PROCESOS E HILOS

- PROCESOS

En cualquier sistema de multiprogramación, la CPU conmuta de un proceso a otro con rapidez, ejecutando cada uno durante décimas o centésimas de milisegundos: hablando en sentido estricto, en cualquier instante la CPU está ejecutando sólo un proceso, y en el transcurso de 1 segundo podría trabajar en varios de ellos, dando la apariencia de un paralelismo (o pseudoparalelismo, para distinguirlo del verdadero paralelismo de hardware de los sistemas multiprocesadores con dos o más CPUs que comparten la misma memoria física)

1. El modelo del proceso

1.1. Todo el software ejecutable en la computadora, que algunas veces incluye al sistema operativo, se organiza en varios procesos secuenciales. Un proceso no es más que una instancia de un programa en ejecución, incluyendo los valores actuales del contador de programa, los registros y las variables.

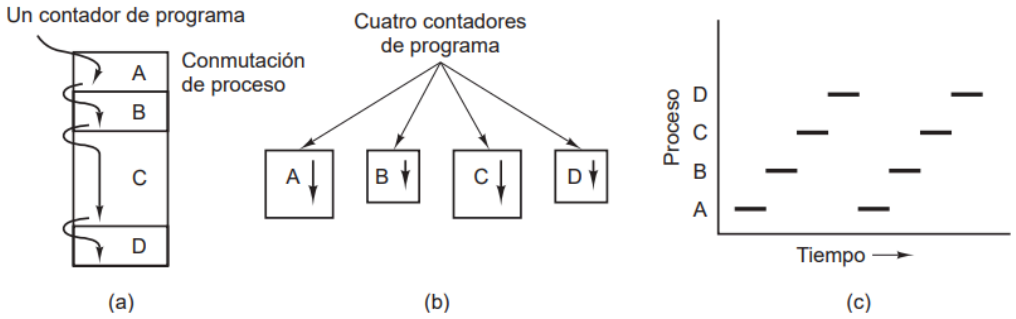


Figura 2-1. (a) Multiprogramación de cuatro programas. (b) Modelo conceptual de cuatro procesos secuenciales independientes. (c) Sólo hay un programa activo a la vez.

2. Creación de un proceso

2.1. Hay cuatro eventos principales que provocan la creación de procesos:

1. El arranque del sistema.

2. La ejecución, desde un proceso, de una llamada al sistema para creación de procesos.

3. Una petición de usuario para crear un proceso.

4. El inicio de un trabajo por lotes.

2.2. Generalmente, cuando se arranca un sistema operativo se crean varios procesos. Algunos de ellos son procesos en primer plano; es decir, procesos que interactúan con los usuarios (humanos) y realizan trabajo para ellos. Otros son procesos en segundo plano, que no están asociados con usuarios

específicos sino con una función específica. Los procesos que permanecen en segundo plano para manejar ciertas actividades como correo electrónico, páginas Web, noticias, impresiones, etcétera, se conocen como demonios (daemons).

2.3. En UNIX sólo hay una llamada al sistema para crear un proceso: fork. Esta llamada crea un clon exacto del proceso que hizo la llamada. Después de fork, los dos procesos (padre e hijo) tienen la misma imagen de memoria, las mismas cadenas de entorno y los mismos archivos abiertos

2.4. En Windows una sola llamada a una función de Win32 (CreateProcess) maneja la creación de procesos y carga el programa correcto en el nuevo proceso. Esta llamada tiene 10 parámetros, que incluyen el programa a ejecutar, los parámetros de la línea de comandos para introducir datos a ese programa, varios atributos de seguridad, bits que controlan si los archivos abiertos se heredan, información de prioridad, una especificación de la ventana que se va a crear para el proceso (si se va a crear una) y un apuntador a una estructura en donde se devuelve al proceso que hizo la llamada la información acerca del proceso recién creado.

3. Terminación de procesos

3.1. Tarde o temprano el nuevo proceso terminará, por lo general debido a una de las siguientes condiciones:

3.1.1. Salida normal (voluntaria): Para indicar al sistema operativo que ha terminado. Esta llamada es exit en UNIX y ExitProcess en Windows. Los programas orientados a pantalla también admiten la terminación voluntaria.

3.1.2. Salida por error (voluntaria): Si un usuario escribe el comando para compilar el programa foo.c y no existe dicho archivo, el compilador simplemente termina. Los procesos interactivos orientados a pantalla por lo general no terminan cuando reciben parámetros incorrectos. En vez de ello, aparece un cuadro de diálogo y se le pide al usuario que intente de nuevo.

3.1.3. Error fatal (involuntaria): El ejecutar una instrucción ilegal, hacer referencia a una parte de memoria no existente o la división entre cero.

3.1.4. Eliminado por otro proceso (involuntaria): Es que ejecute una llamada al sistema que indique al sistema operativo que elimine otros procesos. En UNIX esta llamada es kill. La función correspondiente en Win32 es TerminateProcess. En ambos casos, el proceso eliminador debe tener la autorización necesaria para realizar la eliminación.

4. Jerarquías de procesos

4.1. En algunos sistemas, cuando un proceso crea otro, el proceso padre y el proceso hijo continúan asociados en ciertas formas. El proceso hijo puede crear por sí mismo más procesos, formando una jerarquía de procesos.

4.2. Como otro ejemplo dónde la jerarquía de procesos juega su papel, veamos la forma en que UNIX se inicializa a sí mismo cuando se enciende la computadora. Hay un proceso especial (llamado init) en la imagen de inicio. Cuando empieza a ejecutarse, lee un archivo que le indica cuántas terminales hay. Después utiliza fork para crear un proceso por cada terminal. Estos procesos esperan a que alguien inicie la sesión. Si un inicio de sesión tiene éxito, el proceso de inicio de sesión ejecuta un shell para aceptar comandos. Éstos pueden iniciar más procesos y así sucesivamente.

4.3. Windows no tiene un concepto de una jerarquía de procesos. Todos los procesos son iguales. La única sugerencia de una jerarquía de procesos es que, cuando se crea un proceso, el padre recibe un indicador especial un token (llamado manejador) que puede utilizar para controlar al hijo.

5. Estados de un proceso

5.1. A los tres estados en los que se puede encontrar un proceso:

5.1.1. En ejecución (en realidad está usando la CPU en ese instante).

5.1.2. Listo (ejecutable; se detuvo temporalmente para dejar que se ejecute otro proceso).

5.1.3. Bloqueado (no puede ejecutarse sino hasta que ocurra cierto evento externo).

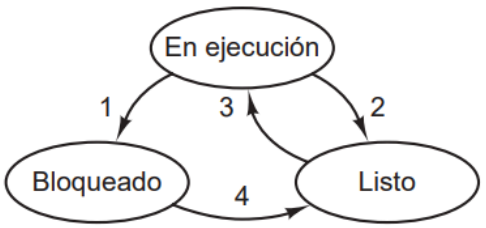


Figura 2-2. Un proceso puede encontrarse en estado "en ejecución", "bloqueado" o "listo". Las transiciones entre estos estados son como se muestran.

6. Implementación de los procesos

6.1. Para implementar el modelo de procesos, el sistema operativo mantiene una tabla (un arreglo de estructuras) llamada tabla de procesos, con sólo una entrada por cada proceso (algunos autores llaman a estas entradas bloques de control de procesos). Esta entrada contiene información importante acerca del estado del proceso, incluyendo su contador de programa, apuntador de pila, asignación de memoria, estado de sus archivos abiertos, información de contabilidad y planificación, y todo lo de más que debe guardarse acerca del proceso cuando éste cambia del estado en ejecución a listo o bloqueado, de manera que se pueda reiniciar posteriormente como si nunca se hubiera detenido.

Administración de procesos	**Administración de memoria**	**Administración de archivos**
Registros	Apuntador a la información del segmento de texto	Directorio raíz
Contador del programa	Apuntador a la información del segmento de datos	Directorio de trabajo
Palabra de estado del programa	Apuntador a la información del segmento de pila	Descripciones de archivos
Apuntador de la pila		ID de usuario
Estado del proceso		ID de grupo
Prioridad		
Parámetros de planificación		
ID del proceso		
Proceso padre		
Grupo de procesos		
Señales		
Tiempo de inicio del proceso		
Tiempo utilizado de la CPU		
Tiempo de la CPU utilizado por el hijo		
Hora de la siguiente alarma		

1. El hardware mete el contador del programa a la pila, etc.
2. El hardware carga el nuevo contador de programa del vector de interrupciones.
3. Procedimiento en lenguaje ensamblador guarda los registros.
4. Procedimiento en lenguaje ensamblador establece la nueva pila.
5. El servicio de interrupciones de C se ejecuta (por lo general lee y guarda la entrada en el búfer).
6. El planificador decide qué proceso se va a ejecutar a continuación.
7. Procedimiento en C regresa al código de ensamblador.
8. Procedimiento en lenguaje ensamblador inicia el nuevo proceso actual.

Figura 2-5. Esqueleto de lo que hace el nivel más bajo del sistema operativo cuando ocurre una interrupción.

7. Modelación de la multiprogramación

7.1. Cuando se utiliza la multiprogramación, el uso de la CPU se puede mejorar. Dicho en forma cruda: si el proceso promedio realiza cálculos sólo 20 por ciento del tiempo que está en la memoria, con cinco procesos en memoria a la vez la CPU deberá estar ocupada todo el tiempo. Sin embargo, este modelo es demasiado optimista, ya que supone que los cinco procesos nunca estarán esperando la E/S al mismo tiempo.

