Lección 5

Procesos e hilos

2.4 PLANIFICACIÓN

- Cuando una computadora se multiprograma, con frecuencia tiene varios procesos o hilos que compiten por la CPU al mismo tiempo. Esta situación ocurre cada vez que dos o más de estos procesos se encuentran al mismo tiempo en el estado listo. Si sólo hay una CPU disponible, hay que decidir cuál proceso se va a ejecutar a continuación. La parte del sistema operativo que realiza esa decisión se conoce como **planificador de procesos** y el algoritmo que utiliza se conoce como **algoritmo de planificación**.
- Muchas de las mismas cuestiones que se aplican a la planificación de procesos también se aplican a la planificación de hilos, aunque algunas son distintas. Cuando el kernel administra hilos, por lo general la planificación se lleva a cabo por hilo, e importa muy poco (o nada) a cuál proceso pertenece ese hilo.

2.4.1 Introducción a la planificación

- La planificación no es tan importante en PCs, pero en servidores o equipos en red solicitando servicios sí.
- Debido a que el tiempo de la CPU es un recurso escaso en estas máquinas, un buen planificador puede hacer una gran diferencia en el rendimiento percibido y la satisfacción del usuario. En consecuencia, se ha puesto gran esfuerzo en idear algoritmos de planificación astutos y eficientes.
- La conmutación de procesos es muy cara y consume mucho CPU, por consiguiente tiempo (debe pasar de modo usuario a modo kernel).

Cuándo planificar procesos

- En primer lugar, cuando se crea un nuevo proceso se debe tomar una decisión en cuanto a si se debe ejecutar el proceso padre o el proceso hijo..
- En segundo lugar, se debe tomar una decisión de planificación cuando un proceso termina.
- En tercer lugar, cuando un proceso se bloquea por esperar una operación de E/S, un semáforo o por alguna otra razón, hay que elegir otro proceso para ejecutarlo.
- En cuarto lugar, cuando ocurre una interrupción de E/S tal vez haya que tomar una decisión de planificación. Si la interrupción proviene de un dispositivo de E/S que ha terminado su trabajo, tal vez ahora un proceso que haya estado bloqueado en espera de esa operación de E/S esté listo para ejecutarse. Es responsabilidad del planificador decidir si debe ejecutar el proceso que acaba de entrar al estado listo, el proceso que se estaba ejecutando al momento de la interrupción, o algún otro.
- Los algoritmos de planificación se pueden dividir en dos categorías con respecto a la forma en que manejan las interrupciones del reloj.
  - Un algoritmo de programación **no apropiativo** (*nonpreemptive*) selecciona un proceso para ejecutarlo y después sólo deja que se ejecute hasta que el mismo se bloquea (ya sea en espera de una operación de E/S o de algún otro proceso) o hasta que libera la CPU en forma voluntaria.
  - Por el contrario, un algoritmo de planificación **apropiativa** selecciona un proceso y deja que se ejecute por un máximo de tiempo fijo. Si sigue en ejecución al final del intervalo de tiempo, se suspende y el planificador selecciona otro proceso para ejecutarlo (si hay uno disponible).

Categorías de los algoritmos de planificación

- De acuerdo al objetivo del SO:
- 1. Procesamiento por lotes: Se puede utilizar un algoritmo no apropiativo, o apropiativo con bastante asignación de tiempo.
- 2. Interactivo: el ideal es el apropiativo, para evitar que un usuario o proceso acapare la CPU por un periodo muy largo.
- 3. De tiempo real: la apropiación no es necesaria, ya que están diseñando para hacer la tarea y bloquearse.

Metas de los algoritmos de planificación

Todos los sistemas

- Equidad - Otorgar a cada proceso una parte justa de la CPU
- Aplicación de políticas - Verificar que se lleven a cabo las políticas establecidas
- Balance - Mantener ocupadas todas las partes del sistema

Sistemas de procesamiento por lotes

- Rendimiento - Maximizar el número de trabajos por hora
- Tiempo de retorno - Minimizar el tiempo entre la entrega y la terminación
- Utilización de la CPU - Mantener ocupada la CPU todo el tiempo

Sistemas interactivos

- Tiempo de respuesta - Responder a las peticiones con rapidez
- Proporcionalidad - Cumplir las expectativas de los usuarios

Sistemas de tiempo real

- Cumplir con los plazos - Evitar perder datos
- Predictibilidad - Evitar la degradación de la calidad en los sistemas multimedia

Figura 2-39. Algunas metas del algoritmo de planificación bajo distintas circunstancias.

