Sistemas operativos

Concurrencia

Los temas centrales del diseño de sistemas operativos están todos relacionados con la gestión de procesos e hilos:

- **Multiprogramación**. Gestión de múltiples procesos dentro de un sistema monoprocesador.
- **Multiprocesamiento**. Gestión de múltiples procesos dentro de un multiprocesador.
- **Procesamiento distribuido**. Gestión de múltiples procesos que ejecutan sobre múltiples sistemas de cómputo distribuidos.

La concurrencia es fundamental en todas estas áreas y en el diseño del sistema operativo. La concurrencia abarca varios aspectos, entre los cuales están la comunicación entre procesos y la compartición de, o competencia por, recursos, la sincronización de actividades de múltiples procesos y la reserva de tiempo de procesador para los procesos.

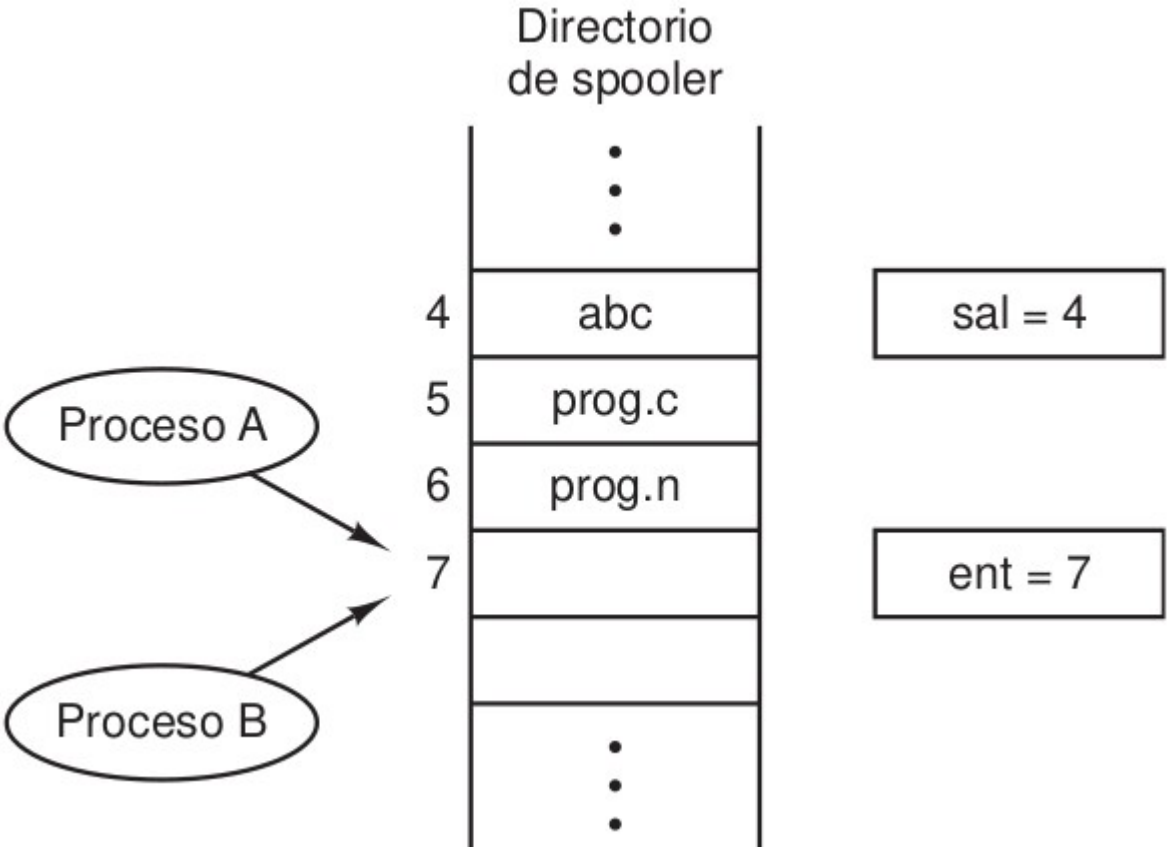
Debemos entender que todos estos asuntos no sólo suceden en el entorno del multiprocesamiento y el procesamiento distribuido, sino también en sistemas monoprocesador multiprogramados.

Condiciones de carrera.

Para ver cómo funciona la comunicación entre procesos en la práctica, consideremos un ejemplo simple pero común: 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.

Imagine que nuestro directorio de spooler tiene una cantidad muy grande de ranuras, numeradas como 0, 1, 2, ..., cada una de ellas capaz de contener el nombre de un archivo. Imagine también que hay dos variables compartidas: sal, que apunta al siguiente archivo a imprimir, y ent, que apunta a la siguiente ranura libre en el directorio. Estas dos variables podrían mantenerse muy bien en un archivo de dos palabras disponible para todos los procesos.

En cierto momento, las ranuras de la 0 a la 3 están vacías (ya se han impreso los archivos) y las ranuras de la 4 a la 6 están llenas (con los nombres de los archivos en la cola de impresión). De manera más o menos simultánea, los procesos A y B deciden que desean poner en cola un archivo para imprimirlo. Esta situación se muestra en la siguiente figura.



Dos procesos desean acceder a la memoria compartida al mismo tiempo.

Podría ocurrir lo siguiente. El proceso A lee ent y guarda el valor 7 en una variable local, llamada siguiente_ranura_libre. Justo entonces ocurre una interrupción de reloj y la CPU decide que el proceso A se ha ejecutado durante un tiempo suficiente, por lo que conmuta al proceso B. El proceso B también lee ent y también obtiene un 7. De igual forma lo almacena en su variable local siguiente_ranura_libre. En este instante, ambos procesos piensan que la siguiente ranura libre es la 7.