7.2. Suponga que un proceso gasta una fracción p de su tiempo esperando a que se complete una operación de E/S. Con n procesos en memoria a la vez, la probabilidad de que todos los n procesos estén esperando la E/S (en cuyo

caso, la CPU estará inactiva) es pn. Entonces, el uso de la CPU se obtiene mediante la fórmula

$$\text{Uso de la CPU} = 1 - p^n$$

7.3. Por ejemplo, suponga que una computadora tiene 512 MB de memoria, de la cual el sistema operativo ocupa 128 MB y cada programa de usuario ocupa otros 128 MB. Estos tamaños permiten que haya tres programas de usuario en memoria a la vez. Con un promedio de 80 por ciento de tiempo de espera de E/S, tenemos una utilización de la CPU (ignorando la sobrecarga del sistema operativo) de 1 – 0.83 o de aproximadamente 49 por ciento. Si agregamos 512 MB más de memoria, el sistema puede pasar de la multiprogramación de tres vías a una multiprogramación de siete vías, con lo cual el uso de la CPU se eleva hasta 79 por ciento. En otras palabras, los 512 MB adicionales elevarán el rendimiento en un 30 por ciento. Si agregamos otros 512 MB, el uso de la CPU sólo se incrementa de 79 a 91 por ciento, con lo cual se elevaría el rendimiento sólo en 12% adicional.

- HILOS

Con frecuencia hay situaciones en las que es conveniente tener varios hilos de control en el mismo espacio de direcciones que se ejecuta en cuasi-paralelo, como si fueran procesos (casi) separados (excepto por el espacio de direcciones compartido).

1. Uso de hilos

1.1. La principal razón de tener hilos es que en muchas aplicaciones se desarrollan varias actividades a la vez. Algunas de ellas se pueden bloquear de vez en cuando. Al descomponer una aplicación en varios hilos secuenciales que se ejecutan en cuasi-paralelo, el modelo de programación se simplifica.

1.2. Sólo que ahora con los hilos agregamos un nuevo elemento: la habilidad de las entidades en paralelo de compartir un espacio de direcciones y todos sus datos entre ellas. Esta habilidad es esencial para ciertas aplicaciones, razón por la cual no funcionará el tener varios procesos (con sus espacios de direcciones separados).

1.3. Un segundo argumento para tener hilos es que, como son más ligeros que los procesos, son más fáciles de crear (es decir, rápidos) y destruir. En muchos sistemas, la creación de un hilo es de 10 a 100 veces más rápida que la de un proceso.

1.4. Los hilos son útiles en los sistemas con varias CPUs, en donde es posible el verdadero paralelismo.

```
while (TRUE) {
    obtener_siguiente_peticion(&buf);
    pasar_trabajo(&buf);
}
```

(a)

```
while (TRUE) {
    esperar_trabajo(&buf)
    buscar_pagina_en_cache(&buf,&pagina);
    if (pagina_no_esta_en_cache(&pagina))
        leer_pagina_de_disco(&buf, &pagina);
    devolver_pagina(&pagina);
}
```

(b)

Figura 2-9. Un bosquejo del código para la figura 2-8. (a) Hilo despachador. (b) Hilo trabajador.

Modelo	Características
Hilos	Paralelismo, llamadas al sistema con bloqueo
Proceso con un solo hilo	Sin paralelismo, llamadas al sistema con bloqueo
Máquina de estados finitos	Paralelismo, llamadas al sistema sin bloqueo, interrupciones

2. El modelo clásico de hilo

El modelo de procesos se basa en dos conceptos independientes: agrupamiento de recursos y ejecución. Algunas veces es útil separarlos; aquí es donde entran los hilos.

2.1.

Una manera de ver a un proceso es como si fuera una forma de agrupar recursos relacionados. Un proceso tiene un espacio de direcciones que contiene texto y datos del programa, así como otros recursos. Estos pueden incluir archivos abiertos, procesos hijos, alarmas pendientes, manejadores de señales, información contable y mucho más. Al reunirlos en forma de un proceso, pueden administrarse con más facilidad.

2.2.

El otro concepto que tiene un proceso es un hilo de ejecución, al que por lo general sólo se le llama hilo. El hilo tiene un contador de programa que lleva el registro de cuál instrucción se va a ejecutar a continuación. Tiene registros que contienen sus variables de trabajo actuales. Tiene una pila, que contiene el historial de ejecución, con un conjunto de valores para cada procedimiento al que se haya llamado, pero del cual no se haya devuelto todavía.

Elementos por proceso	Elementos por hilo
Espacio de direcciones	Contador de programa
Variables globales	Registros
Archivos abiertos	Pila
Procesos hijos	Estado
Alarmas pendientes	
Señales y manejadores de señales	
Información contable	

Figura 2-12. La primera columna lista algunos elementos compartidos por todos los hilos en un proceso; la segunda, algunos elementos que son privados para cada hilo.

2.3. Cuando hay multihilamiento, por lo general los procesos empiezan con un solo hilo presente. Este hilo tiene la habilidad de crear hilos mediante la llamada a un procedimiento de biblioteca, como thread_create. Comúnmente, un parámetro para thread_create especifica el nombre de un procedimiento para que se ejecute el nuevo hilo

2.4. En algunos sistemas con hilos, un hilo puede esperar a que un hilo (específico) termine mediante la llamada a un procedimiento, por ejemplo thread_join. Este procedimiento bloquea al hilo llamador hasta que un hilo (específico) haya terminado.

3. Hilos en POSIX

3.1. El IEEE ha definido un estándar para los hilos conocido como 1003.1c. El paquete de hilos que define se conoce como Pthreads. El estándar define más de 60 llamadas a funciones

Llamada de hilo	Descripción
Pthread_create	Crea un nuevo hilo
Pthread_exit	Termina el hilo llamador
Pthread_join	Espera a que un hilo específico termine
Pthread_yield	Libera la CPU para dejar que otro hilo se ejecute
Pthread_attr_init	Crea e inicializa la estructura de atributos de un hilo
Pthread_attr_destroy	Elimina la estructura de atributos de un hilo

Figura 2-14. Algunas de las llamadas a funciones de Pthreads.

4. Implementación de hilos en el espacio de usuario

Hay dos formas principales de implementar un paquete de hilos: en espacio de usuario y en el kernel.

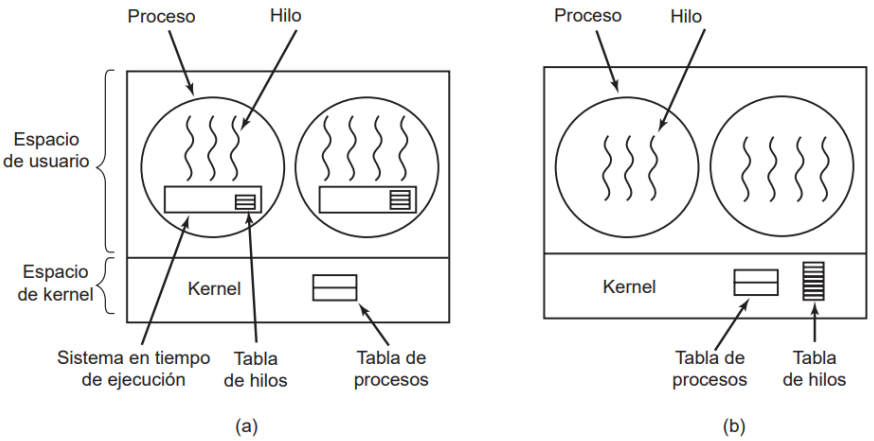


Figura 2-16. (a) Un paquete de hilos de nivel usuario. (b) Un paquete de hilos administrado por el kernel.

4.1. El primer método es colocar el paquete de hilos completamente en espacio de usuario. El kernel no sabe nada acerca de ellos. En lo que al kernel concierne, está administrando procesos ordinarios con un solo hilo. La primera ventaja, la más obvia, es que un paquete de hilos de nivel usuario puede implementarse en un sistema operativo que no acepte hilos. Con este método, los hilos se implementan mediante una biblioteca.

4.2. Ventajas:

4.2.1. Realizar una conmutación de hilos como éste es por lo menos una orden de magnitud (o tal vez más) más veloz que hacer el trap al kernel y es un sólido argumento a favor de los paquetes de hilos de nivel usuario.

4.2.2. Permiten que cada proceso tenga su propio algoritmo de planificación personalizado. Por ejemplo, para algunas aplicaciones, las que tienen un hilo recolector de basura, es una ventaja no tener que preocuparse porque un hilo se detenga en un momento inconveniente. También se escalan mejor, ya que los hilos del kernel requieren sin duda algo de espacio en la tabla y en la pila del kernel, lo cual puede ser un problema si hay una gran cantidad de hilos.

4.2.3. Si el programa llama o salta a una instrucción que no esté en memoria, ocurre un fallo de página y el sistema operativo obtiene la instrucción faltante (y las instrucciones aledañas) del disco. A esto se le conoce como fallo de página.

