigue siendo válido, pero el valor de la predicción disminuye con el retraso y es nulo en 24 horas, cuando el instante predicho pasa a estar en el pasado. [12]

Cada vez más, muchos sistemas de tiempo real son componentes de otros sistemas, en los que realizan funciones de control. Se les denomina **sistemas empotrados** (*embebded systems*). Generalmente el sistema de control no se percibe desde fuera. Los sistemas empotrados empiezan a estar presentes en todo tipo de máquinas y dispositivos en la industria y en el hogar. Por ejemplo, en automoción son sistemas empotrados los sistemas de control de frenada y de estabilidad.

Finalmente hay que advertir que en un sistema de tiempo real las tareas pueden interaccionar entre sí, comunicándose información. Sin embargo, debido al carácter introductorio de este texto y dada la complejidad de este aspecto, no lo vamos a considerar aquí y supondremos que las tareas son independientes.

3.6.1 Criterios para la planificación de tiempo real

El criterio fundamental de un algoritmo de planificación de tiempo real para un conjunto de tareas de tiempo real $\{T_1, T_2, \ldots, T_N\}$ es proporcionar **planificación viable**, es decir, debe ser posible planificar la ejecución de las tareas de forma que se cumplan los plazos de todas ellas. Una condición necesaria para ello es:

$$\sum_i e_i / P_i < 1$$

[12] En otros casos el tipo de plazo depende de si el sistema admite o no un *funcionamiento degradado* o si debe proporcionar un estado de *parada segura* cuando no cumple el plazo.

Con tareas periódicas es posible un análisis de viabilidad estático (a priori). Si existen tareas aperiódicas, es preciso convertirlas en periódicas con periodo el mínimo con el que la tarea pueda liberarse. En caso contrario, el análisis sólo puede hacerse en tiempo de ejecución.

El criterio de planificación viable es absoluto para los sistemas de tiempo real crítico. En sistemas acríticos, si los plazos no se pueden cumplir, los criterios a aplicar se dependen del tipo de sistema. En sistemas de plazo firme, el criterio es minimizar el número de retrasos; por el contrario, en sistemas de plazo flexible, el criterio es minimizar la magnitud de los retrasos.

3.6.2 Políticas de planificación de tiempo real

Se dividen en dos grupos, considerando si admiten un plan de ejecución estático, lo que implica que las tareas deben ser periódicas, o si la ejecución debe planificarse dinámicamente. Estas últimas no sirven para sistemas críticos.

3.6.2.1 Planificación estática

Cuando se conocen los periodos y los tiempos de ejecución es posible establecer a priori un plan de ejecución. En el caso más sencillo, se establece un **ejecutivo cíclico**, asociando a instantes periódicos de tiempo la liberación de las tareas. Otros métodos más generales siguen criterios basados en prioridades:

Frecuencia monótona *(Rate Monotonic, RM)*

Es una política expulsora donde la más prioritaria es la tarea de menor periodo.

Planificación de plazo más cercano *(Earlier Deadline First, EDF)*

Es una política expulsora que planifica la tarea cuyo plazo está más próximo a expirar. Esta política es óptima con respecto al criterio de planificación viable (es decir, la condición necesaria para la viabilidad es también suficiente, lo que no ocurre con RM).

3.6.2.2 Planificación dinámica

Con tareas aperiódicas no es posible establecer un plan de ejecución a priori. En el momento en que se libera una tarea, se establece la planificación con el objetivo de hacerla viable. En algunos sistemas, ni siquiera en ese momento es posible un plan de viabilidad (por ejemplo, si no se conocen los tiempos de ejecución). Sólo una vez cumplido el plazo se sabe que no es viable, y entonces se aborta la ejecución de la tarea. Este último tipo de planificación, habitual en

sistemas de propósito general, se denomina **planificación dinámica del mejor resultado** (*best-effort*).

3.7 Ejemplos de planificación

3.7.1 Planificación en sistemas UNIX clásicos

Cada familia UNIX implementa su propia planificación de procesos, aunque, al menos en lo que respecta a los *UNIX clásicos* (en particular System V y 4.3BSD, que tomaremos como referencia) todas siguen la misma filosofía. Todos los sistemas UNIX están muy orientados a tiempo compartido. Los procesos del sistema ejecutan código no interrumpible y están asociados a las prioridades más altas. Los de usuario tienen planificación expulsora y de tiempo compartido con gestión dinámica de prioridades. Los UNIX clásicos no proporcionan planificación de tiempo real. Detalles sobre la planificación de procesos en las diferentes familias de UNIX pueden encontrase en [VAH96].