Ahora el proceso B continúa su ejecución. Almacena el nombre de su archivo en la ranura 7 y actualiza ent para que sea 8. Después realiza otras tareas.

En cierto momento el proceso A se ejecuta de nuevo, partiendo del lugar en el que se quedó. Busca en siguiente_ranura_libre, encuentra un 7 y escribe el nombre de su archivo en la ranura 7, borrando el nombre que el proceso B acaba de poner ahí. Luego calcula siguiente_ranura_libre + 1, que es 8 y fija ent para que sea 8.

El directorio de spooler es ahora internamente consistente, por lo que el demonio de impresión no detectará nada incorrecto, pero el proceso B nunca recibirá ninguna salida.

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

Depurar programas que contienen condiciones de carrera no es nada divertido. Los resultados de la mayoría de las ejecuciones de prueba están bien, pero en algún momento poco frecuente ocurrirá algo extraño e inexplicable.

Regiones críticas

¿Cómo evitamos las condiciones de carrera? La clave para evitar problemas aquí y en muchas otras situaciones en las que se involucran la memoria compartida, los archivos compartidos y todo lo demás compartido es buscar alguna manera de prohibir que más de un proceso lea y escriba los datos compartidos al mismo tiempo. Dicho en otras palabras, 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.

La dificultad antes mencionada ocurrió debido a que el proceso B empezó a utilizar una de las variables compartidas antes de que el proceso A terminara con ella.

El problema de evitar las condiciones de carrera también se puede formular de una manera abstracta. Parte del tiempo, un proceso está ocupado realizando cálculos internos y otras cosas que no producen condiciones de carrera. Sin embargo, algunas veces un proceso tiene que acceder a la memoria compartida o a archivos compartidos, o hacer otras cosas críticas que pueden producir carreras.

Esa parte del programa en la que se accede a la memoria compartida se conoce como región crítica o sección crítica. Si pudiéramos ordenar las cosas de manera que dos procesos nunca estuvieran en sus regiones críticas al mismo tiempo, podríamos evitar las carreras.

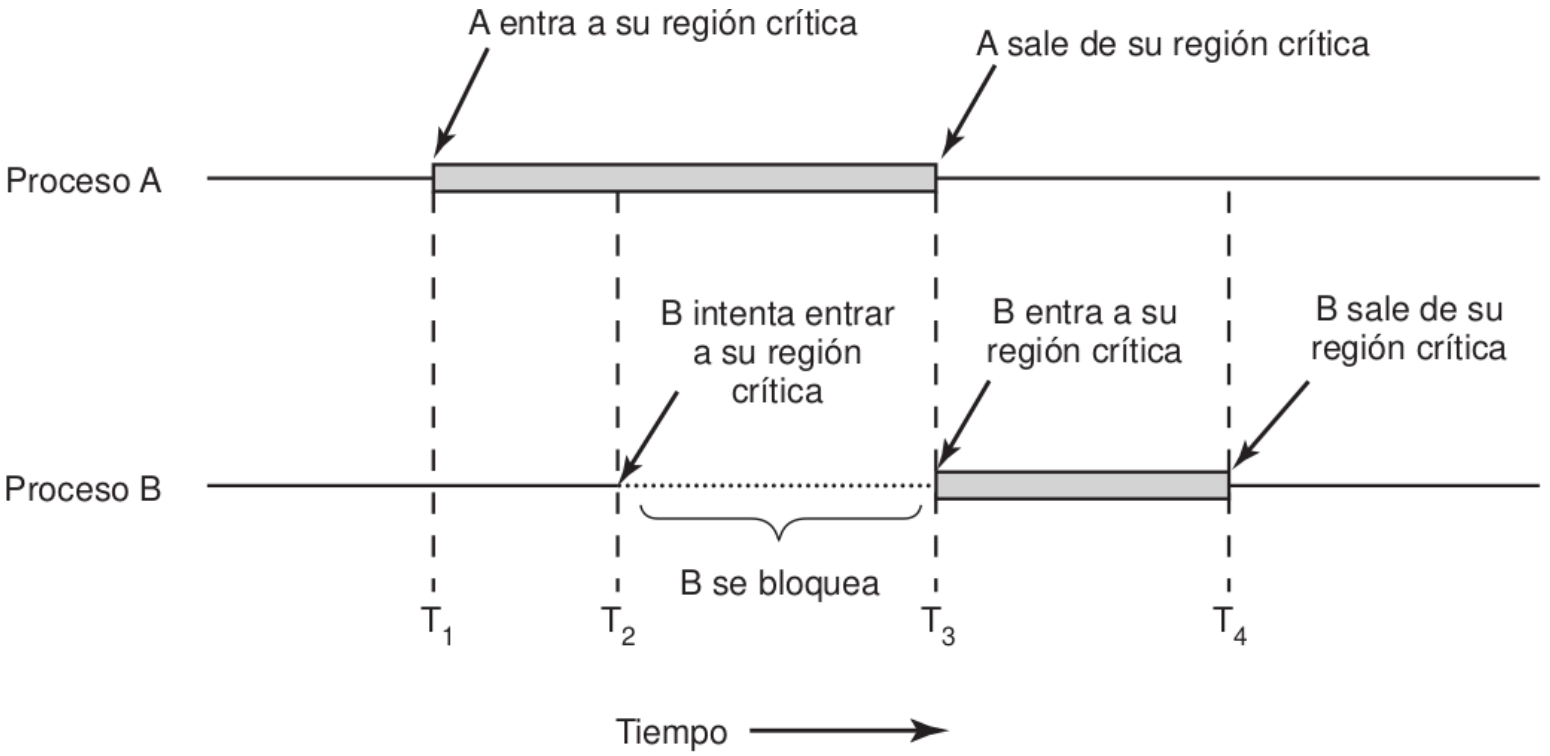
Aunque este requerimiento evita las condiciones de carrera, no es suficiente para que los procesos en paralelo cooperen de la manera correcta y eficiente al utilizar datos compartidos.