4.3. Desventajas:

4.3.1. El primero de todos es la manera en que se implementan las llamadas al sistema de bloqueo. Suponga que un hilo lee del teclado antes de que se haya oprimido una sola tecla. Es inaceptable permitir que el hilo realice la llamada al sistema, ya que esto detendrá a todos los hilos. Uno de los principales objetivos de tener hilos en primer lugar era permitir que cada uno utilizara llamadas de bloqueo, pero para evitar que un hilo bloqueado afectará a los demás. Con las llamadas al sistema de bloqueo, es difícil ver cómo se puede lograr este objetivo sin problemas

4.3.2. Es posible otra alternativa si se puede saber de antemano si una llamada se va a bloquear. En algunas versiones de UNIX existe una llamada al sistema (select), la cual permite al procedimiento que hace la llamada saber si una posible llamada a read realizará un bloqueo.El código colocado alrededor de la llamada al sistema que se encarga de la comprobación se conoce como envoltura.

4.3.3. Si un hilo empieza a ejecutarse, ningún otro hilo en ese proceso se ejecutará a menos que el primero renuncie de manera voluntaria a la

CPU. Dentro de un solo proceso no hay interrupciones de reloj, lo cual hace que sea imposible planificar procesos en el formato round robin (tomando turnos).

5. Implementación de hilos en el kernel.

    5.1. No se necesita un sistema en tiempo de ejecución para ninguna de las dos acciones. Además, no hay tabla de hilos en cada proceso. En vez de ello, el kernel tiene una tabla de hilos que lleva la cuenta de todos los hilos en el sistema. Cuando un hilo desea crear un nuevo hilo o destruir uno existente, realiza una llamada al kernel, la cual se encarga de la creación o destrucción mediante una actualización en la tabla de hilos del kernel.

    5.2. Todas las llamadas que podrían bloquear un hilo se implementan como llamadas al sistema, a un costo considerablemente mayor que una llamada a un procedimiento del sistema en tiempo de ejecución. Cuando un hilo se bloquea, el kernel, según lo que decida, puede ejecutar otro hilo del mismo proceso (si hay uno listo) o un hilo de un proceso distinto. Con los hilos de nivel usuario, el sistema en tiempo de ejecución ejecuta hilos de su propio proceso hasta que el kernel le quita la CPU (o cuando ya no hay hilos para ejecutar).

    5.3. Ventajas:

        5.3.1. Los hilos de kernel no requieren de nuevas llamadas al sistema sin bloqueo. Además, si un hilo en un proceso produce un fallo de página, el kernel puede comprobar con facilidad si el proceso tiene otros hilos que puedan ejecutarse y de ser así, ejecuta uno de ellos mientras espera a que se traiga la página requerida desde el disco.

    5.4. Desventajas:

        5.4.1. Su principal desventaja es que el costo de una llamada al sistema es considerable, por lo que si las operaciones de hilos (de creación o terminación, por ejemplo) son comunes, se incurrirá en una mayor sobrecarga.

6. Implementaciones híbridas

    6.1. Una de esas formas es utilizar hilos de nivel kernel y después multiplexar los hilos de nivel usuario con alguno o con todos los hilos de nivel kernel. Cuando se utiliza este método, el programador puede determinar cuántos hilos de kernel va a utilizar y cuántos hilos de nivel usuario va a multiplexar en cada uno. Este modelo proporciona lo último en flexibilidad.

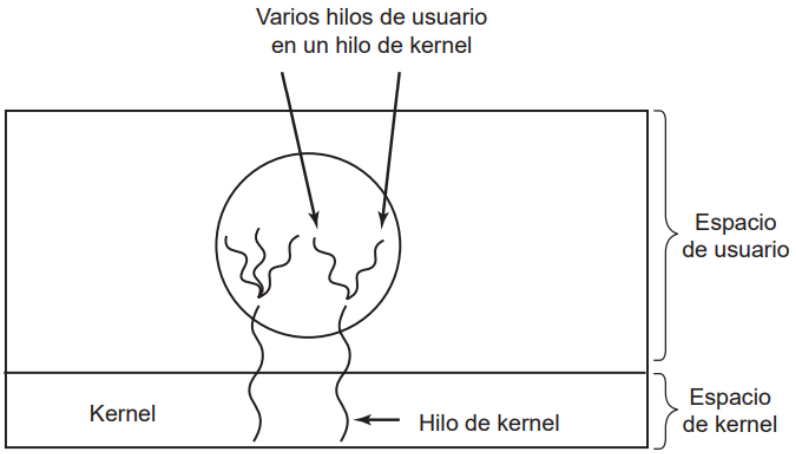


Figura 2-17. Multiplexaje de hilos del nivel usuario sobre hilos del nivel kernel.

7. Activaciones del planificador

7.1. Los objetivos del trabajo de una activación del planificador son imitar la funcionalidad de los hilos de kernel, pero con el mejor rendimiento y la mayor flexibilidad que por lo general se asocian con los paquetes de hilos implementados en espacio de usuario. En especial, los hilos de usuario no deben tener que realizar llamadas especiales al sistema sin bloqueo, ni comprobar de antemano que sea seguro realizar ciertas llamadas al sistema. Sin embargo, cuando un hilo se bloquea en una llamada al sistema o un fallo de página, debe ser posible ejecutar otros hilos dentro del mismo proceso, si hay alguno listo.

7.2. La eficiencia se obtiene evitando transiciones innecesarias entre los espacios de usuario y de kernel. Por ejemplo, si un hilo se bloquea en espera de que otro hilo realice alguna acción, no hay razón para involucrar al kernel, con lo cual se ahorra la sobrecarga de la transición de kernel a usuario. El sistema en tiempo de ejecución en espacio de usuario puede bloquear el hilo sincronizador y programar uno nuevo por sí solo.

7.3. Cuando se utilizan las activaciones del planificador, el kernel asigna cierto número de procesadores virtuales a cada proceso y deja que el sistema en tiempo de ejecución (en espacio de usuario) asigne hilos a los procesadores. Este mecanismo también se puede utilizar en un multiprocesador, donde los procesadores virtuales podrían ser CPUs reales.

7.4. La idea básica que hace que este esquema funcione es que, cuando el kernel sabe que un hilo se ha bloqueado (por ejemplo, al ejecutar una llamada al sistema de bloqueo o al ocasionar un fallo de página), se lo notifica al sistema en tiempo de ejecución del proceso, pasándole como parámetros a la pila el número del hilo en cuestión y una descripción del evento que ocurrió. Para realizar la notificación, el kernel activa el sistema en tiempo de ejecución en una dirección inicial conocida, no muy similar a una

señal en UNIX. A este mecanismo se le conoce como llamada ascendente (upcall).

8. Hilos emergentes

8.1. Los hilos se utilizan con frecuencia en los sistemas distribuidos. Un importante ejemplo es la forma en que se manejan los mensajes entrantes (por ejemplo, las peticiones de servicio). El método tradicional es hacer que un proceso o hilo, que está bloqueado en una llamada al sistema receive, espere un mensaje entrante. Cuando llega un mensaje, lo acepta, lo desempaqueta, examina su contenido y lo procesa.

8.2. También es posible utilizar un método completamente distinto, en el cual la llegada de un mensaje hace que el sistema cree un nuevo hilo para manejar el mensaje. A dicho hilo se le conoce como hilo emergente (pop-up thread)

8.3. Una ventaja clave de los hilos emergentes es que, como son nuevos, no tienen historial (registros, pila, etcétera) que sea necesario restaurar. Cada uno empieza desde cero y es idéntico a los demás. Esto hace que sea posible crear dicho hilo con rapidez. El nuevo hilo recibe el mensaje entrante que va a procesar. El resultado de utilizar hilos emergentes es que la latencia entre la llegada del mensaje y el inicio del procesamiento puede ser muy baja.

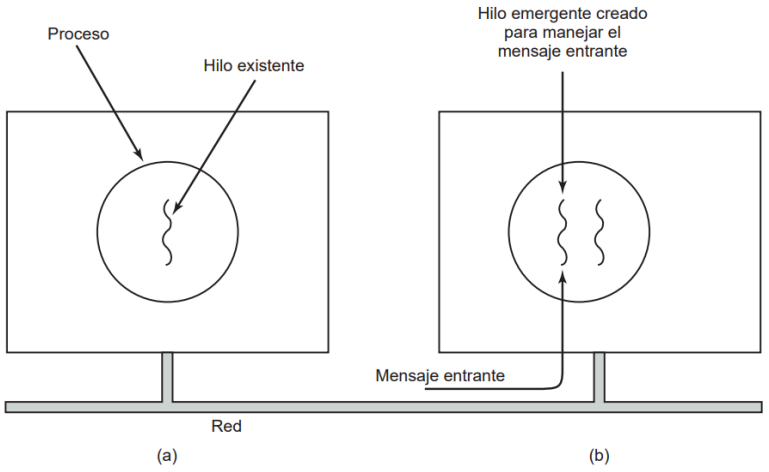


Figura 2.18. Creación de un nuevo hilo cuando llega un mensaje. (a) Antes de que llegue el mensaje. (b) Después de que llega el mensaje.

Por otro lado, un hilo de kernel con errores puede hacer más daño que un hilo de usuario con errores. Por ejemplo, si se ejecuta durante demasiado tiempo y no hay manera de quitarlo, los datos entrantes se pueden perder.

9. Conversión de código de hilado simple a multi hilado

9.1. Para empezar, el código de un hilo normalmente consiste de varios procedimientos, al igual que un proceso. Éstos pueden tener variables locales, variables globales y parámetros. Las variables y parámetros locales

no ocasionan problemas, pero las variables que son globales a un hilo, pero no globales para todo el programa, son un problema.