En el System V se distinguen 128 niveles de prioridad, siendo la prioridad 0 la más alta y la 127 la más baja[13]. Los niveles de prioridad de 0 a 49, estáticos, sólo se asignan a los procesos del sistema, no interrumpibles o interrumpibles, dependiendo del nivel de prioridad (el *swapper* tiene la prioridad 0).

Los procesos de usuario pasan a modo *kernel* (sistema) cuando ejecutan una llamada al sistema, ocupando el nivel de prioridad de sistema que le corresponde a ese servicio. Tras ejecutar el servicio vuelven a modo usuario en estado preparado.

La planificación del tiempo compartido se implementa mediante colas multinivel realimentadas. Se asocia una cola a cada nivel de prioridad. Cada cola sigue una disciplina FCFS.

Un proceso parte con una *prioridad base*. La prioridad dinámica del proceso se ajusta a partir del consumo de CPU del proceso. Cada interrupción de reloj (cada 1/60 s o cada 1/100 s, dependiendo del sistema), se incrementa en una unidad el consumo de CPU del proceso en ejecución. Este parámetro (*p_cpu*) se usa para recalcular las prioridades de los procesos preparados cada cierto periodo de tiempo (1 segundo en UNIX System V). Con este periodo se ajusta para todos los procesos el parámetro *p_cpu* dividiéndolo por 2. En ese momento también se recalculan las prioridades de todos los procesos preparados usando este parámetro como factor de ajuste de la prioridad:

---

[13] En el SVR4, como veremos, valores mayores corresponden a prioridades más altas.

nueva_prioridad= prioridad_base + p_cpu/2

La evolución de la prioridad dinámica de un proceso a través del tiempo puede verse en la Figura 3.7. Se observa cómo la prioridad de un proceso sube suave y periódicamente (T vale típicamente 1 segundo) hacia su prioridad base cuando no consume CPU, para caer bruscamente después de ser planificado.

Esta política presenta un par de problemas:

- La prioridad de un proceso, al incrementarse en función del tiempo sin ser planificado, asciende al mismo ritmo en situaciones de carga elevada, en detrimento de los procesos con una prioridad base menor, que tienden a sufrir inanición. Para rectificar este comportamiento, 4.3BSD incorpora a la expresión de cálculo de la prioridad el grado de multiprogramación del sistema.

- Recalcular las prioridades de todos los procesos una vez por segundo requiere un gasto elevado de CPU, mayor cuanto más cargado está el sistema.

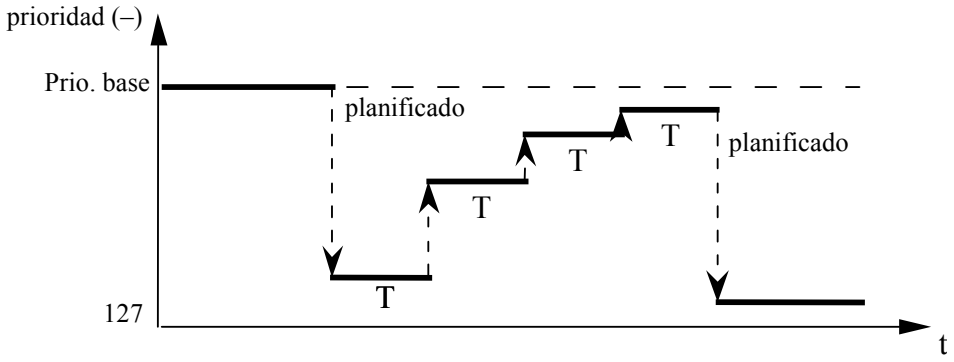

Figura 3.7. Evolución de la prioridad de un proceso en UNIX a lo largo del tiempo

La prioridad dinámica también se puede modificar a partir de la llamada al sistema *nice()*. El *shell* de UNIX proporciona también un comando *nice* para ejecutar un programa con una prioridad base inicial diferente a la establecida, lo que permite una forma de planificación a largo plazo.