Necesitamos cumplir con cuatro condiciones para tener una buena solución:

1. No puede haber dos procesos de manera simultánea dentro de sus regiones críticas.
2. No pueden hacerse suposiciones acerca de las velocidades o el número de CPUs.
3. Ningún proceso que se ejecute fuera de su región crítica puede bloquear otros procesos.
4. Ningún proceso tiene que esperar para siempre para entrar a su región crítica.

En sentido abstracto, el comportamiento que deseamos se muestra en la siguiente figura. Aquí el proceso A entra a su región crítica en el tiempo T1. Un poco después, en el tiempo T2 el proceso B intenta entrar a su región crítica, pero falla debido a que otro proceso ya se encuentra en su región crítica y sólo se permite uno a la vez. En consecuencia, B se suspende temporalmente hasta el tiempo T3 cuando A sale de su región crítica, con lo cual se permite a B entrar de inmediato.

En algún momento dado B sale (en T4) y regresamos a la situación original, sin procesos en sus regiones críticas.


A entra a su región crítica
A sale de su región crítica
Proceso A
B intenta entrar a su región crítica
B entra a su región crítica
B sale de su región crítica
Proceso B
B se bloquea
T1
T2
T3
T4
Tiempo

Exclusión mutua con espera ocupada

Examinaremos varias proposiciones para lograr la exclusión mutua, de manera que mientras un proceso esté ocupado actualizando la memoria compartida en su región crítica, ningún otro proceso puede entrar a su región crítica y ocasionar problemas.

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.

- Deshabilitando interrupciones

En un sistema con un solo procesador, 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.

Por lo general este método es poco atractivo, ya que no es conveniente dar a los procesos de usuario el poder para desactivar las interrupciones. Suponga que uno de ellos lo hiciera y nunca las volviera a activar.

Ése podría ser el fin del sistema; aún más: si el sistema es un multiprocesador (con dos o posiblemente más CPUs), al deshabilitar las interrupciones sólo se ve afectada la CPU que ejecutó la instrucción disable. Las demás continuarán ejecutándose y pueden acceder a la memoria compartida.

Por otro lado, con frecuencia es conveniente para el mismo kernel deshabilitar las interrupciones por unas cuantas instrucciones mientras actualiza variables o listas. La conclusión es que a menudo deshabilitar interrupciones es una técnica útil dentro del mismo sistema operativo, pero no es apropiada como mecanismo de exclusión mutua general para los procesos de usuario.

- Variables de candado

Como segundo intento, busquemos una solución de software. 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.

Por desgracia, esta idea contiene exactamente el mismo error 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.

Se podría pensar que podemos resolver este problema si leemos primero el valor de candado y después lo verificamos de nuevo justo antes de almacenar el nuevo valor en él, pero en realidad eso no ayuda. La condición de carrera se produce ahora si el segundo proceso modifica el candado justo después que el primer proceso haya terminado su segunda verificación.

- Solución de Peterson

En 1981, G.L. Peterson descubrió una manera de lograr la exclusión mutua. El algoritmo de Peterson se muestra a continuación. Este algoritmo consiste de dos procedimientos escritos en ANSI C, lo cual significa que se deben suministrar prototipos para todas las funciones definidas y utilizadas. Sin embargo, para ahorrar espacio no mostraremos los prototipos en este ejemplo ni en los siguientes.

```
#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 */
}
```

Algoritmo de Peterson para lograr la exclusión mutua.

Antes de utilizar las variables compartidas (es decir, antes de entrar a su región crítica), cada proceso llama a entrar_region con su propio número de proceso (0 o 1) como parámetro. Esta llamada hará que espere, si es necesario, hasta que sea seguro entrar. Una vez que haya terminado con las variables compartidas, el proceso llama a salir_region para indicar que ha terminado y permitir que los demás procesos entren, si así lo desea.

Veamos cómo funciona esta solución. Al principio ningún proceso se encuentra en su región crítica. Ahora el proceso 0 llama a entrar_region. Indica su interés estableciendo su elemento del arreglo y fija turno a 0. Como el proceso 1 no está interesado, entrar_region regresa de inmediato.

Si ahora el proceso 1 hace una llamada a entrar_region, se quedará ahí hasta que interesado[0] sea FALSE, un evento que sólo ocurre cuando el proceso 0 llama a salir_region para salir de la región crítica.

Ahora considere el caso en el que ambos procesos llaman a entrar_region casi en forma simultánea. Ambos almacenarán su número de proceso en turno. Cualquier almacenamiento que se haya realizado al último es el que cuenta; el primero se sobrescribe y se pierde. Suponga que el proceso 1 almacena al último, por lo que turno es 1. Cuando ambos procesos llegan a la instrucción while, el proceso 0 la ejecuta 0 veces y entra a su región crítica. El proceso 1 itera y no entra a su región crítica sino hasta que el proceso 0 sale de su región crítica.

- Dormir y despertar

La solución de Peterson es correcta, pero tiene el defecto de requerir la espera ocupada. En esencia, esta solución comprueba si se permite la entrada cuando un proceso desea entrar a su región crítica. Si no se permite, el proceso sólo espera en un ciclo estrecho hasta que se permita la entrada.