2.4.2 Planificación en sistemas de procesamiento por lotes

- **Primero en entrar, primero en ser atendido**
- El más simple de todos los algoritmos de planificación. (FCFS, *First-Come, First-Served*) no apropiativo. Con este algoritmo, la CPU se asigna a los procesos en el orden en el que la solicitan. En esencia hay una sola cola de procesos listos. Cuando el primer trabajo entra al sistema, se inicia de inmediato y se le permite ejecutarse todo el tiempo que desee. No se interrumpe debido a que se ha ejecutado demasiado tiempo. A medida que van entrando otros trabajos, se colocan al final de la cola. Si el proceso en ejecución se bloquea, el primer proceso en la cola se ejecuta a continuación.
- Cuando un proceso bloqueado pasa al estado listo, al igual que un trabajo recién llegado, se coloca al final de la cola.
- La gran fuerza de este algoritmo es que es fácil de comprender e igualmente sencillo de programar.
- Por desgracia, el algoritmo tipo “primero en entrar, primero en ser atendido” también tiene una importante desventaja. Cuál es?

El trabajo más corto primero

- Es no apropiativo, supone que los tiempos de ejecución se conocen de antemano. Cuando hay varios trabajos de igual importancia esperando a ser iniciados en la cola de entrada, el planificador selecciona **el trabajo más corto primero** (SJF, *Shortest Job First***)**. Analicemos la figura 2-40. Cuadro a tarda 14 minutos en promedio, cuadro b tarda 11 minutos en promedio.

Figura 2-40. Un ejemplo de planificación tipo el trabajo más corto primero. (a) Ejecución de cuatro trabajos en el orden original. (b) Ejecución de cuatro trabajos en el orden del tipo "el trabajo más corto primero".

El menor tiempo restante a continuación

- Una versión apropiativa del algoritmo tipo el trabajo más corto primero es **el menor tiempo restante a continuación** (SRTN, *Shortest Remaining Time Next*). Con este algoritmo, el planificador siempre selecciona el proceso cuyo tiempo restante de ejecución sea el más corto. De nuevo, se debe conocer el tiempo de ejecución de antemano. Cuando llega un nuevo trabajo, su tiempo total se compara con el tiempo restante del proceso actual. Si el nuevo trabajo necesita menos tiempo para terminar que el proceso actual, éste se suspende y el nuevo trabajo se inicia. Ese esquema permite que los trabajos cortos nuevos obtengan un buen servicio.

2.4.3 Planificación en sistemas interactivos

- **Planificación por turno circular**
- Uno de los algoritmos más antiguos, simples, equitativos y de mayor uso es el de **turno circular** (*round-robin*). A cada proceso se le asigna un intervalo de tiempo, conocido como **quántum**, durante el cual se le permite ejecutarse. Si el proceso se sigue ejecutando al final del quántum, la CPU es apropiada para dársela a otro proceso. Si el proceso se bloquea o termina antes de que haya transcurrido el quántum, la conmutación de la CPU se realiza cuando el proceso se bloquea, desde luego.
- Es fácil implementar el algoritmo de turno circular.
- La única cuestión interesante con el algoritmo de turno circular es la longitud del quántum.
- Para conmutar de un proceso a otro se requiere cierta cantidad de tiempo para realizar la administración: guardar y cargar tanto registros como mapas de memoria, actualizar varias tablas y listas, vaciar y recargar la memoria caché y así sucesivamente. Suponga que esta **conmutación de proceso** o **conmutación de contexto** (como algunas veces se le llama) requiere 1 mseg, incluyendo el cambio de los mapas de memoria, el vaciado y recarga de la caché, etc. Suponga además que el quántum se establece a 4 mseg. Con estos parámetros, después de realizar 4 mseg de trabajo útil, la CPU tendrá que gastar (es decir, desperdiciar) 1 mseg en la conmutación de procesos. Por ende, 20 por ciento de la CPU se desperdiciará por sobrecarga administrativa.
- Qué sucede si se establece el quántum a 100 mseg?.