Finalmente, en UNIX podemos considerar como planificación a medio plazo el swapping de procesos. El proceso *swapper* (cuyo identificador es el 0) se encarga de sacar procesos de memoria cuando el paginador tiene dificultades para encontrar marcos de memoria libres[14]. La elección del proceso suele basarse en criterios de tiempo que el proceso lleva en memoria y cantidad de memoria asignada.

[14] El funcionamiento del paginador se estudiará en la gestión de memoria.

3.7.2 Planificación en UNIX SVR4

UNIX SVR4 (*System V Release 4*) introdujo suficientes novedades sobre la versión anterior como para considerarlo como un sistema operativo diferente en cuanto a su estructura interna. Se puede considerar como modelo de los UNIX modernos, incluidas las últimas versiones de Linux.

El código de las llamadas al sistema cuenta con *puntos de expulsión* para permitir la existencia de una clase de procesos, de tiempo real. Los procesos de tiempo real ocupan los 60 niveles más altos de prioridad (de un total de 160) y pueden desbancar incluso a un proceso del núcleo cuando éste alcance el punto de expulsión.

La gestión de las prioridades de los procesos de tiempo compartido (que ocupan los 60 niveles menos prioritarios), utiliza un mecanismo radicalmente diferente al de versiones anteriores. Las prioridades de los procesos ya no se recalculan por tiempo, sino que cada proceso ajusta su prioridad por evento, de acuerdo a un conjunto de parámetros asociados a cada nivel de prioridad, que son reconfigurables en la instalación.

3.7.3 Planificación en Windows

Microsoft comenzó en 1988 el desarrollo del sistema operativo Windows NT[15], destinado a estaciones de trabajo y servidores de tamaño pequeño y medio. A partir de 1994 se han comercializado sucesivas versiones (3.51, 4.0, 2000, XP, Vista). Se trata de un sistema operativo que podemos considerar moderno en virtud de su estructura modular cliente-servidor (aunque no posee propiamente un micronúcleo) y de su orientación hacia soportes hardware distribuidos. Uno de los objetivos de diseño fue la transportabilidad, que se proporciona a través de una interfaz de abstracción del hardware (HAL), existiendo implementaciones para diferentes arquitecturas, como 80x86 de Intel, Alpha de Digital y MIPS.

Windows NT planifica threads por prioridades, siguiendo un esquema parecido al de UNIX. Cuenta con 32 niveles de prioridad divididos en dos clases. Los 16 niveles superiores, de prioridades estáticas, constituyen la clase de *tiempo real*. Los 16 inferiores (clase *variable*), donde se ubican los threads de usuario, son de tiempo compartido y se gestionan con disciplina FCFS. Un thread de usuario perteneciente a un proceso *P* parte con una prioridad base inicial que puede estar en el rango *[prio(P) –2 , prio(P)+2].* La prioridad del thread fluctúa dinámicamente durante la ejecución en el rango *[prio(P)–2, 15]*, incrementándose cuando el thread se bloquea y decrementándose cuando el thread es expulsado por fin de quantum. La Figura 3.8

---

[15] NT viene de *New Technology*.

muestra los rangos de prioridades de un thread perteneciente a un proceso con una prioridad base de 6.

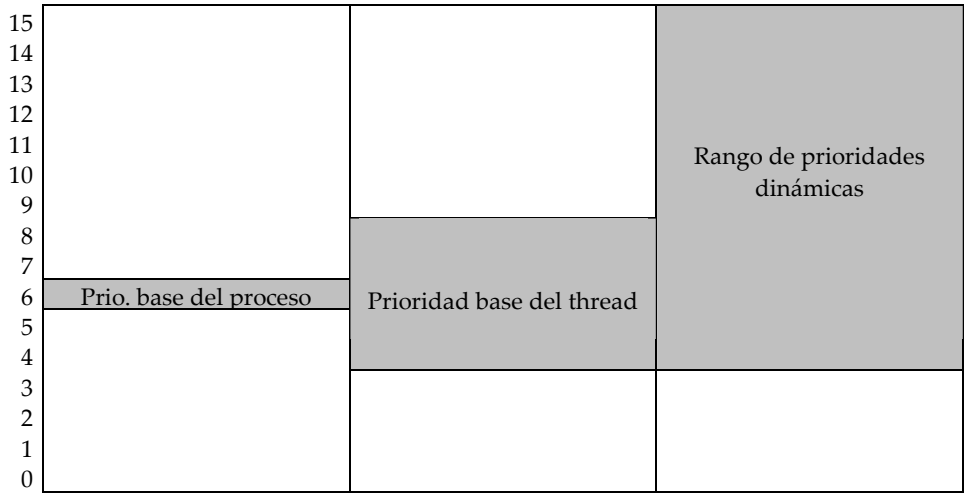

Figura 3.8. Rangos de prioridades para un thread de un proceso con prioridad base 6 en Windows NT.