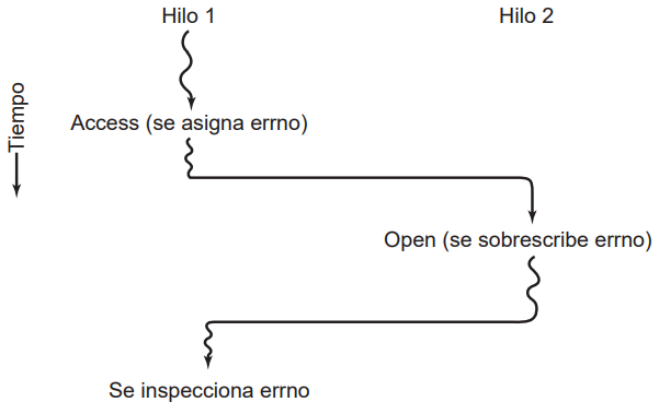


Figura 2-19. Conflictos entre los hilos por el uso de una variable global.

Otra solución es asignar a cada hilo sus propias variables globales privadas. De esta forma, cada hilo tiene su propia copia privada de errno y de otras variables globales, por lo que se evitan los conflictos.

9.2. Al convertir un programa con un solo hilo en un programa con múltiples hilos es que muchos procedimientos de biblioteca no son re-entrantes; es decir, no se diseñaron para hacer una segunda llamada a cualquier procedimiento dado mientras que una llamada anterior no haya terminado. Por ejemplo, podemos programar el envío de un mensaje a través de la red ensamblando el mensaje en un búfer fijo dentro de la biblioteca, para después hacer un trap al kernel para enviarlo. ¿Qué ocurre si un hilo ha ensamblado su mensaje en el búfer y después una interrupción de reloj obliga a que se haga la conmutación a un segundo hilo que de inmediato sobrescribe el búfer con su propio mensaje?

9.2.1. Para corregir estos problemas de manera efectiva, tal vez sea necesario reescribir la biblioteca completa, lo que no es insignificante.

9.2.2. Una solución distinta es proporcionar a cada procedimiento una envoltura que fije un bit para marcar la librería como si estuviera en uso. Si otro hilo intenta usar un procedimiento de biblioteca mientras no se haya completado una llamada anterior, se bloquea. Aunque se puede hacer que este método funcione, elimina en gran parte el paralelismo potencial.

9.3. Un último problema que introducen los hilos es la administración de la pila. En muchos sistemas, cuando la pila de un proceso se desborda, el kernel sólo proporciona más pila a ese proceso de manera automática. Cuando un proceso tiene múltiples hilos, también debe tener varias pilas. Si el kernel no está al tanto de todas ellas, no puede hacer que su tamaño aumente de manera automática cuando ocurra un fallo de la pila.De hecho, ni siquiera puede detectar que un fallo de memoria está relacionado con el aumento de tamaño de la pila de algún otro hilo.

- COMUNICACIÓN ENTRE PROCESOS

  Hay tres cuestiones aquí. La primera se alude a lo anterior: cómo un proceso puede pasar información a otro. La segunda está relacionada con hacer que dos o más procesos no se interpongan entre sí; por ejemplo, dos procesos en un sistema de reservaciones de una aerolínea, cada uno de los cuales trata de obtener el último asiento en un avión para un cliente distinto. La tercera trata acerca de obtener la secuencia apropiada cuando hay dependencias presentes: si el proceso A produce datos y el proceso B los imprime, B tiene que esperar hasta que A haya producido algunos datos antes de empezar a imprimir.

1. Condiciones de carrera

   1.1. Un spooler de impresión. Cuando un proceso desea imprimir un archivo, introduce el nombre del archivo en un directorio de spooler especial. Otro proceso, el demonio de impresión, comprueba en forma periódica si hay archivos que deban imprimirse y si los hay, los imprime y luego elimina sus nombres del directorio.

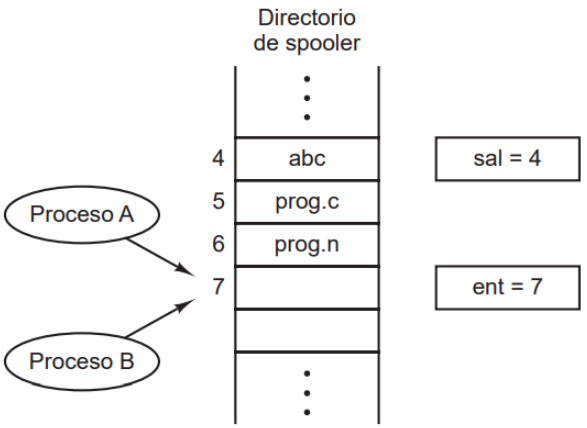


Figura 2-21. Dos procesos desean acceder a la memoria compartida al mismo tiempo.

   1.2. Situaciones como ésta, en donde dos o más procesos están leyendo o escribiendo algunos datos compartidos y el resultado final depende de quién se ejecuta y exactamente cuándo lo hace, se conocen como condiciones de carrera.

2. Regiones críticas

   2.1. ¿Cómo evitamos las condiciones de carrera? Lo que necesitamos es exclusión mutua, cierta forma de asegurar que si un proceso está utilizando una variable o archivo compartido, los demás procesos se excluirán de hacer lo mismo.

   2.2. Esa parte del programa en la que se accede a la memoria compartida se conoce como región crítica o sección crítica.

2.3. Necesitamos cumplir con cuatro condiciones para tener una buena solución:

2.3.1. 1. No puede haber dos procesos de manera simultánea dentro de sus regiones críticas.

2.3.2. 2. No pueden hacerse suposiciones acerca de las velocidades o el número de CPUs.

2.3.3. 3. Ningún proceso que se ejecute fuera de su región crítica puede bloquear otros procesos.

2.3.4. 4. Ningún proceso tiene que esperar para siempre para entrar a su región crítica

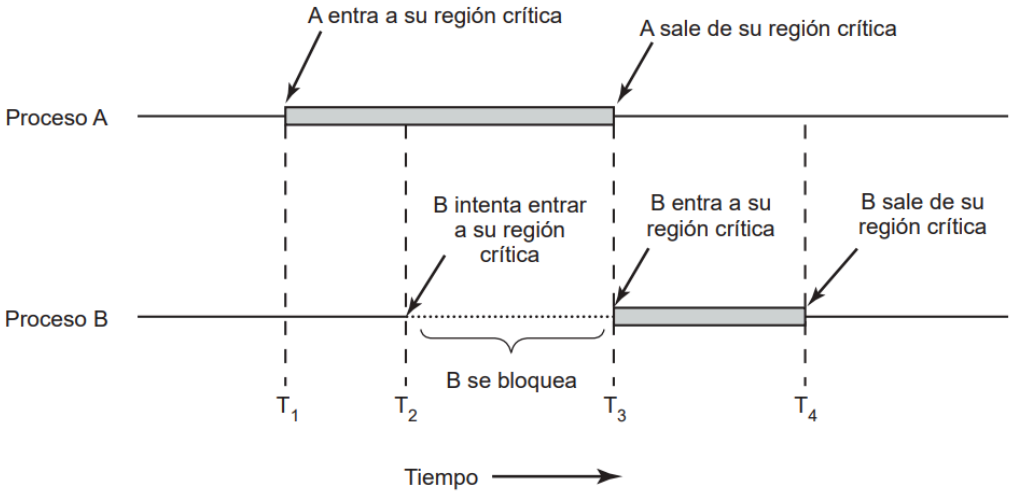


Figura 2-22. Exclusión mutua mediante el uso de regiones críticas.

3. Exclusión mutua con espera ocupada

3.1. Deshabilitando interrupciones

3.1.1. La solución más simple es hacer que cada proceso deshabilite todas las interrupciones justo después de entrar a su región crítica y las rehabilite justo después de salir. Con las interrupciones deshabilitadas, no pueden ocurrir interrupciones de reloj. Después de todo, la CPU sólo se conmuta de un proceso a otro como resultado de una interrupción del reloj o de otro tipo, y con las interrupciones desactivadas la CPU no se conmutará a otro proceso. Por ende, una vez que un proceso ha deshabilitado las interrupciones, puede examinar y actualizar la memoria compartida sin temor de que algún otro proceso intervenga.

3.1.2. La posibilidad de lograr la exclusión mutua al deshabilitar las interrupciones (incluso dentro del kernel) está disminuyendo día con día debido al creciente número de chips multinúcleo que se encuentran hasta en las PCs de bajo rendimiento. Ya es común que haya dos núcleos, las máquinas actuales de alto rendimiento tienen cuatro y dentro de poco habrá ocho o 16.

3.2. Variables de candado

3.2.1. Considere tener una sola variable compartida (de candado), que al principio es 0. Cuando un proceso desea entrar a su región crítica primero evalúa el candado. Si este candado es 0, el proceso lo fija en 1 y entra a la región crítica. Si el candado ya es 1 sólo espera hasta que el candado se haga 0. Por ende, un 0 significa que ningún proceso está en su región crítica y un 1 significa que algún proceso está en su región crítica.