Planificación por prioridad

- La idea básica es simple: a cada proceso se le asigna una prioridad y el proceso ejecutable con la prioridad más alta es el que se puede ejecutar.
- Para evitar que los procesos con alta prioridad se ejecuten de manera indefinida, el planificador puede reducir la prioridad del proceso actual en ejecución en cada pulso del reloj (es decir, en cada interrupción del reloj). Si esta acción hace que su prioridad se reduzca a un valor menor que la del proceso con la siguiente prioridad más alta, ocurre una conmutación de procesos.
- De manera alternativa, a cada proceso se le puede asignar un quántum de tiempo máximo que tiene permitido ejecutarse. Cuando este quántum se utiliza, el siguiente proceso con la prioridad más alta recibe la oportunidad de ejecutarse.
- Qué es la asignación de prioridad estática y la dinámica?

Múltiples colas

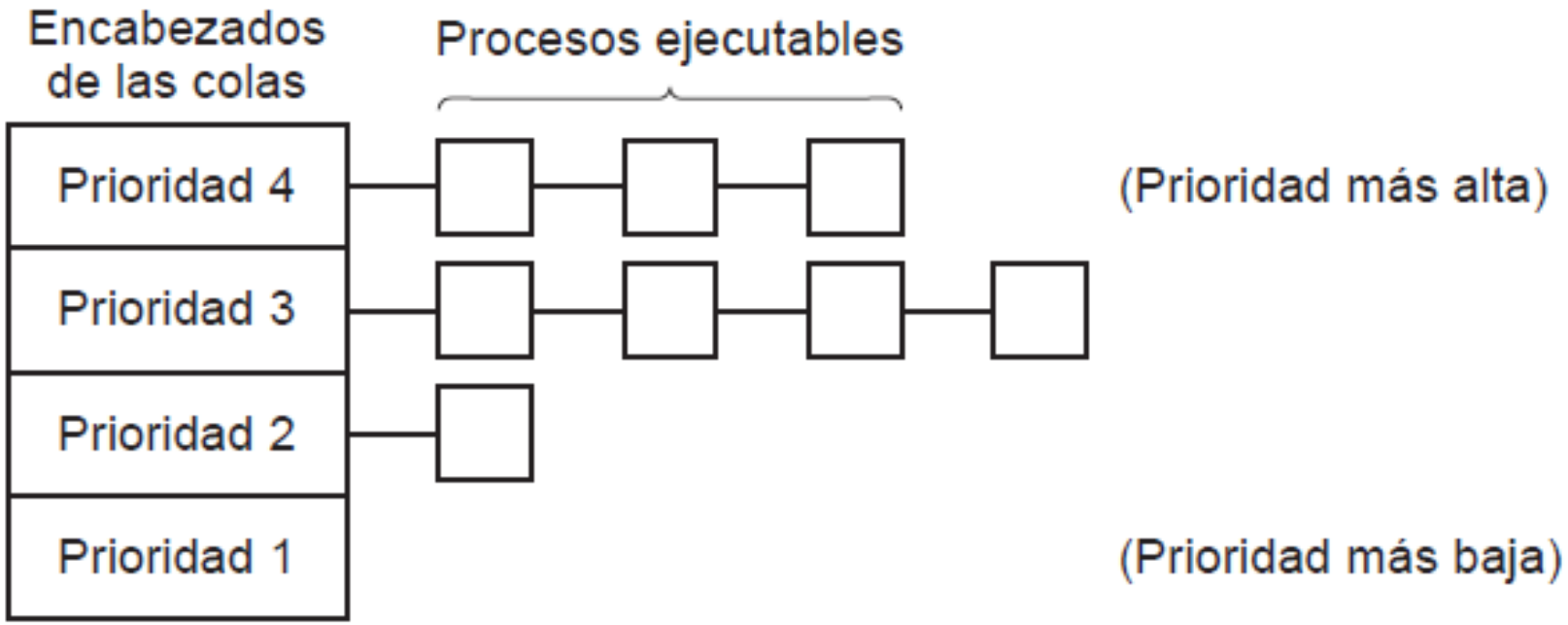

Figura 2-42. Un algoritmo de planificación con cuatro clases de prioridad.