En un multiprocesador con *N* procesadores, la planificación de threads en Windows 2000 se basa en dos criterios: (1) afinidad al procesador, y (2) asignar *N–1* procesadores a los *N–1* threads más prioritarios y el restante a todos los demás.

Más información puede encontrarse en [SOL00].

3.8 Bibliografía

[STA05] (capítulos 9 y 10) es el texto que mejor se adapta al contenido de este capítulo, incluyendo un breve estudio de la planificación en UNIX y Windows. Otros textos interesantes son: [SIL08], y [TAN08].

Para la planificación en multiprocesadores puede consultarse [STA05] como texto general. [BLA90] es una referencia para quien desee profundizar más en el tema. En cuanto a la planificación en sistemas de tiempo real, si bien hay muchos textos específicos, [SAN05] (capítulo 8) proporciona un buen complemento a lo visto aquí.

La bibliografía sobre UNIX es muy amplia: [BAC86] (capítulo 8) y [LEF89] (capítulo 4) describen la planificación de procesos en los denominados sistemas UNIX *tradicionales* (System V y 4.3BSD respectivamente); [GOO94] (capítulo 4) trata el SVR4, y [VAH96] (capítulo 5) es un compendio de todos ellos. [CUS93, SOL98, SOL00] son las referencias para Windows.

3.9 Ejercicios

1 Dados cuatro programas, se sabe que van a consumir los siguientes tiempos de CPU: A, 9 ms; B, 3 ms; C, 7 ms, y D, 5 ms. Los programas no se bloquean por E/S ni por ninguna otra causa. Considérese un sistema operativo donde los programas se ejecutan en el orden que llegan: A, B, C, D. Si los cuatro llegan en el mismo milisegundo:

(a) ¿Cuál es el tiempo medio de respuesta (latencia) de estos programas en este sistema operativo? ¿Cuál es el tiempo medio de finalización?

(b) ¿En qué orden deberían entrar a ejecutarse para que el tiempo de respuesta fuese el mínimo? Calcular dicho valor mínimo.

(c) ¿Cómo influye sobre el tiempo de respuesta el hecho de modificar la planificación de procesos introduciendo tiempo compartido mediante Round-Robin? Calcular la cota máxima del tiempo de respuesta para (c1) q = 1 ms, y (c2) q = 2 ms.

(d) Considerando una planificación de procesos Round-Robin con quantum de 1 ms, calcular el tiempo medio de respuesta para esos cuatro procesos y su tiempo medio de finalización.

2 En un sistema típico de hoy en día, con un procesador de más de 1 GHz y con un grado de multiprogramación de cerca de 1000 procesos, donde un cambio de contexto supone ejecutar unos 1000 ciclos de reloj extras, comprobar si es posible soportar el concepto de procesador compartido. Para ello, estimar una cota del tiempo de respuesta suficientemente baja y considerar que la pérdida de eficiencia sea razonable.

3 En un sistema de tiempo compartido, la frecuencia de la interrupción de reloj es de 100 Hz. Se han hecho medidas para ajustar el quantum y se ha observado la siguiente distribución de intervalos de CPU:

Menos de 1 tick	25%
Entre 1 tick y 2 ticks	35%
Entre 2 ticks y 3 ticks	35%
Entre 3 ticks y 4 ticks	2%
Más de 4 ticks	3%

siendo el intervalo medio de aproximadamente 2 ticks.

(a) ¿Cuál es un valor de quantum razonable?

(b) Para un tiempo medio de cambio de contexto de 0,02 ms y para el valor de quantum elegido, ¿qué pérdida aproximada de rendimiento (eficiencia) de CPU se produce por el tiempo compartido?

(c) Calcular el tiempo de respuesta medio con este quantum si por término medio hay 10 procesos en la cola de preparados.

(d) Calcular la cota del tiempo de respuesta con grado de multiprogramación 100.

4 Si *t* es el tiempo medio de ejecución de los procesos entre dos operaciones de E/S, discutir los efectos que tiene sobre los parámetros de rendimiento la elección de q, para los casos: (a) q ~ t, (b) q << t, y (c) q >> t.