Veamos ahora ciertas primitivas de comunicación entre procesos que bloquean en vez de desperdiciar tiempo de la CPU cuando no pueden entrar a sus regiones críticas. 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.

El problema del productor-consumidor

Como ejemplo de la forma en que se pueden utilizar estas primitivas, consideremos el problema del productor-consumidor (conocido también como el problema del búfer limitado). Dos procesos comparten un búfer común, de tamaño fijo. Uno de ellos (el productor) coloca información en el búfer y el otro (el consumidor) la saca (también es posible generalizar el problema de manera que haya m productores y n consumidores, pero sólo consideraremos el caso en el que hay un productor y un consumidor, ya que esta suposición simplifica las soluciones).

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.

Este método suena lo bastante simple, pero produce los mismos tipos de condiciones de carrera que vimos antes con el directorio de spooler. Para llevar la cuenta del número de elementos en el búfer, necesitamos una variable (cuenta). Si el número máximo de elementos que puede contener el búfer es N, el código del productor comprueba primero si cuenta es N. Si lo es, el productor se duerme; si no lo es, el productor agrega un elemento e incrementará cuenta.

El código del consumidor es similar: primero evalúa cuenta para ver si es 0. Si lo es, se duerme; si es distinta de cero, quita un elemento y disminuye el contador cuenta. Cada uno de los procesos también comprueba si el otro se debe despertar y de ser así, lo despierta. El código para el productor y el consumidor se muestra a continuación.

```
#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 */
        }
}
```

Para expresar llamadas al sistema como sleep y wakeup en C, las mostramos como llamadas a rutinas de la biblioteca. No forman parte de la biblioteca estándar de C, pero es de suponer que estarán disponibles en cualquier sistema que tenga realmente estas llamadas al sistema.

Los procedimientos insertar_elemento y quitar_elemento, que no se muestran aquí, se encargan de colocar elementos en el búfer y sacarlos del mismo.

La condición de carrera puede ocurrir debido a que el acceso a cuenta no está restringido. Es posible que ocurra la siguiente situación: el búfer está vacío y el consumidor acaba de leer cuenta para ver si es 0. En ese instante, el planificador decide detener al consumidor en forma temporal y empieza a ejecutar el productor. El productor inserta un elemento en el búfer, incrementa cuenta y observa que ahora es 1. Razonando que cuenta era antes 0, y que por ende el consumidor debe estar dormido, el productor llama a wakeup para despertar al consumidor.

Por desgracia, el consumidor todavía no está lógicamente dormido, por lo que la señal para despertarlo se pierde. Cuando es turno de que se ejecute el consumidor, evalúa el valor de cuenta que leyó antes, encuentra que es 0 y pasa a dormirse. Tarde o temprano el productor llenará el búfer y también pasará a dormirse. Ambos quedarán dormidos para siempre.

La esencia del problema aquí es que una señal que se envía para despertar a un proceso que no está dormido (todavía) se pierde. Si no se perdiera, todo funcionaría. 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. Más adelante, cuando el proceso intenta pasar a dormir, si el bit de espera de despertar está encendido, se apagará pero el proceso permanecerá despierto. Este bit es una alcancía para almacenar señales de despertar.

Aunque el bit de espera de despertar logra su cometido en este ejemplo simple, es fácil construir ejemplos con tres o más procesos en donde un bit de espera de despertar es insuficiente.

- Semáforos

En 1965 E. W. Dijkstra sugirió el uso de una variable entera para contar el número de señales de despertar, guardadas para un uso futuro. En su propuesta introdujo un nuevo tipo de variable, al cual él le 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.

Un semáforo S es una variable entera a la que, dejando aparte la inicialización, sólo se accede mediante dos operaciones atómicas estándar: wait() y signal() (generalizaciones de sleep y wakeup, respectivamente). Originalmente, la operación wait() se denominaba P (del término holandés proberen, probar); mientras que signal() se denominaba originalmente V (verhogen, incrementar).

En algunos textos las operaciones wait y signal se llaman down y up respectivamente.

La definición de wait() es la que sigue:

```
wait(S) {
        while (S <= 0)
          ; // espera ocupada
        S--;
}
```

La definición de signal() es:

```
signal(S) {
        S++;
}
```

Todas las modificaciones del valor entero del semáforo en las operaciones wait() y signal() deben ejecutarse de forma indivisible. Es decir, cuando un proceso modifica el valor del semáforo, ningún otro proceso puede modificar simultáneamente el valor de dicho semáforo.

Además, en el caso de wait(), la prueba del valor entero de S (S <= 0), y su posible modificación (S--) también se deben ejecutar sin interrupción.

Implementación

La principal desventaja de la definición de semáforo dada aquí es que requiere una espera activa. Mientras un proceso está en su sección crítica, cualquier otro proceso que intente entrar en su sección crítica debe ejecutar continuamente un bucle en el código de entrada. Este bucle continuo plantea claramente un problema en un sistema real de multiprogramación, donde una sola CPU se comparte entre muchos procesos. La espera activa desperdicia ciclos de CPU que algunos otros procesos podrían usar de forma productiva.

Este tipo de semáforo también se denomina cerrojo mediante bucle sin fin (spinlock), ya que el proceso “permanece en un bucle sin fin” en espera de adquirir el cerrojo. (Los cerrojos mediante bucle sin fin tienen una ventaja y es que no requiere ningún cambio de contexto cuando un proceso tiene que esperar para adquirir un cerrojo. Los cambios de contexto pueden llevar un tiempo considerable. Por tanto, cuando se espera que los cerrojos se mantengan durante un período de tiempo corto, los cerrojos mediante bucle sin fin pueden resultar útiles; por eso se emplean a menudo en los sistemas multiprocesador, donde un hilo puede “ejecutar un bucle” sobre un procesador mientras otro hilo ejecuta su sección crítica en otro procesador).