3.2.2. Esta idea contiene exactamente el mismo error fatal que vimos en el directorio de spooler. Suponga que un proceso lee el candado y ve que es 0. Antes de que pueda fijar el candado a 1, otro proceso se planifica para ejecutarse y fija el candado a 1. Cuando el primer proceso se ejecuta de nuevo, también fija el candado a 1 y por lo tanto dos procesos se encontrarán en sus regiones críticas al mismo tiempo

3.3. Alternancia estricta

3.3.1. A la acción de evaluar en forma continua una variable hasta que aparezca cierto valor se le conoce como espera ocupada. Por lo general se debe evitar, ya que desperdicia tiempo de la CPU. La espera ocupada sólo se utiliza cuando hay una expectativa razonable de que la espera será corta. A un candado que utiliza la espera ocupada se le conoce como candado de giro.

3.3.2. Aunque este algoritmo evita todas las condiciones de carrera, en realidad no es un candidato serio como solución, ya que viola la condición 3.

3.4. Solución de Peterson

```
#define FALSE 0
#define TRUE 1
#define N   2                                   /* número de procesos */

int turno;                                      /* ¿de quién es el turno? */
int interesado[N];                              /* al principio todos los valores son 0 (FALSE) */

void entrar_region(int proceso);                /* el proceso es 0 o 1 */
{
    int otro;                                   /* número del otro proceso */

    otro = 1 – proceso;                         /* el opuesto del proceso */
    interesado[proceso] = TRUE;                 /* muestra que está interesado */
    turno = proceso;                            /* establece la bandera */
    while (turno == proceso && interesado[otro] == TRUE)  /* instrucción nula */;
}

void salir_region(int proceso)                  /* proceso: quién está saliendo */
{
    interesado[proceso] = FALSE;                /* indica que salió de la región crítica */
}
```

Figura 2-24. Solución de Peterson para lograr la exclusión mutua.

3.5. La instrucción TSL

3.5.1. Lee el contenido de la palabra de memoria candado y lo guarda en el registro RX, y después almacena un valor distinto de cero en la dirección de memoria candado. Se garantiza que las operaciones de leer la palabra y almacenar un valor en ella serán indivisibles; ningún otro procesador puede acceder a la palabra de memoria sino hasta que termine la instrucción. La CPU que ejecuta la instrucción TSL bloquea el bus de memoria para impedir que otras CPUs accedan a la memoria hasta que termine

3.5.2. Para usar la instrucción TSL necesitamos una variable compartida (candado) que coordine el acceso a la memoria compartida. Cuando el candado es 0, cualquier proceso lo puede fijar en 1 mediante el uso de la instrucción TSL y después una lectura o escritura en la memoria compartida. Cuando termina, el proceso establece candado de vuelta a 0 mediante una instrucción move ordinaria.

```
entrar_region:
    TSL REGISTRO,CANDADO        |copia candado al registro y fija candado a 1
    CMP REGISTRO,#0             |¿era candado cero?
    JNE entrar_region           |si era distinto de cero, el candado está cerrado, y se repite
    RET                         |regresa al llamador; entra a región crítica

salir_region:
    MOVE CANDADO,#0             |almacena 0 en candado
    RET                         |regresa al llamador
```

Figura 2-25. Cómo entrar y salir de una región crítica mediante la instrucción TSL.

4. Dormir y despertar

4.1. Tanto la solución de Peterson como las soluciones mediante TSL o XCHG son correctas, pero todas tienen el defecto de requerir la espera ocupada

4.2. Una de las más simples es el par sleep (dormir) y wakeup (despertar). Sleep es una llamada al sistema que hace que el proceso que llama se bloquee o desactive, es decir, que se suspenda hasta que otro proceso lo despierte. La llamada wakeup tiene un parámetro, el proceso que se va a despertar o activar. De manera alternativa, tanto sleep como wakeup tienen un parámetro, una dirección de memoria que se utiliza para asociar las llamadas a sleep con las llamadas a wakeup.

4.3. El problema del productor-consumidor

4.3.1. El problema surge cuando el productor desea colocar un nuevo elemento en el búfer, pero éste ya se encuentra lleno. La solución es que el productor se vaya a dormir (se desactiva) y que se despierte (se active) cuando el consumidor haya quitado uno o más elementos.

De manera similar, si el consumidor desea quitar un elemento del búfer y ve que éste se encuentra vacío, se duerme hasta que el productor coloca algo en el búfer y lo despierta.

```
#define N 100                                   /* número de ranuras en el búfer */
int cuenta = 0;                                 /* número de elementos en el búfer */

void productor(void)
{
        int elemento;

        while (TRUE) {                          /* se repite en forma indefinida */
                elemento = producir_elemento();         /* genera el siguiente elemento */
                if (cuenta == N) sleep();               /* si el búfer está lleno, pasa a inactivo */
                insertar_elemento(elemento);            /* coloca elemento en búfer */
                cuenta = cuenta + 1;                    /* incrementa cuenta de elementos en búfer */
                if (cuenta == 1) wakeup(consumidor);    /* ¿estaba vacío el búfer? */
        }
}


void consumidor(void)
{
        int elemento;

        while (TRUE) {                          /* se repite en forma indefinida */
                if (cuenta == 0) sleep();               /* si búfer está vacío, pasa a inactivo */
                elemento = quitar_elemento();           /* saca el elemento del búfer */
                cuenta = cuenta – 1;                    /* disminuye cuenta de elementos en búfer */
                if (cuenta==N–1) wakeup(productor);     /* ¿estaba lleno el búfer? */
                consumir_elemento(elemento);            /* imprime el elemento */
        }
}
```

Figura 2-27. El problema del productor-consumidor con una condición de carrera fatal.

4.4. Ahora regresemos a la condición de carrera. Puede ocurrir debido a que el acceso a cuenta no está restringido. Una solución rápida es modificar las reglas para agregar al panorama un bit de espera de despertar. Cuando se envía una señal de despertar a un proceso que sigue todavía despierto, se fija este bit.

5. Semáforos

5.1. En su propuesta introdujo un nuevo tipo de variable, al cual él llamó semáforo. Un semáforo podría tener el valor 0, indicando que no se guardaron señales de despertar o algún valor positivo si estuvieran pendientes una o más señales de despertar

5.2. La operación down en un semáforo comprueba si el valor es mayor que 0. De ser así, disminuye el valor y sólo continúa. Si el valor es 0, el proceso se pone a dormir sin completar la operación down por el momento. Las acciones de comprobar el valor, modificarlo y posiblemente pasar a dormir, se realizan en conjunto como una sola acción atómica indivisible.

5.3. Cómo resolver el problema del productor-consumidor mediante el uso de semáforos

5.3.1. Esta solución utiliza tres semáforos: uno llamado llenas para contabilizar el número de ranuras llenas, otro llamado vacías para contabilizar el número de ranuras vacías y el último llamado mutex, para asegurar que el productor y el consumidor no tengan acceso al búfer al mismo tiempo.

```
#define N 100                                   /* número de ranuras en el búfer */
typedef int semaforo;                           /* los semáforos son un tipo especial de int */
semaforo mutex = 1;                             /* controla el acceso a la región crítica */
semaforo vacias = N;                            /* cuenta las ranuras vacías del búfer */
semaforo llenas = 0;                            /* cuenta las ranuras llenas del búfer */

void productor(void)
{
     int elemento;

     while(TRUE){                               /* TRUE es la constante 1 */
          elemento = producir_elemento();       /* genera algo para colocar en el búfer */
          down(&vacias);                        /* disminuye la cuenta de ranuras vacías */
          down(&mutex);                         /* entra a la región crítica */
          insertar_elemento(elemento);          /* coloca el nuevo elemento en el búfer */
          up(&mutex);                           /* sale de la región crítica */
          up(&llenas);                          /* incrementa la cuenta de ranuras llenas */
     }
}


void consumidor(void)
{
     int elemento;

     while(TRUE){                               /* ciclo infinito */
          down(&llenas);                        /* disminuye la cuenta de ranuras llenas */
          down(&mutex);                         /* entra a la región crítica */
          elemento = quitar_elemento();         /* saca el elemento del búfer */
          up(&mutex);                           /* sale de la región crítica */
          up(&vacias);                          /* incrementa la cuenta de ranuras vacías */
          consumir_elemento(elemento);          /* hace algo con el elemento */
     }
}
```

Figura 2-28. El problema del productor-consumidor mediante el uso de semáforos.

6. Mutexes

6.1. Cuando no se necesita la habilidad del semáforo de contar, algunas veces se utiliza una versión simplificada, llamada mutex. Los mutexes son buenos sólo para administrar la exclusión mutua para cierto recurso compartido o pieza de código.

6.2. Un mutex es una variable que puede estar en uno de dos estados: abierto (desbloqueado) o cerrado (bloqueado). En consecuencia, se requiere sólo 1 bit para representarla, pero en la práctica se utiliza con frecuencia un entero, en donde 0 indica que está abierto y todos los demás valores indican que está cerrado.

```
mutex_lock:
        TSL REGISTRO,MUTEX      |copia el mutex al registro y establece mutex a 1
        CMP REGISTRO,#0         |¿el mutex era 0?
        JZE ok                  |si era cero, el mutex estaba abierto, entonces regresa
        CALL thread_yield       |el mutex está ocupado; planifica otro hilo
        JMP mutex_lock          |intenta de nuevo
ok:     RET                     |regresa al procedimiento llamador; entra a la región crítica

mutex_unlock:
        MOVE MUTEX,#0           |almacena un 0 en el mutex
        RET                     |regresa al procedimiento llamador
```

Figura 2-29. Implementaciones de *mutex_lock* y *mutex_unlock*.

