

CAPÍTULO 2 PROCESOS.

2.1 Introducción.

El concepto central de cualquier SO es el proceso.

Un **proceso** es la instancia en ejecución de un programa.

La multiprogramación es diferente de la capacidad de una computadora de hacer más de una cosa al mismo tiempo.

Existe confusión de conceptos, entre ellos el de Paralelismo vs. Seudo paralelismo.

2.2 El modelo de procesos.

En este modelo, todo el software ejecutable de la computadora está organizado en una serie de **procesos secuenciales** o **procesos**.

Cada proceso incluye los valores actuales del contador de programa, los registros y las variables, además de que conceptualmente, tienen su propia CPU virtual.

La capacidad de conmutación entre procesos ejecutándose en una misma CPU dando la ilusión de paralelismo se denomina **multiprogramación**.

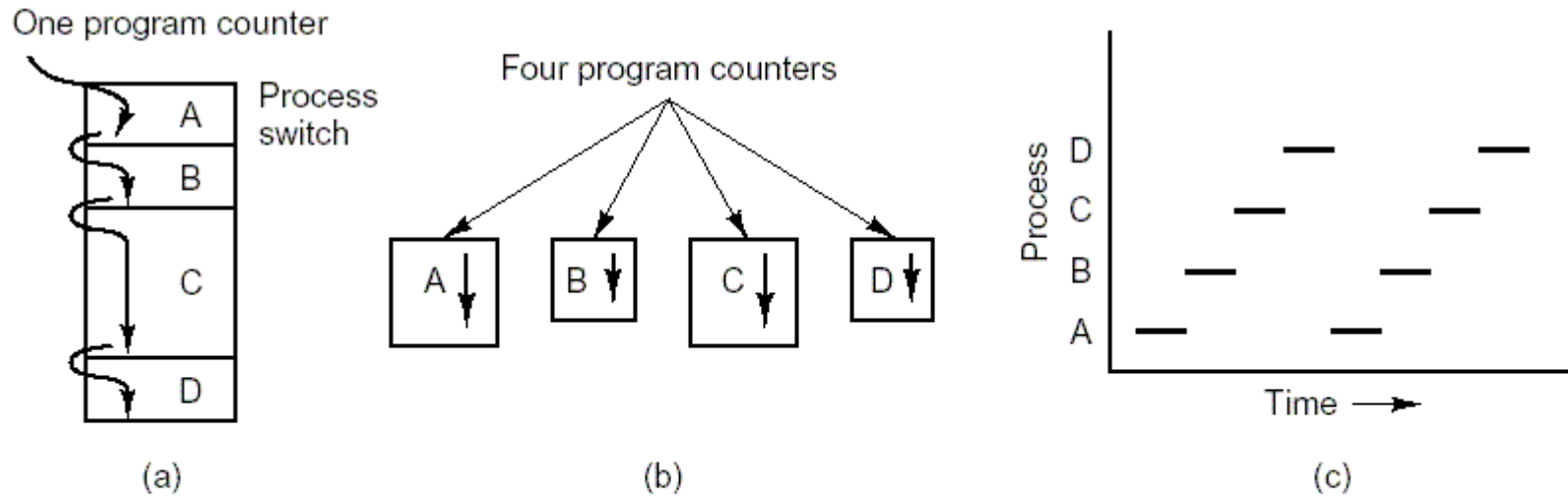


Figure 2-1. (a) Multiprogramming of four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at any instant.

La rapidez con la que un proceso realiza sus cálculos no es uniforme ni reproducible exactamente debido a la conmutación,

por lo que no se debe hacer ninguna suposición respecto al tiempo de los procesos.

Como ya se ha comentado puede existir un árbol de procesos constituyendo así una **jerarquía de procesos**.

¿Cuál es el proceso inicial de UNIX?

Aunque cada proceso es una entidad independiente, a menudo necesitan interactuar con otros procesos (shell por ejemplo).