Para evitar la necesidad de la espera activa, podemos modificar la definición de las operaciones de semáforo wait() y signal(). Cuando un proceso ejecuta la operación wait() y determina que el valor del semáforo no es positivo, tiene que esperar. Sin embargo, en lugar de entrar en una espera activa, el proceso puede bloquearse a sí mismo. La operación de bloqueo coloca al proceso en una cola de espera asociada con el semáforo y el estado del proceso pasa al estado de espera. A continuación, el control se transfiere al planificador de la CPU, que selecciona otro proceso para su ejecución.

Un proceso bloqueado, que está esperando en un semáforo S, debe reiniciarse cuando algún otro proceso ejecuta una operación signal(). El proceso se reinicia mediante una operación wakeup(), que cambia al proceso del estado de espera al estado de listo. El proceso se coloca en la cola de procesos listos. (La CPU puede o no conmutar del proceso en ejecución al proceso que se acaba de pasar al estado de listo, dependiendo del algoritmo de planificación de la CPU.)

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

Los semáforos resuelven el problema de pérdida de señales de despertar, como se muestra en el siguiente código. Para que funcionen de manera correcta, es esencial que se implementen de una forma indivisible. Lo normal es implementar signal y wait como llamadas al sistema, en donde el sistema operativo deshabilita brevemente todas las interrupciones, mientras evalúa el semáforo, lo actualiza y pone el proceso a dormir, si es necesario.

Como todas estas acciones requieren sólo unas cuantas instrucciones, no hay peligro al deshabilitar las interrupciones. Si se utilizan varias CPUs, cada semáforo debe estar protegido por una variable de candado para asegurar que sólo una CPU a la vez pueda examinar el semáforo.

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. Al principio llenas es 0, vacías es igual al número de ranuras en el búfer y mutex es 1. Los semáforos que se inicializan a 1 y son utilizados por dos o más procesos para asegurar que sólo uno de ellos pueda entrar a su región crítica en un momento dado se llaman semáforos binarios.

Si cada proceso realiza una operación wait() justo antes de entrar a su región crítica y una operación signal() justo después de salir de ella, se garantiza la exclusión mutua.

En el código de ejemplo, en realidad hemos utilizado semáforos de dos maneras distintas. El semáforo mutex se utiliza para la exclusión mutua. Está diseñado para garantizar que sólo un proceso pueda leer o escribir en el búfer y sus variables asociadas en un momento dado.

El otro uso de los semáforos es para la sincronización. Los semáforos vacías y llenas se necesitan para garantizar que ciertas secuencias de eventos ocurran o no. En este caso, aseguran que el productor deje de ejecutarse cuando el búfer esté lleno y que el consumidor deje de ejecutarse cuando el búfer esté vacío. Este uso es distinto de la exclusión mutua.

```
#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 */
        }
}
```

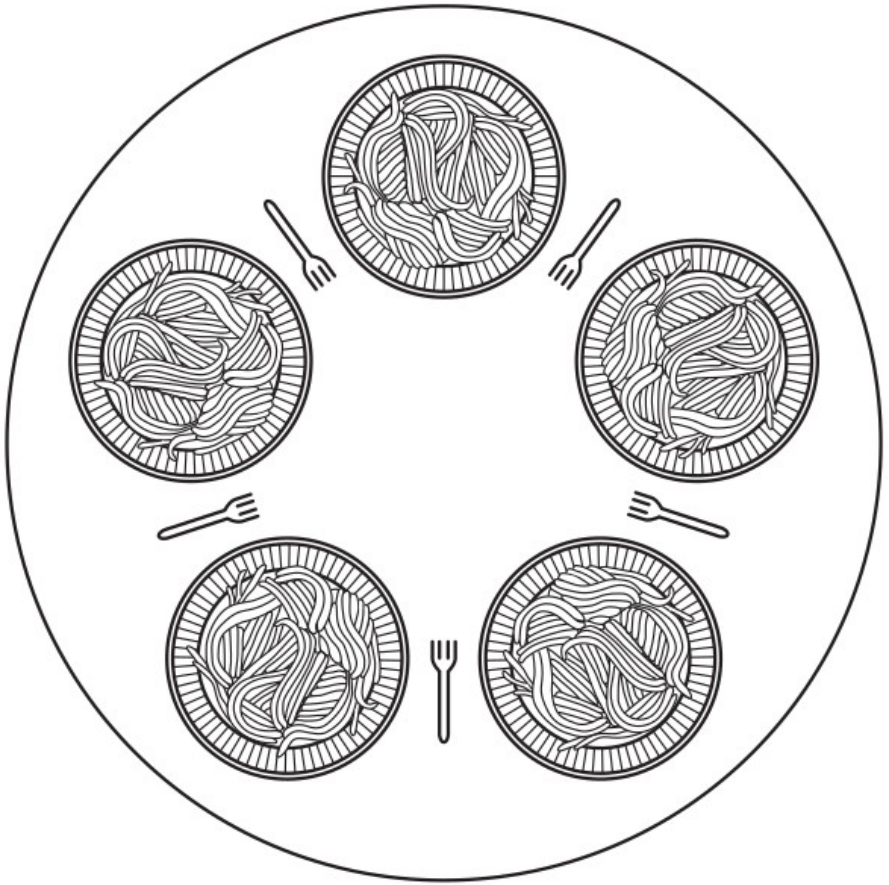
Problemas clásicos de comunicación entre procesos (IPC = Inter-Process Communication)

El problema de los filósofos comelones

En 1965, Dijkstra propuso y resolvió un problema de sincronización al que llamó el problema de los filósofos comelones. Desde ese momento, todos los que inventaban otra primitiva de sincronización se sentían obligados a demostrar qué tan maravillosa era esa nueva primitiva, al mostrar con qué elegancia resolvía el problema de los filósofos comelones.