6.3. Mutexes en Pthread: Pthreads proporciona varias funciones que se pueden utilizar para sincronizar los hilos. El mecanismo básico utiliza una variable mutex, cerrada o abierta, para resguardar cada región crítica.

Llamada de hilo	Descripción
Pthread_mutex_init	Crea un mutex
Pthread_mutex_destroy	Destruye un mutex existente
Pthread_mutex_lock	Adquiere un mutex o se bloquea
Pthread_mutex_trylock	Adquiere un mutex o falla
Pthread_mutex_unlock	Libera un mutex

Figura 2-30. Algunas de las llamadas de Pthreads relacionadas con mutexes.

6.4. Además de los mutexes, pthreads ofrece un segundo mecanismo de sincronización: las variables de condición. Las variables de condición permiten que los hilos se bloqueen debido a que cierta condición no se está cumpliendo. Casi siempre se utilizan los dos métodos juntos.

Llamada de hilo	Descripción
Pthread_cond_init	Crea una variable de condición
Pthread_cond_destroy	Destruye una variable de condición
Pthread_cond_wait	Bloquea en espera de una señal
Pthread_cond_signal	Envia señal a otro hilo y lo despierta
Pthread_cond_broadcast	Envia señal a varios hilos y los despierta

Figura 2-31. Algunas de las llamadas a Pthreads que se relacionan con las variables de condición.

7. Monitores

7.1. propusieron una primitiva de sincronización de mayor nivel, conocida como monitor. Sus proposiciones tienen ligeras variaciones, como se describe a continuación. Un monitor es una colección de procedimientos, variables y estructuras de datos que se agrupan en un tipo especial de módulo o paquete. Los procesos pueden llamar a los procedimientos en un monitor cada vez que lo desean, pero no pueden acceder de manera directa a las estructuras de datos internas del monitor desde procedimientos declarados fuera de éste.

7.2. Los monitores tienen una importante propiedad que los hace útiles para lograr la exclusión mutua: sólo puede haber un proceso activo en un monitor en cualquier instante. Los monitores son una construcción del lenguaje de programación, por lo que el compilador sabe que son especiales y puede manejar las llamadas a los procedimientos del monitor en forma distinta a las llamadas a otros procedimientos.

7.3. También necesitamos una forma en la que los procesos se bloqueen cuando no puedan continuar: La solución está en la introducción de las variables de condición, junto con dos operaciones de éstas: wait y signal. Cuando un procedimiento de monitor descubre que no puede continuar realiza una operación wait en alguna variable de condición. La operación wait debe ir antes de la operación signal. Esta regla facilita la implementación en forma considerable. En la práctica no es un problema debido a que es fácil llevar el registro del estado de cada proceso con variables, si es necesario.

7.4. Al automatizar la exclusión mutua de las regiones críticas, los monitores hacen que la programación en paralelo sea mucho menos propensa a errores que con los semáforos.

8. Pasaje (transmisión) de mensajes

8.1. Ese “algo más” es el pasaje de mensajes (message passing). Este método de comunicación entre procesos utiliza dos primitivas (send y receive) que, al igual que los semáforos y a diferencia de los monitores, son llamadas al sistema en vez de construcciones del lenguaje.

9. Barreras

9.1. Algunas aplicaciones se dividen en fases y tienen la regla de que ningún proceso puede continuar a la siguiente fase sino hasta que todos los procesos estén listos para hacerlo. Para lograr este comportamiento, se coloca una barrera al final de cada fase. Cuando un proceso llega a la barrera, se bloquea hasta que todos los procesos han llegado a ella.

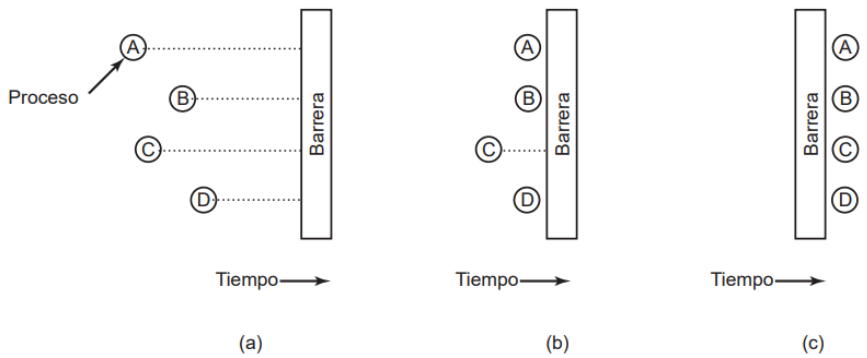


Figura 2-37. Uso de una barrera. (a) Procesos acercándose a una barrera. (b) Todos los procesos, excepto uno, bloqueados en la barrera. (c) Cuando el último proceso llega a la barrera, todos se dejan pasar.

- PLANIFICACIÓN

1. Introducción a la planificación

1.1. Comportamiento de un proceso

1.1.1. Casi todos los procesos alternan ráfagas de cálculos con peticiones de E/S (de disco). Por lo general la CPU opera durante cierto tiempo sin detenerse, después se realiza una llamada al sistema para leer datos de un archivo o escribirlos en el mismo. Cuando se completa la llamada al sistema, la CPU realiza cálculos de nuevo hasta que necesita más datos o tiene que escribir más datos y así sucesivamente.

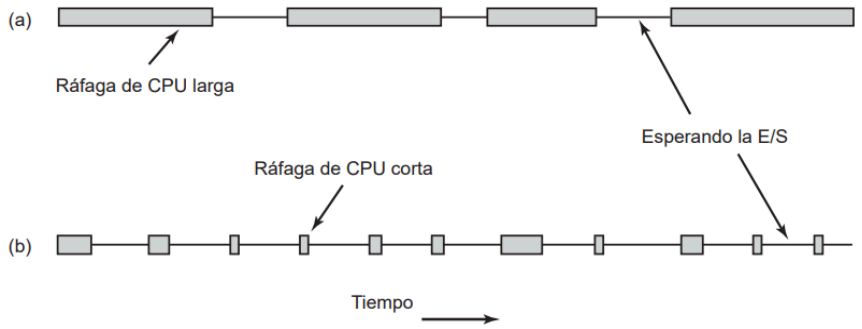


Figura 2-38. Las ráfagas de uso de la CPU se alternan con los periodos de espera por la E/S. (a) Un proceso ligado a la CPU. (b) Un proceso ligado a la E/S.

1.2. Cuándo planificar procesos

1.2.1. 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.

1.2.2. En segundo lugar, se debe tomar una decisión de planificación cuando un proceso termina.

1.2.3. 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.

1.2.4. En cuarto lugar, cuando ocurre una interrupción de E/S tal vez haya que tomar una decisión de planificación.

1.3. No Apropiativos

1.3.1. Un algoritmo de programación no apropiativo (nonpreemptive) selecciona un proceso para ejecutarlo y después sólo deja que se ejecute hasta que el mismo se bloqueao hasta que libera la CPU en forma voluntaria.

1.4. Apropiativos

1.4.1. 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

1.5. Categorías de los algoritmos de planificación

1.5.1. 1. Procesamiento por lotes: En consecuencia, son aceptables los algoritmos no apropiativos. Este método reduce la conmutación de procesos y por ende, mejora el rendimiento.

1.5.2. 2. Interactivo: La apropiación es esencial para evitar que un proceso acapara la CPU y niegue el servicio a los demás. Los servidores también entran en esta categoría, ya que por lo general dan servicio a varios usuarios (remotos), todos los cuales siempre tienen mucha prisa.

1.5.3. 3. De tiempo real: La apropiación a veces es no necesaria debido a que los procesos saben que no se pueden ejecutar durante periodos extensos, que por lo general realizan su trabajo y se bloquean con rapidez.

1.6. 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. Planificación en sistemas de procesamiento por lotes

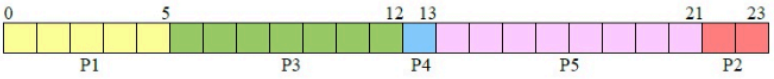
2.1. FCFS, (First-Come First-Served) (primero en entrar primero en ser atendido) (no apropiativo):

2.1.1. La CPU se asigna a los procesos en el orden en el que la solicitan. 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.

Ejemplo práctico:

PROCESO	LLEGADA	DURACIÓN
P1	0	5
P2	4	2
P3	1	7
P4	2	1
P5	3	8



Tiempos de espera:		Tiempos de retorno:
P1	(0-0)=0	P1 = 5
P2	(21-4)=17	P2 = 23
P3	(5-1)=4	P3 = 12
P4	(12-2)=10	P4 = 13
P5	(13-3)=10	P5 = 21

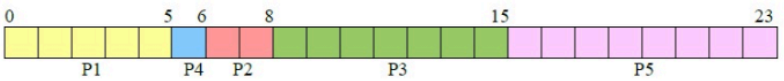
Tiempo medio de espera: (0 + 17 + 4 + 10 + 10) / 5 = 8,2
Tiempo medio de retorno: (5 + 23 + 12 + 13 + 21)/5 = 14,8

2.2. SJF (Shortest Job First) (el trabajo más corto primero) (no apropiativo):

2.2.1. En este algoritmo , da bastante prioridad a los procesos más cortos a la hora de ejecución y los coloca en la cola. Selecciona el proceso con el próximo tiempo de ejecución más corto y lo ejecuta hasta que finaliza el proceso. Si hay dos procesos cuyas ráfagas de la CPU tiene la misma duración, se emplea el algoritmo FCFS o FIFO para romper el empate.