El proceso más corto a continuación

- Por lo general, los procesos interactivos siguen el patrón de esperar un comando, ejecutarlo, esperar un comando, ejecutarlo, etcétera. Si consideramos la ejecución de cada comando como un “trabajo” separado, entonces podríamos minimizar el tiempo de respuesta total mediante la ejecución del más corto primero. El único problema es averiguar cuál de los procesos actuales ejecutables es el más corto.
- Un método es realizar estimaciones con base en el comportamiento anterior y ejecutar el proceso con el tiempo de ejecución estimado más corto.
- Suponga que el tiempo estimado por cada comando para cierta terminal es *T0*. Ahora suponga que su siguiente ejecución se mide como *T1*.
- Podríamos actualizar nuestra estimación mediante una suma ponderada de estos dos números, es decir, aT_0 (1 - a)T_1. Por medio de la elección de a podemos decidir hacer que el proceso de estimación olvide las ejecuciones anteriores rápidamente o que las recuerde por mucho tiempo. Con a 1/2, obtenemos estimaciones sucesivas de
    - $T_0, T_0/2 + T_1/2, T_0/4 + T_1/4 + T_2/2, T_0/8 + T_1/8 + T_2/4 + T_3/2$

Planificación garantizada

- Un método completamente distinto para la planificación es hacer promesas reales a los usuarios acerca del rendimiento y después cumplirlas.
- Una de ellas, que es realista y fácil de cumplir es: si hay *n* usuarios conectados mientras usted está trabajando, recibirá aproximadamente 1/*n* del poder de la CPU. De manera similar, en un sistema de un solo usuario con *n* procesos en ejecución, mientras no haya diferencias, cada usuario debe obtener 1/*n* de los ciclos de la CPU.
- Para cumplir esta promesa, el sistema debe llevar la cuenta de cuánta potencia de CPU ha tenido cada proceso desde su creación. Después calcula cuánto poder de la CPU debe asignarse a cada proceso, a saber el tiempo desde que se creó dividido entre *n*. Como la cantidad de tiempo de CPU que ha tenido cada proceso también se conoce, es simple calcular la proporción de tiempo de CPU que se consumió con el tiempo de CPU al que cada proceso tiene derecho. Una proporción de 0.5 indica que un proceso solo ha tenido la mitad del tiempo que debería tener, y una proporción de 2.0 indica que un proceso ha tenido el doble de tiempo del que debería tener. Entonces, el algoritmo es para ejecutar el proceso con la menor proporción hasta que se haya desplazado por debajo de su competidor más cercano.

Planificación por sorteo

- La idea básica es dar a los procesos boletos de lotería para diversos recursos del sistema, como el tiempo de la CPU. Cada vez que hay que tomar una decisión de planificación, se selecciona un boleto de lotería al azar y el proceso que tiene ese boleto obtiene el recurso. Cuando se aplica a la planificación de la CPU, el sistema podría realizar un sorteo 50 veces por segundo y cada ganador obtendría 20 mseg de tiempo de la CPU como premio.
- Los procesos más importantes pueden recibir boletos adicionales, para incrementar su probabilidad de ganar. En contraste a un planificador por prioridad, en donde es muy difícil establecer lo que significa tener una prioridad de 40, aquí la regla es clara: un proceso que contenga una fracción *f* de los boletos recibirá aproximadamente una fracción *f* del recurso en cuestión.
- La planificación por lotería tiene varias propiedades interesantes. Por ejemplo, si aparece un nuevo proceso y recibe algunos boletos, en el siguiente sorteo tendrá la oportunidad de ganar en proporción al número de boletos que tenga. En otras palabras, la planificación por lotería tiene un alto grado de respuesta.
- Los procesos cooperativos pueden intercambiar boletos si lo desean.