Este problema se puede enunciar simplemente de la siguiente manera. 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 siguiente figura.



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?

El siguiente código muestra la solución obvia. El procedimiento tomar_tenedor espera hasta que el tenedor específico esté disponible y luego lo toma. Por desgracia, la solución obvia está mal. Suponga que los cinco filósofos toman sus tenedores izquierdos al mismo tiempo. Ninguno podrá tomar sus tenedores derechos y habrá un interbloqueo.

```
#define N 5                                 /* número de filósofos */

void filosofo(int i)                        /* i: número de filósofo, de 0 a 4 */
{
  while(TRUE){
        pensar();                           /* el filósofo está pensando */
        tomar_tenedor(i);                   /* toma tenedor izquierdo */
        tomar_tenedor((i+1) % N);           /* toma tenedor derecho; % es el operador módulo */
        comer();                            /* come espagueti */
        poner_tenedor(i);                   /* pone tenedor izquierdo de vuelta en la mesa */
        poner_tenedor((i+1) % N);           /* pone tenedor derecho de vuelta en la mesa */
  }
}
```

Podríamos modificar el programa de manera que después de tomar el tenedor izquierdo, el programa compruebe para ver si el tenedor derecho está disponible. Si no lo está, el filósofo regresa el tenedor izquierdo, espera cierto tiempo y después repite todo el proceso. Esta proposición falla también, aunque por una razón distinta.

Con un poco de mala suerte, todos los filósofos podrían iniciar el algoritmo en forma simultánea, tomarían sus tenedores izquierdos, verían que sus tenedores derechos no están disponibles, regresarían sus tenedores izquierdos, esperarían, volverían a tomar sus tenedores izquierdos al mismo tiempo y así en lo sucesivo, eternamente.

Una situación como ésta, en la que todos los programas continúan ejecutándose en forma indefinida pero no progresan se conoce como inanición (starvation).

Ahora podríamos pensar que si los filósofos sólo esperan por un tiempo aleatorio en vez de esperar durante el mismo tiempo al no poder adquirir el tenedor derecho, la probabilidad de que todo continúe bloqueado durante incluso una hora es muy pequeña. Esta observación es verdad y en casi todas las aplicaciones intentar de nuevo en un tiempo posterior no representa un problema.

Sin embargo, en algunas cuantas aplicaciones sería preferible una solución que funcione siempre y que no pueda fallar debido a una serie improbable de números aleatorios. Piense acerca del control de la seguridad en una planta de energía nuclear.

La solución que se presenta en el siguiente código está libre de interbloqueos y permite el máximo paralelismo para un número arbitrario de filósofos. Utiliza un arreglo llamado estado para llevar el registro de si un filósofo está comiendo, pensando o hambriento (tratando de adquirir tenedores).

Un filósofo sólo se puede mover al estado de comer si ningún vecino está comiendo. Los i vecinos del filósofo se definen mediante las macros IZQUIERDO y DERECHO. En otras palabras, si i es 2, IZQUIERDO es 1 y DERECHO es 3.

El programa utiliza un arreglo de semáforos, uno por cada filósofo, de manera que los filósofos hambrientos puedan bloquearse si los tenedores que necesitan están ocupados. Observe que cada proceso ejecuta el procedimiento filosofo como su código principal, pero los demás procedimientos (tomar_tenedores, poner_tenedores y probar) son ordinarios y no procesos separados.

```
#define N                   5                   /* número de filósofos */
#define IZQUIERDO           (i+N-1)%N           /* número del vecino izquierdo de i */
#define DERECHO             (i+1)%N             /* número del vecino derecho de i */
#define PENSANDO            0                   /* el filósofo está pensando */
#define HAMBRIENTO          1                   /* el filósofo trata de obtener los tenedores */
#define COMIENDO            2                   /* el filósofo está comiendo */
typedef int semaforo;                           /* los semáforos son un tipo especial de int */
int estado[N];                                  /* arreglo que lleva registro del estado de todos*/
semaforo mutex = 1;                             /* exclusión mutua para las regiones críticas */
semaforo s[N];                                  /* un semáforo por filósofo */

void filosofo(int i)                            /* i: número de filósofo, de 0 a N−1 */
{
      while(TRUE){                              /* se repite en forma indefinida */
            pensar();                           /* el filósofo está pensando */
            tomar_tenedores(i);                 /* adquiere dos tenedores o se bloquea */
            comer();                            /* come espagueti */
            poner_tenedores(i);                 /* pone de vuelta ambos tenedores en la mesa */
      }
}
```

```
void tomar_tenedores(int i)                /* i: número de filósofo, de 0 a N−1 */
{
        down(&mutex);                      /* entra a la región crítica */
        estado[i] = HAMBRIENTO;            /* registra el hecho de que el filósofo i está hambriento */
        probar(i);                         /* trata de adquirir 2 tenedores */
        up(&mutex);                        /* sale de la región crítica */
        down(&s[i]);                       /* se bloquea si no se adquirieron los tenedores */
}

void poner_tenedores(i)                    /* i: número de filósofo, de 0 a N–1 */
{
        down(&mutex);                      /* entra a la región crítica */
        estado[i] = PENSANDO;              /* el filósofo terminó de comer */
        probar(IZQUIERDO);                 /* verifica si el vecino izquierdo puede comer ahora */
        probar(DERECHO);                   /* verifica si el vecino derecho puede comer ahora */
        up(&mutex);                        /* sale de la región crítica */
}
```

```
void probar(i)                  /* i: número de filósofo, de 0 a N−1 */
{
        if (estado[i] == HAMBRIENTO && estado[IZQUIERDO] != COMIENDO && estado[DERECHO] != COMIENDO) {
                estado[i] = COMIENDO;
                up(&s[i]);
        }
}
```