5 Los cronogramas de la Figura 3.9 representan la ejecución de tres procesos en cuatro sistemas operativos diferentes. (a) Determinar las características de la planificación de procesos correspondientes a cada uno de los cronogramas en cuanto a si es FCFS o por prioridades (estáticas), o si es expulsora (por evento o por tiempo). (b) Calcular en cada caso algunos parámetros del rendimiento del sistema: tiempos medios de respuesta, finalización, y tasas de CPU. Suponer despreciable el tiempo de cambio de contexto.

6 La tabla siguiente muestra un conjunto de procesos en la cola de preparados, identificados según su orden de llegada, con sus respectivos intervalos de CPU y prioridades (mayor valor, mayor prioridad).

Proceso	Intervalo de CPU	Prioridad
1	10	3
2	1	1
3	6	3
4	5	4
5	3	2
6	2	1
7	3	3

Suponiendo que todos los procesos llegan en el mismo tick, dibujar el diagrama de ejecución e indicar el tiempo medio de respuesta en los siguientes casos:

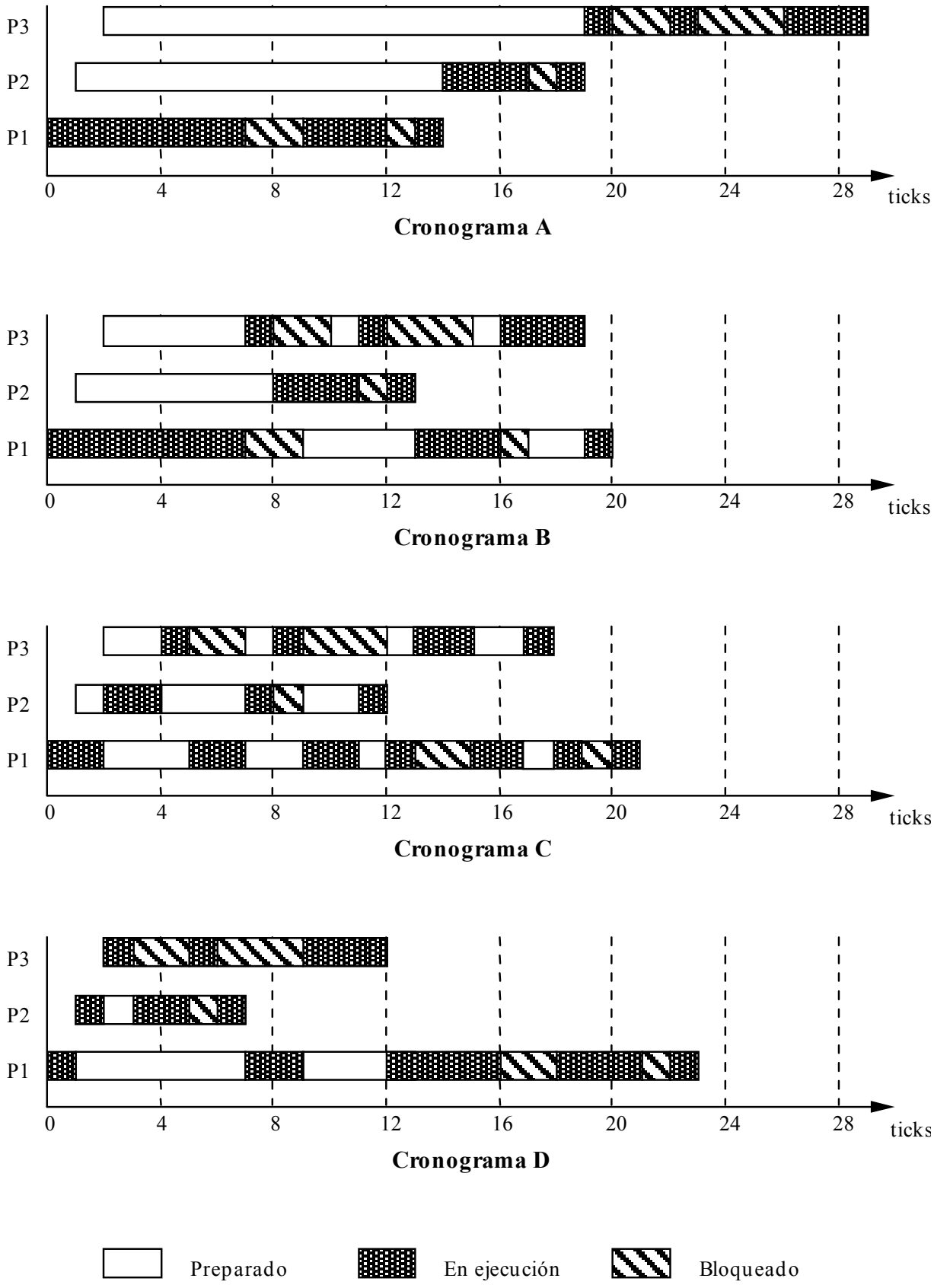

Figura 3.9. Cronogramas para el Ejercicio 5

(a) SJF

(b) Prioridades estáticas

(c) Round-Robin con quantum 2

(d) Prioridades dinámicas, con prioridad inicial la indicada. Cuando agotan el quantum (2 ticks) se les expulsa y disminuye la prioridad en 1.

(e) Multinivel, considerando los procesos 2, 3, 5, 6 interactivos y los 1, 4, 7 batch. De cada 10 ticks los 6 primeros son para procesos interactivos (con gestión RR y q=1) y el resto para batch (con política FCFS).

7 La tabla muestra un conjunto de procesos llegados a la cola de preparados, indicando para cada uno su identificador, el tick de llegada, el intervalo de CPU que necesita, el tiempo que estará bloqueado cada vez, y su prioridad (mayor valor, mayor prioridad). Nótese que hay dos tipos de procesos: los que, agotado su intervalo de CPU, terminan (el 1 y el 6), y los que cíclicamente se bloquean durante un tiempo y vuelven a la CPU (el resto). Estos últimos no terminan nunca.

Proceso	Tick llegada	Intervalo de CPU	Tiempo bloqueado	Prioridad
1	0	4	(termina)	3
2	0	1	6	1
3	10	6	5	3
4	15	5	4	4
5	17	3	3	2
6	20	2	(termina)	1