Ejemplo práctico:

PROCESO	LLEGADA	DURACIÓN
P1	0	5
P2	4	2
P3	1	7
P4	2	1
P5	3	8



Tiempos de espera:		Tiempos de retorno:
P1	(0-0)=0	P1 = 5
P2	(6-4)=2	P2 = 8
P3	(8-1)=7	P3 = 15
P4	(5-2)=3	P4 = 6
P5	(15-3)=12	P5 = 23

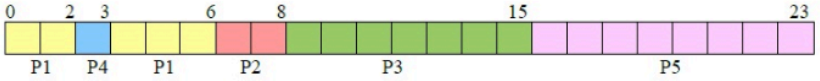
Tiempo medio de espera: (0 + 2 + 7 + 3 + 12) / 5 = 4,8
Tiempo medio de retorno: (5 + 8 + 15 + 6 + 23) / 5 = 11,4

2.3. SRTN (Shortest Remaining Time Next) (el menor tiempo restante a continuación) (apropiativa):

2.3.1. Con este algoritmo, el planificador siempre selecciona el proceso cuyo tiempo restante de ejecución sea el más corto. 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.

Ejemplo práctico:

PROCESO	LLEGADA	DURACIÓN
P1	0	5
P2	4	2
P3	1	7
P4	2	1
P5	3	8



Tiempos de espera:		Tiempos de retorno:
P1	(0-0)=0	P1 = 6
P2	(6-4)=2	P2 = 8
P3	(8-1)=7	P3 = 15
P4	(2-2)=0	P4 = 3
P5	(15-3)=12	P5 = 23

Tiempo medio de espera: (0 + 2 + 7 + 0 + 12) / 5 = 4,2
Tiempo medio de retorno: (6 + 8 + 15 + 3 + 23) / 5 = 11

3. Planificación en sistemas interactivos

3.1. round-robin (turno circular) (apropiativo): Uno de los algoritmos más antiguos, simples, equitativos y de mayor uso

3.1.1. 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 cuento, 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. La cola de procesos se estructura como una cola circular. La organización de la cola es FIFO.

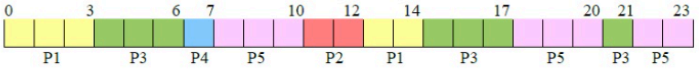
3.1.2. Suponga que está conmutación de proceso o conmutación de contexto requiere 1 mseg, incluyendo el cambio de los mapas de memoria, el vaciado y recarga de la caché, etc. Suponga además que el quántum se establece a 4 mseg. Con estos parámetros, después de realizar 4 mseg de trabajo útil, la CPU tendrá que gastar (es decir, desperdiciar) 1 mseg en la conmutación de procesos.

3.1.3. 20 por ciento de la CPU se desperdiciará por sobrecarga administrativa.

Ejemplo práctico:

PROCESO	LLEGADA	DURACIÓN
P1	0	5
P2	4	2
P3	1	7
P4	2	1
P5	3	8

Q=3



Tiempos de espera:		Tiempos de retorno:
P1	(0-0)=0	P1 = 14
P2	(10-4)=6	P2 = 12
P3	(3-1)=2	P3 = 21
P4	(6-2)=4	P4 = 7
P5	(7-3)=4	P5 = 23

Tiempo medio de espera: (0 + 6 + 2 + 4 + 4) / 5 = 3,2
Tiempo medio de retorno: (14 + 12 + 21 + 7 + 23) / 5 = 15,4

3.2. Planificación por prioridad:

3.2.1. 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.

3.2.2. 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. 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.

3.3. Múltiples colas

3.3.1. Su solución fue la de establecer clases de prioridades. Los procesos en la clase más alta se ejecutaban durante un quántum. Los procesos en la siguiente clase más alta se ejecutaban por dos quántums. Los procesos en la siguiente clase se ejecutaban por cuatro quántums, y así sucesivamente. Cada vez que un proceso utilizaba todos los quántums que tenía asignados, se movía una clase hacia abajo en la jerarquía

3.3.2. Cuando un proceso en espera de un bloque de disco pasaba al estado listo, se enviaba a la segunda clase. Cuando a un proceso que estaba todavía en ejecución se le agotaba su quántum, al principio se colocaba en la tercera clase. No obstante, si un proceso utilizaba todo su quántum demasiadas veces seguidas sin bloquearse en espera de la terminal o de otro tipo de E/S, se movía hacia abajo hasta la última

cola. Muchos otros sistemas utilizan algo similar para favorecer a los usuarios y procesos interactivos en vez de los que se ejecutan en segundo plano.

3.4. El proceso más corto a continuación:

3.4.1. 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.

3.4.2. La técnica de estimar el siguiente valor en una serie mediante la obtención del promedio ponderado del valor actual medido y la estimación anterior se conoce algunas veces como envejecimiento.

3.5. Planificación garantizada:

3.5.1. Un método completamente distinto para la planificación es hacer promesas reales a los usuarios acerca del rendimiento y después cumplirlas.

3.5.2. 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.

3.6. Planificación por sorteo:

3.6.1. Aunque hacer promesas a los usuarios y cumplirlas es una buena idea, es algo difícil de implementar. Este algoritmo se conoce como planificación por sorteo

3.6.2. 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.

3.6.3. La planificación por lotería tiene un alto grado de respuesta.

3.6.4. Los procesos cooperativos pueden intercambiar boletos si lo desean. Por ejemplo, cuando un proceso cliente envía un mensaje a un proceso servidor y después se bloquea, puede dar todos sus boletos al servidor para incrementar la probabilidad de que éste se ejecute a continuación.

3.7. Planificación por partes equitativas:

3.7.1. 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.

3.7.2. 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.

4. Planificación en sistemas de tiempo real

4.1. 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.

4.2. 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.

4.3. En ambos casos, el comportamiento en tiempo real se logra dividiendo el programa en varios procesos, donde el comportamiento de cada uno de éstos es predecible y se conoce de antemano. Por lo general, estos procesos tienen tiempos de vida cortos y pueden ejecutarse hasta completarse en mucho menos de 1 segundo.

4.4. Los algoritmos de planificación en tiempo real pueden ser estáticos o dinámicos.

4.4.1. Los primeros toman sus decisiones de planificación antes de que el sistema empiece a ejecutarse. La planificación estática sólo funciona cuando hay información perfecta disponible de antemano acerca del trabajo que se va a realizar y los tiempos límite que se tienen que cumplir.

4.4.2. Los segundos lo hacen durante el tiempo de ejecución. Los algoritmos de planificación dinámicos no tienen estas restricciones.

5. Política contra mecanismo

5.1. 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.

5.2. 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.

6. Planificación de hilos

6.1. 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).

6.2. Consideremos primero los hilos a nivel usuario. Como el kernel no está consciente de la existencia de los hilos, opera de la misma forma de siempre. El algoritmo de planificación utilizado por el sistema en tiempo de ejecución puede ser cualquiera de los antes descritos. La única restricción es la ausencia de un reloj para interrumpir a un proceso que se ha ejecutado por mucho tiempo

6.3. 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.

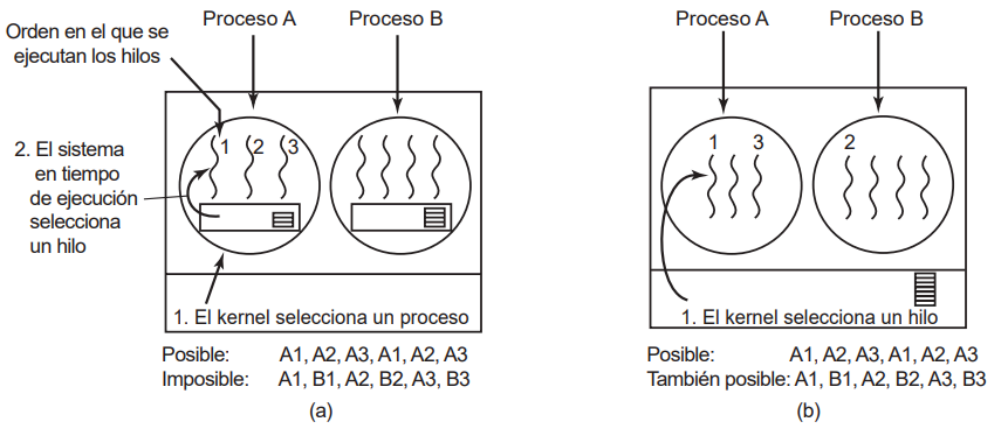


Figura 2-43. (a) Posible planificación de hilos a nivel usuario con un quántum de 50 mseg para cada proceso e hilos que se ejecutan durante 5 mseg por cada ráfaga de la CPU. (b) Posible planificación de hilos a nivel kernel con las mismas características que (a).

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.