El problema de los lectores y escritores

Otro problema famoso es el de los lectores y escritores (Courtois y colaboradores, 1971), que 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?.

En la solución que se presenta a continuación, 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.

```
typedef int semaforo;                    /* use su imaginación */
semaforo mutex=1;                        /* controla el acceso a 'cl' */
semaforo bd=1;                           /* controla el acceso a la base de datos */
int cl=0;                                /* # de procesos que leen o desean hacerlo */

void lector(void)
{
        while(TRUE){                     /* se repite de manera indefinida */
            down(&mutex);                /* obtiene acceso exclusivo a 'cl' */
            cl = cl + 1;                 /* ahora hay un lector más */
            if (cl == 1) down(&bd);      /* si este es el primer lector ... */
            up(&mutex);                  /* libera el acceso exclusivo a 'cl' */
            leer_base_de_datos();        /* accede a los datos */
            down(&mutex);                /* obtiene acceso exclusivo a 'cl' */
            cl = cl – 1;                 /* ahora hay un lector menos */
            if (cl == 0) up(&bd);        /* si este es el último lector ... */
            up(&mutex);                  /* libera el acceso exclusivo a 'cl' */
            usar_lectura_datos();        /* región no crítica */
        }
}
```

```
void escritor(void)
{
        while(TRUE){                    /* se repite de manera indefinida */
        pensar_datos();                 /* región no crítica */
        down(&bd);                      /* obtiene acceso exclusivo */
        escribir_base_de_datos();       /* actualiza los datos */
        up(&bd);                        /* libera el acceso exclusivo */
    }
}
```

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.

Interbloqueos

Los sistemas computacionales están llenos de recursos que pueden ser utilizados por sólo un proceso a la vez. En consecuencia, todos los sistemas operativos tienen la habilidad de otorgar (en forma temporal) a un proceso el acceso exclusivo a ciertos recursos.

Para muchas aplicaciones, un proceso necesita acceso exclusivo no sólo a un recurso, sino a varios.

Suponga que cada uno de dos procesos quiere grabar un documento digitalizado en un CD. El proceso A pide permiso para utilizar el escáner y se le otorga. El proceso B se programa de manera distinta y solicita primero el grabador de CDs, y también se le otorga. Ahora A pide el grabador de CDs, pero la petición se rechaza hasta que B lo libere. Por desgracia, en vez de liberar el grabador de CD, B pide el escáner. En este punto ambos procesos están bloqueados y permanecerán así para siempre. A esta situación se le conoce como interbloqueo.

Los interbloqueos también pueden ocurrir entre máquinas. Por ejemplo, muchas oficinas tienen una red de área local con muchas computadoras conectadas. A menudo, los dispositivos como impresoras, escáneres, etc., se conectan a la red como recursos compartidos, disponibles para cualquier usuario en cualquier equipo.

Las situaciones más complicadas pueden ocasionar interbloqueos que involucren a tres, cuatro o más dispositivos y usuarios.

Los interbloqueos pueden ocurrir en una variedad de situaciones, además de solicitar dispositivos de E/S dedicados. Por ejemplo, en un sistema de bases de datos, un programa puede tener que bloquear varios registros que esté utilizando para evitar condiciones de competencia. Si el proceso A bloquea el registro R1 y el proceso B bloquea el registro R2, y después cada proceso trata de bloquear el registro del otro, también tenemos un interbloqueo. Por ende, los interbloqueos pueden ocurrir en los recursos de hardware o de software.

Recursos

Los interbloqueos pueden ocurrir cuando a los procesos se les otorga acceso exclusivo a los dispositivos, registros de datos, archivos, etcétera. Para que el análisis sobre los interbloqueos sea lo más general posible, nos referiremos a los objetos otorgados como recursos. Un recurso puede ser un dispositivo de hardware (por ejemplo, una impresora) o una pieza de información (como un registro bloqueado en una base de datos).

Por lo general, una computadora tendrá muchos recursos que se pueden adquirir. Para algunos recursos puede haber disponibles varias instancias idénticas, como tres unidades de cinta. Cuando hay disponibles varias copias de un recurso, se puede utilizar sólo una de ellas para satisfacer cualquier petición de ese recurso. En resumen, un recurso es cualquier cosa que se debe adquirir, utilizar y liberar con el transcurso del tiempo.

- Recursos apropiativos y no apropiativos

Los recursos son de dos tipos: apropiativos y no apropiativos. Un recurso apropiativo es uno que se puede quitar al proceso que lo posee sin efectos dañinos. La memoria es un ejemplo de un recurso apropiativo. Por ejemplo, considere un sistema con 256 MB de memoria de usuario, una impresora y dos procesos de 256 MB, cada uno de los cuales quiere imprimir algo. El proceso A solicita y obtiene la impresora, y después empieza a calcular los valores a imprimir. Antes de terminar con el cálculo, excede su quantum de tiempo y se intercambia por el otro proceso.