Planificación por partes equitativas

- Hasta ahora hemos asumido que cada proceso se planifica por su cuenta, sin importar quién sea su propietario. Como resultado, si el usuario 1 inicia 9 procesos y el usuario 2 inicia 1 proceso, con la planificación por turno circular o por prioridades iguales, el usuario 1 obtendrá 90 por ciento del tiempo de la CPU y el usuario 2 sólo recibirá 10 por ciento.
- Para evitar esta situación, algunos sistemas toman en consideración quién es el propietario de un proceso antes de planificarlo. En este modelo, a cada usuario se le asigna cierta fracción de la CPU y el planificador selecciona procesos de tal forma que se cumpla con este modelo. Por ende, si a dos usuarios se les prometió 50 por ciento del tiempo de la CPU para cada uno, eso es lo que obtendrán sin importar cuántos procesos tengan en existencia.
- Como ejemplo, considere un sistema con dos usuarios, y a cada uno de los cuales se les prometió 50 por ciento de la CPU. El usuario 1 tiene cuatro procesos (*A*, *B*, *C* y *D*) y el usuario 2 sólo tiene 1 proceso (*E*). Si se utiliza la planificación por turno circular, una posible secuencia de planificación que cumple con todas las restricciones es:
  - AE B E C E D E A E B E C E D E …
- Por otro lado, si el usuario 1 tiene derecho al doble de tiempo de la CPU que el usuario 2, podríamos obtener la siguiente secuencia:
  - AB E C D E A B E C D E …
- Desde luego que existen muchas otras posibilidades, y se pueden explotar dependiendo de cuál sea la noción de equidad.

2.4.4 Planificación en sistemas de tiempo real

- En un sistema de **tiempo real**, el tiempo desempeña un papel esencial. Por ejemplo, la computadora en un reproductor de disco compacto recibe los bits a medida que provienen de la unidad y debe convertirlos en música, en un intervalo de tiempo muy estrecho. Si el cálculo tarda demasiado, la música tendrá un sonido peculiar. Otros sistemas de tiempo real son el monitoreo de pacientes en una unidad de cuidados intensivos de un hospital, el autopiloto en una aeronave y el control de robots en una fábrica automatizada. En todos estos casos, tener la respuesta correcta pero demasiado tarde es a menudo tan malo como no tenerla.
- En general, los sistemas de tiempo real se categorizan como de **tiempo real duro**, lo cual significa que hay tiempos límite absolutos que se deben cumplir, y como de **tiempo real suave**, lo cual significa que no es conveniente fallar en un tiempo límite en ocasiones, pero sin embargo es tolerable.

- Los eventos a los que puede llegar a responder un sistema de tiempo real se pueden categorizar como **periódicos** (que ocurren a intervalos regulares) o **aperiódicos** (que ocurren de manera impredecible).
- Tal vez un sistema tenga que responder a varios flujos de eventos periódicos. Dependiendo de cuánto tiempo requiera cada evento para su procesamiento, tal vez ni siquiera sea posible manejarlos a todos. Por ejemplo, si hay *m* eventos periódicos y el evento *i* ocurre con el periodo *Pi* y requiere *Ci* segundos de tiempo de la CPU para manejar cada evento, entonces la carga sólo se podrá manejar si

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

Como ejemplo, considere un sistema de tiempo real con tres eventos periódicos, con periodos de 100, 200 y 500 mseg, respectivamente. Si estos eventos requieren 50, 30 y 100 mseg de tiempo de la CPU por evento, respectivamente.

Cuántos segundos necesita para los 3 procesos?