7	20	3	10	3

(a) Dibujar el cronograma de ejecución durante los primeros 40 ciclos para planificación de tiempo compartido combinado con prioridad y quantum 2 (a1) sin expulsión por evento, y (a2) con expulsión por evento.

(b) Calcular en cada caso el porcentaje del tiempo de CPU que no hay nadie en ejecución.

8 En un sistema de tiempo real con las siguientes tareas (en orden decreciente de prioridad):

Tarea	T. de ejecución	Periodo
T1	5	10
T2	8	20
T3	5	55

(a) ¿Existe una planificación de tiempo real viable? Razona la respuesta.

(b) Dibuja el cronograma resultante de aplicar planificación (1) RM, y (2) EDF. Comprobar en ambos casos si se cumplen siempre los plazos.

9 Considerar un sistema donde los procesos acceden a los recursos compartidos en exclusión mutua utilizando cerrojos de espera activa mediante primitivas *lock()* y *unlock()*. El sistema operativo utiliza planificación con expulsión por la llegada de un proceso a la cola de preparados y prioridades estáticas. Dos procesos, A y B, se ejecutan en dicho sistema. El proceso A se lanza en el instante de tiempo T con prioridad 7 y el proceso B se lanza en el instante T+4 ms con prioridad 9 (mayor valor, mayor prioridad). Aparte del proceso nulo, no hay otros procesos en el sistema. El código que ejecutan los procesos es el siguiente:

Proceso A

```
calculo_A1();   -- 3 ms de CPU
lock(mutex);
usar_SC();      -- 3 ms de CPU
unlock(mutex);
calculo_A2();   -- 5 ms de CPU
```

Proceso B

```
calculo_B1();   -- 5 ms de CPU
lock(mutex);
usar_SC();      -- 3 ms de CPU
unlock(mutex);
calculo_B2();   -- 6 ms de CPU
```

(a) Dibujar el diagrama de ejecución, indicando también cuándo se está ejecutando la sección crítica de código.

(b) Considerar ahora que la ejecución se produce en un sistema operativo con prioridades dinámicas y expulsión por tiempo (además de por evento) y quantum de 5 ms. Cuando un proceso cumple su quantum, su prioridad se decrementa en 1. Considerando que el resto de las condiciones son las mismas que las del primer apartado, dibujar ahora el cronograma de la nueva ejecución.