Ahora el proceso B se ejecuta y trata (sin éxito) de adquirir la impresora: se crea una situación potencial de interbloqueo, ya que A tiene la impresora y B tiene la memoria, y ninguno puede proceder sin el recurso que el otro posee. Por fortuna, es posible apropiarse de la memoria de B al intercambiarlo y colocar el proceso A de vuelta. Ahora A se puede ejecutar, realizar su impresión y después liberar la impresora. Así no ocurre el interbloqueo.

Por el contrario, un recurso no apropiativo es uno que no se puede quitar a su propietario actual sin hacer que el cómputo falle. Si un proceso ha empezado a quemar un CD-ROM y tratamos de quitarle de manera repentina el grabador de CD y otorgarlo a otro proceso, se obtendrá un CD con basura. Los grabadores de CD no son apropiativos en un momento arbitrario.

En general, los interbloqueos involucran a los recursos no apropiativos. Los interbloqueos potenciales que pueden involucrar a los recursos apropiativos por lo general se pueden resolver mediante la reasignación de los recursos de un proceso a otro.

La secuencia de eventos requerida para utilizar un recurso se proporciona a continuación, en un formato abstracto.

1. Solicitar el recurso.
2. Utilizar el recurso.
3. Liberar el recurso.

Si el recurso no está disponible cuando se le solicita, el proceso solicitante se ve obligado a esperar. En algunos sistemas operativos, el proceso se bloquea de manera automática cuando falla la solicitud de un recurso, y se despierta cuando el recurso está disponible. En otros sistemas, la solicitud falla con un código de error y depende del proceso que hizo la llamada decidir si va a esperar un poco e intentar de nuevo.

Un proceso al que se le ha negado la petición de un recurso por lo general permanece en un ciclo estrecho solicitando el recurso, después pasa al estado inactivo y después intenta de nuevo.

Aunque este proceso no está bloqueado, para toda intención y propósito es como si lo estuviera, debido a que no puede realizar ningún trabajo útil.

Adquisición de recursos

Para ciertos tipos de recursos, como los registros de una base de datos, es responsabilidad de los procesos de usuario administrar su uso. Una manera de permitir que los usuarios administren los recursos es asociar un semáforo con cada recurso. Estos semáforos se inicializan con 1. Se pueden utilizar mutexes de igual forma. Los tres pasos antes listados se implementan como una operación down en el semáforo para adquirir el recurso, usarlo y finalmente realizar una operación up en el recurso para liberarlo. Estos pasos se muestran a continuación.

```
typedef int semaforo;
semaforo recurso_1;


void proceso_A(void) {
   down(&recurso_1);
   usar_recurso_1();
   up(&recurso_1);
}
```

(a)

```
typedef int semaforo;
semaforo recurso_1;
semaforo recurso_2;

void proceso_A(void) {
   down(&recurso_1);
   down(&recurso_2);
   usar_ambos_recursos();
   up(&recurso_2);
   up(&recurso_1);
}
```

(b)

Algunas veces los procesos necesitan dos o más recursos. Se pueden adquirir de manera secuencial, como se muestra en la columna (b) de la diapositiva anterior. Si se necesitan más de dos recursos, sólo se adquieren uno después del otro.

Hasta ahora todo está bien. Mientras sólo haya un proceso involucrado, todo funciona sin problemas. Desde luego que con sólo un proceso, no hay necesidad de adquirir formalmente los recursos, ya que no hay competencia por ellos.

Ahora consideremos una situación con dos procesos, A y B, y dos recursos. En el código que se presentará a continuación se ilustran dos escenarios. En la columna (a), ambos procesos piden los recursos en el mismo orden.

En la columna (b), los piden en un orden distinto. Esta diferencia puede parecer insignificante, pero no lo es.

En la columna (a), uno de los procesos adquirirá el primer recurso antes del otro. Después ese proceso adquirirá con éxito el segundo recurso y realizará su trabajo. Si el otro proceso intenta adquirir el recurso 1 antes de que sea liberado, el otro proceso simplemente se bloqueará hasta que esté disponible.

En la columna (b), la situación es distinta. Podría ocurrir que uno de los procesos adquiera ambos recursos y en efecto bloquee al otro proceso hasta que termine. Sin embargo, también podría ocurrir que el proceso A adquiera el recurso 1 y el proceso B adquiera el recurso 2. Ahora cada uno se bloqueará al tratar de adquirir el otro. Ninguno de los procesos se volverá a ejecutar. Esa situación es un interbloqueo.

```
typedef int semaforo;
  semaforo recurso_1;
  semaforo recurso_2;

  void proceso_A(void) {
    down(&recurso_1);
    down(&recurso_2);
    usar_ambos_recursos();
    up(&recurso_2);
    up(&recurso_1);
  }

  void proceso_B(void) {
    down(&recurso_1);
    down(&recurso_2);
    usar_ambos_recursos();
    up(&recurso_2);
    up(&recurso_1);
  }
```

(a)

```
semaforo recurso_1;
semaforo recurso_2;

void proceso_A(void) {
  down(&recurso_1);
  down(&recurso_2);
  usar_ambos_recursos();
  up(&recurso_2);
  up(&recurso_1);
}

void proceso_B(void) {
  down(&recurso_2);
  down(&recurso_1);
  usar_ambos_recursos();
  up(&recurso_1);
  up(&recurso_2);
}
```

(b)

Aquí podemos ver cómo lo que parece ser una diferencia insignificante en el estilo de codificación (cuál recurso adquirir primero) constituye la diferencia entre un programa funcional y uno que falle de una manera difícil de detectar.