- PROBLEMAS CLÁSICOS DE COMUNICACIÓN ENTRE PROCESOS (IPC)

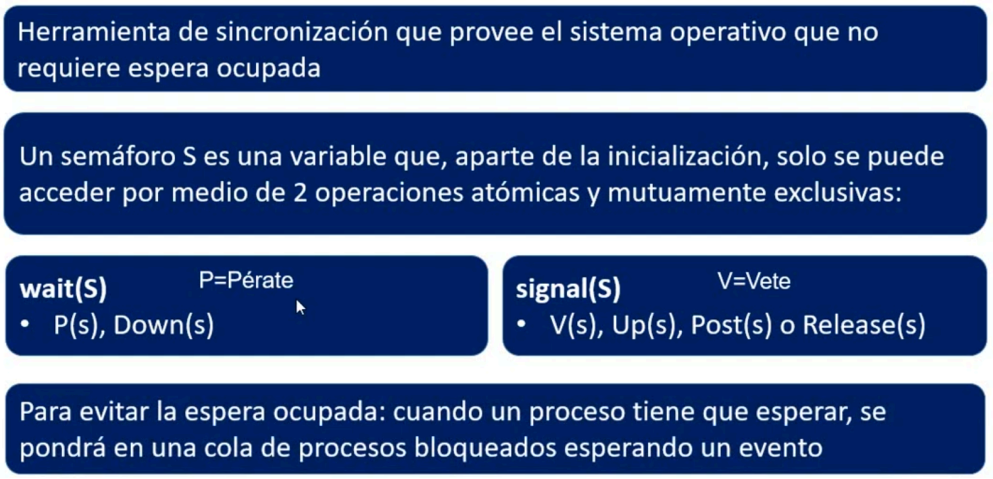
1. El problema de los filósofos comelones

   Sistemas Operativos, Interbloqueo 5 El problema de los filósofos

2. El problema de los lectores y escritores

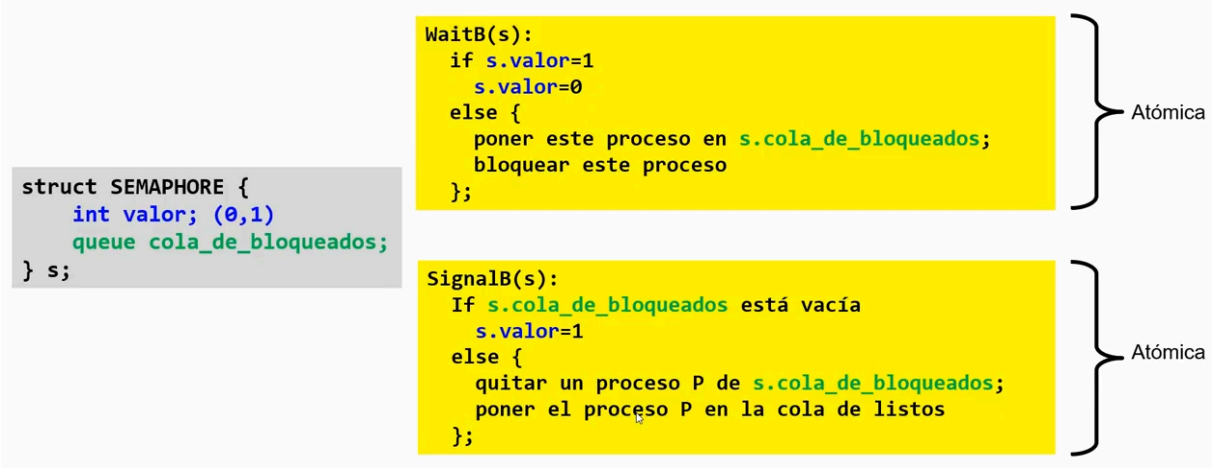
Sistemas Operativos, Problema de la concurrencia 29 Lectores escritores, prioridad a los escritores

- EXTRAS SEMÁFOROS



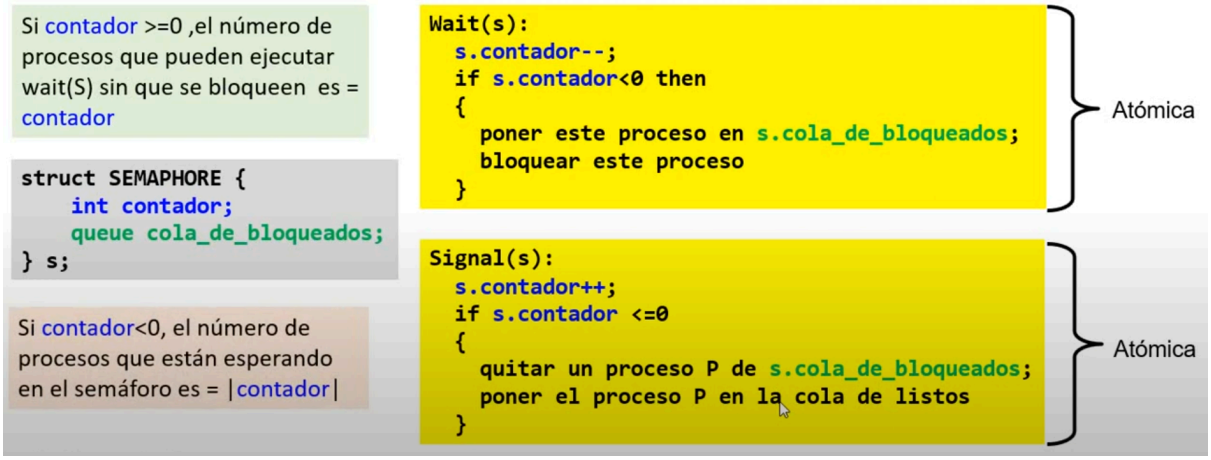
- MUTEX

SEMÁFORO BINARIO

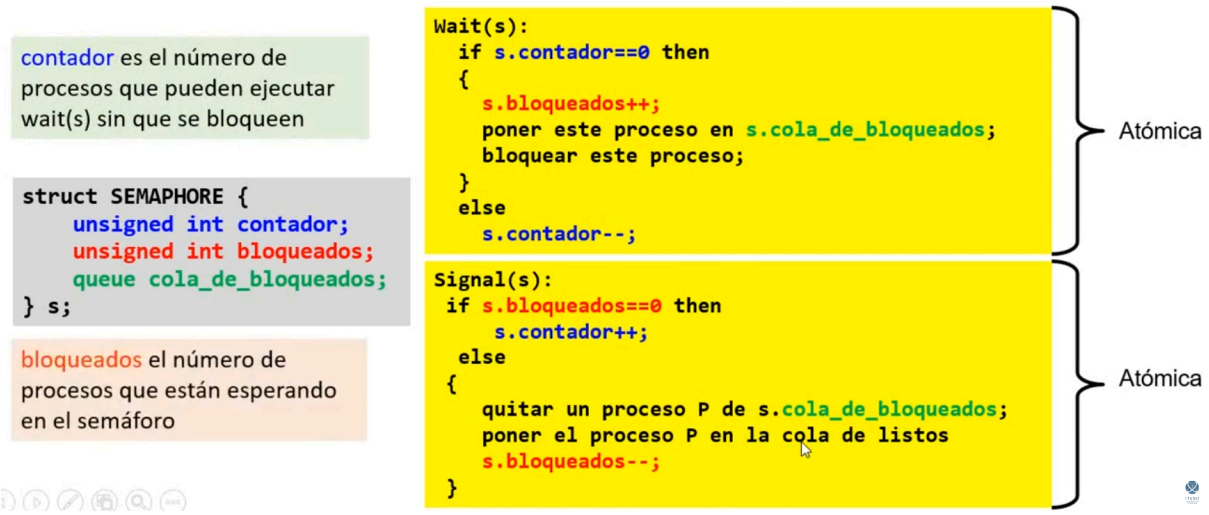


SEMÁFORO ENTERO

1. Para semáforos que pueden tomar valores negativos



2. Para semáforos que no pueden tomar valores negativos



- MONITORES

Los monitores son estructuras de un lenguaje de programación que ofrecen una funcionalidad equivalente a la de los semáforos y son más fáciles de controlar.

Un monitor es un tipo de objeto que tiene la característica de que solo un proceso puede estar ejecutando cualquiera de sus métodos.

- Otro proceso que haya invocado al monitor quedará bloqueado mientras espera a que el monitor esté disponible

Funciones para operar las colas de condición

Cwait(condición)

- Suspende la ejecución del proceso llamado bajo la condición.
- El monitor está disponible para ser usado por otro proceso.

Csignal(condición)

- Reaunuda la ejecución de algún proceso suspendido por un `cwait` con la misma condición.
- Si hay varios procesos elige uno de ellos
- Si no hay ninguno no hace nada.
  - Si un proceso de un monitor ejecuta un `csignal` y no hay tareas esperando en la variable de condición, el `csignal` se pierde.

```
monitor buffer_acotado
{
        char buffer[TAM_BUFFER];        // Espacio para N elementos
        int sigent,sigsal;      // Apuntadores al buffer
        int contador;           // Número de elementos en el buffer
        condition no_lleno,no_vacio;    // Para sincronización

        añadir(char x) {
                if (contador==TAM_BUFFER) cwait(no_lleno);
                                // Buffer lleno; se impide producir
                buffer[sigent]=x;
                sigent=sigent+1 % TAM_BUFFER;
                contador++;// Un elemento más en el buffer
                csignal(no_vacío);// Reanudar un consumidor en espera
        }
        tomar(char x)
        {
                if (contador==0) cwait(no_vacío);
                                // Buffer vacío; se impide consumir
                x=buffer[sigsal];
                sigsal=(sigsal+1) % TAM_BUFFER;
                contador--;
                                // Un elemento menos en el buffer
                csignal(no_lleno); // Reanudar un productor en espera
        }
    initialize
    {           // Cuerpo del monitor
                sigent=0;sigsal=0;contador=0; // Buffer inicia vacío
    }
} // Termina el monitor
```