2.4.5 Política contra mecanismo

- Algunas veces sucede que un proceso tiene muchos hijos ejecutándose bajo su control. Por ejemplo, un proceso de un sistema de administración de bases de datos puede tener muchos hijos. Cada hijo podría estar trabajando en una petición distinta (análisis de consultas o acceso al disco, por ejemplo). Es posible que el proceso principal tenga una idea excelente acerca de cuál de sus hijos es el más importante y cuál es el menos. Por desgracia, ninguno de los planificadores antes descritos acepta entrada de los procesos de usuario acerca de las decisiones de planificación. Como resultado, raras veces el planificador toma la mejor decisión.
- La solución a este problema es separar el **mecanismo de planificación** de la **política de planificación**, un principio establecido desde hace tiempo (Levin y colaboradores, 1975). Esto significa que el algoritmo de planificación está parametrizado de cierta forma, pero los procesos de usuario pueden llenar los parámetros. Consideremos el ejemplo de la base de datos una vez más.
- Suponga que el kernel utiliza un algoritmo de planficación por prioridad, pero proporciona una llamada al sistema mediante la cual un proceso puede establecer (y modificar) las prioridades de sus hijos. De esta forma, el padre puede controlar con detalle la forma en que se planifican sus hijos, aun y cuando éste no se encarga de la planificación. Aquí el mecanismo está en el kernel, pero la política se establece mediante un proceso de usuario.

2.4.6 Planificación de hilos

- Cuando varios procesos tienen múltiples hilos cada uno, tenemos dos niveles de paralelismo presentes: procesos e hilos. La planificación en tales sistemas difiere en forma considerable, dependiendo de si hay soporte para hilos a nivel usuario o para hilos a nivel kernel (o ambos).
- **Los hilos a nivel usuario:** Como el kernel no está consciente de la existencia de los hilos, opera en la misma forma de siempre: selecciona un proceso, por decir *A*, y otorga a este proceso el control de su quántum. El planificador de hilos dentro de *A* decide cuál hilo ejecutar, por decir *A1*. Como no hay interrupciones de reloj para multiprogramar hilos, este hilo puede continuar ejecutándose todo el tiempo que quiera. Si utiliza todo el quántum del proceso, el kernel seleccionará otro proceso para ejecutarlo.
- Cuando el proceso *A* por fin se ejecute de nuevo, el hilo *A1* continuará su ejecución. Seguirá consumiendo todo el tiempo de *A* hasta que termine. Sin embargo, su comportamiento antisocial no afectará a los demás procesos. Estos recibirán lo que el planificador de procesos considere que es su parte apropiada, sin importar lo que esté ocurriendo dentro del proceso *A*.

- Considere el caso en el que los hilos de *A* tienen relativamente poco trabajo que realizar por cada ráfaga de la CPU; por ejemplo, 5 mseg de trabajo dentro de un quántum de 50 mseg. En consecuencia, cada uno se ejecuta por unos instantes y después entrega la CPU al planificador de hilos. Esto podría producir la secuencia *A1*, *A2*, *A3*, *A1*, *A2*, *A3*, *A1*, *A2*, *A3*, *A1* antes de que el kernel conmute al proceso *B*. Esta situación se ilustra en la figura 2-43(a).
- Ahora considere la situación con hilos a nivel kernel. Aquí el kernel selecciona un hilo específico para ejecutarlo. No tiene que tomar en cuenta a cuál proceso pertenece el hilo, pero puede hacerlo si lo desea. El hilo recibe un quántum y se suspende obligatoriamente si se excede de este quántum. Con un quántum de 50 mseg pero hilos que se bloquean después de 5 mseg, el orden de los hilos para cierto periodo de 30 mseg podría ser *A1*, *B1*, *A2*, *B2*, *A3*, *B3*, algo que no sería posible con estos parámetros e hilos a nivel usuario. Esta situación se ilustra en forma parcial en la figura 2-43(b).
- Una diferencia importante entre los hilos a nivel usuario y los hilos a nivel kernel es el rendimiento. Para realizar un conmutación de hilos con hilos a nivel usuario se requiere de muchas instrucciones de máquina. Con hilos a nivel kernel se requiere una conmutación de contexto total, cambiar el mapa de memoria e invalidar la caché, lo cual es varias órdenes de magnitud más lento. Por otro lado, con los hilos a nivel kernel, cuando un hilo se bloquea en espera de E/S no se suspende todo el proceso, como con los hilos a nivel usuario.

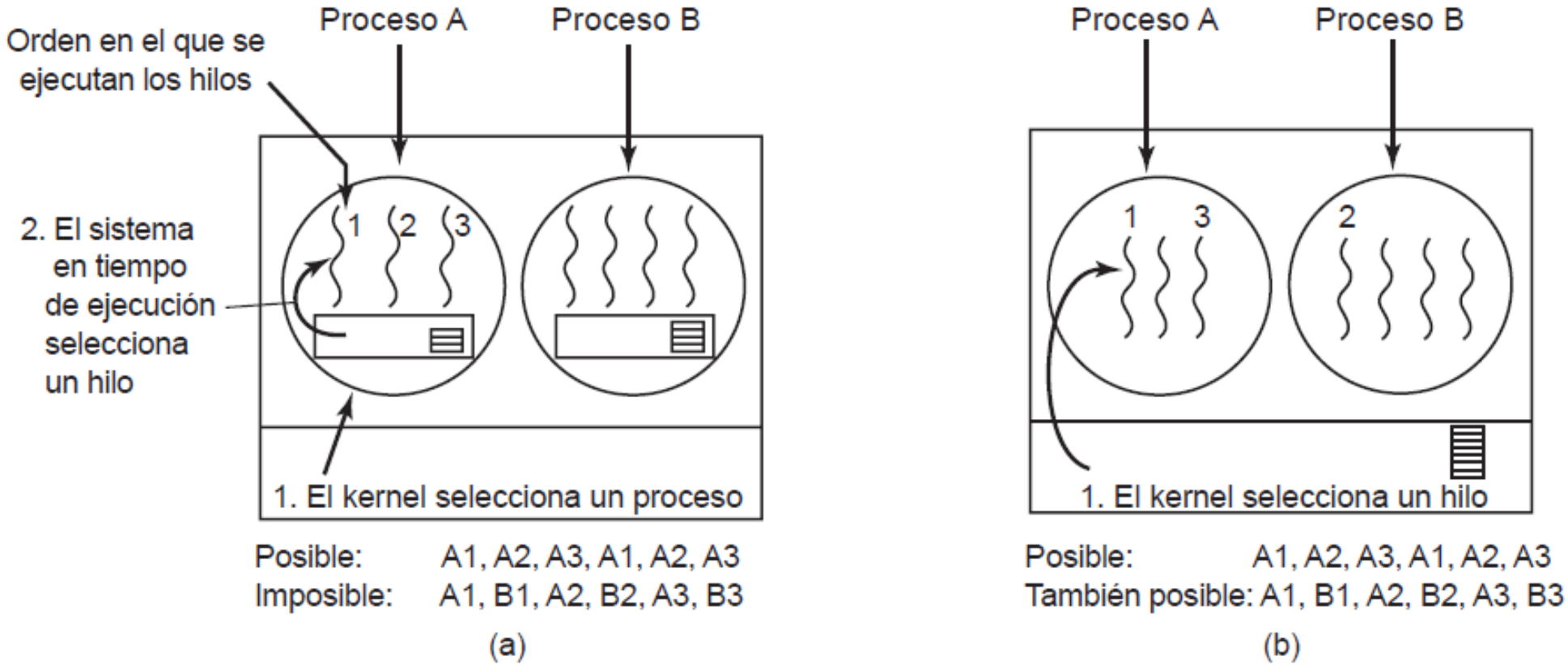

Figura 2-43. (a) Posible planificación de hilos a nivel usuario con un quántum de 50 mseg para cada proceso e hilos que se ejecutan durante 5 mseg por cada ráfaga de la CPU. (b) Posible planificación de hilos a nivel kernel con las mismas características que (a).

2.5 PROBLEMAS CLÁSICOS DE COMUNICACIÓN ENTRE PROCESOS (IPC)

• 2.5.1 El problema de los filósofos comelones

- Cinco filósofos están sentados alrededor de una mesa circular. Cada filósofo tiene un plato de espagueti. El espagueti es tan resbaloso, que un filósofo necesita dos tenedores para comerlo. Entre cada par de platos hay un tenedor. La distribución de la mesa se ilustra en la figura 2-44.

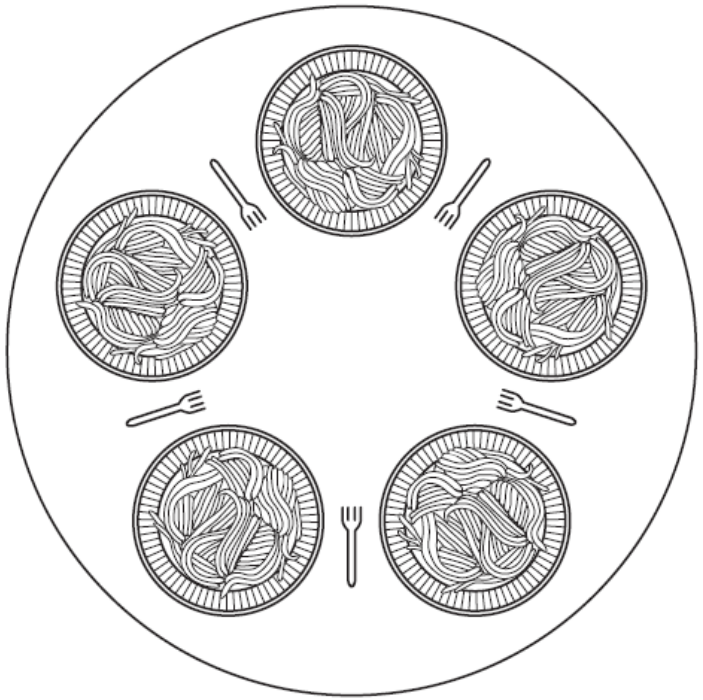

Figura 2-44. Hora de comer en el Departamento de Filosofía.

Cuando un filósofo tiene hambre, trata de adquirir sus tenedores izquierdo y derecho, uno a la vez, en cualquier orden. Si tiene éxito al adquirir dos tenedores, come por un momento, después deja los tenedores y continúa pensando.

La pregunta clave es: **¿puede usted escribir un programa para cada filósofo, que haga lo que se supone debe hacer y nunca se trabe?**

2.5.2 El problema de los lectores y escritores

- Modela el acceso a una base de datos.
- Por ejemplo, imagine un sistema de reservación de aerolíneas, con muchos procesos en competencia que desean leer y escribir en él. Es aceptable tener varios procesos que lean la base de datos al mismo tiempo, pero si un proceso está actualizando (escribiendo) la base de datos, ningún otro proceso puede tener acceso a la base de datos, ni siquiera los lectores. La pregunta es, ¿cómo se programan los lectores y escritores? Una solución se muestra en la figura 2-47.
- En esta solución, el primer lector en obtener acceso a la base de datos realiza una operación down en el semáforo *bd*. Los siguientes lectores simplemente incrementan un contador llamado *cl*.
- A medida que los lectores van saliendo, decrementan el contador y el último realiza una operación up en el semáforo, para permitir que un escritor bloqueado (si lo hay) entre.
- La solución que se presenta aquí contiene en forma implícita una decisión sutil que vale la pena observar. Suponga que mientras un lector utiliza la base de datos, llega otro lector. Como no es un problema tener dos lectores al mismo tiempo, el segundo lector es admitido. También se pueden admitir más lectores, si es que llegan.
- Ahora suponga que aparece un escritor. Tal vez éste no sea admitido a la base de datos, ya que los escritores deben tener acceso exclusivo y por ende, el escritor se suspende. Más adelante aparecen lectores adicionales. Mientras que haya un lector activo, se admitirán los siguientes lectores.
- Como consecuencia de esta estrategia, mientras que haya un suministro continuo de lectores, todos entrarán tan pronto lleguen. El escritor estará suspendido hasta que no haya un lector presente. Si llega un nuevo lector, por decir cada 2 segundos y cada lector requiere 5 segundos para hacer su trabajo, el escritor nunca entrará.
- Para evitar esta situación, el programa se podría escribir de una manera ligeramente distinta:
- cuando llega un lector y hay un escritor en espera, el lector se suspende detrás del escritor, en vez de ser admitido de inmediato. De esta forma, un escritor tiene que esperar a que terminen los lectores que estaban activos cuando llegó, pero no tiene que esperar a los lectores que llegaron después de él. La desventaja de esta solución es que logra una menor concurrencia y por ende, un menor rendimiento. Courtois y sus colaboradores presentan una solución que da prioridad a los escritores.