Las velocidades de los procesos varían por diversas circunstancias, y es posible que aunque un proceso esté listo para ejecutarse, no tenga datos que procesar ¿Que debe hacerse?

Cuando un proceso se bloquea, lo hace porque le es imposible continuar lógicamente.

En otra situación, un proceso puede no ser ejecutado porque el SO ha decidido asignarle la CPU a otro proceso.

En base a lo anterior, un proceso puede estar en uno de tres estados diferentes:

1. En ejecución (esta usando la CPU).
2. Listo (listo pero suspendido porque otro proceso se está ejecutando).
3. Bloqueado (está esperando algún evento externo).

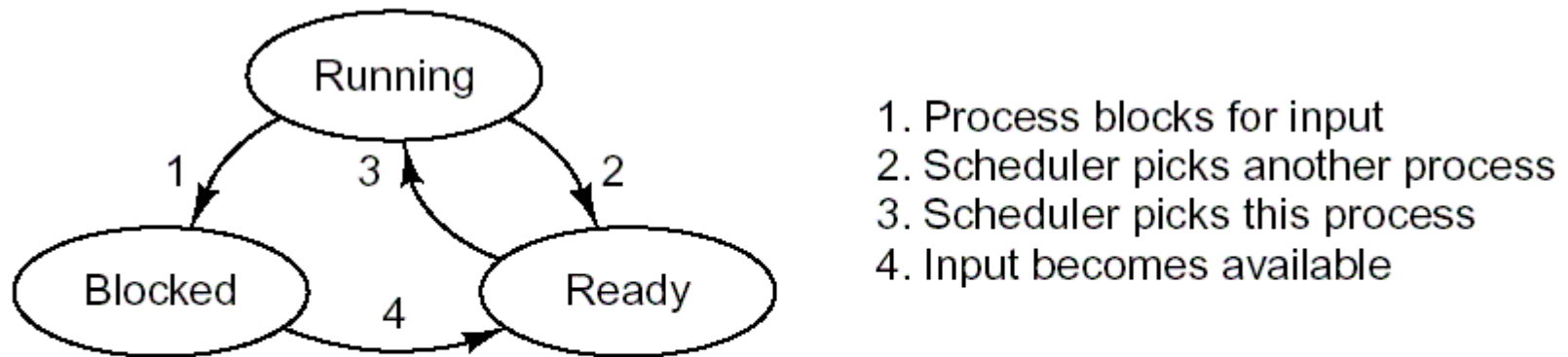


Figure 2-2. A process can be in running, blocked, or ready state. Transitions between these states are as shown.

2.2.1 Implementación de procesos.

El modelo de procesos es mantenido por el SO en una **tabla de procesos**, que es un arreglo de estructuras con una entrada por cada proceso.

Cada entrada contiene información acerca del estado del proceso: contador de programa, apuntador de pila, reparto de memoria, archivos abiertos, información de planificación, etc., de tal manera que el proceso “no se entere” de que fue interrumpido y regrese a su ejecución como si nada hubiera pasado.

Un aspecto del manejo de interrupciones genérico se ilustra a continuación.

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler marks waiting task as ready.
7. Scheduler decides which process is to run next.
8. C procedure returns to the assembly code.
9. Assembly language procedure starts up new current process.

Figure 2-5. Skeleton of what the lowest level of the operating system does when an interrupt occurs.

2.2.2 Hilos.

En los procesos “normales” existe un solo hilo de control y un solo contador de programa para cada proceso.

Algunos SO utilizan múltiples **hilos** de control (**procesos ligeros**) dentro de un mismo proceso.

Los hilos pueden usarse en donde se requiera un multiprocesamiento independiente dentro de un mismo proceso: servidor de archivos, navegadores, etc.

En este esquema, algunos de los campos de la tabla de procesos tienen información por hilo y no sólo por proceso.

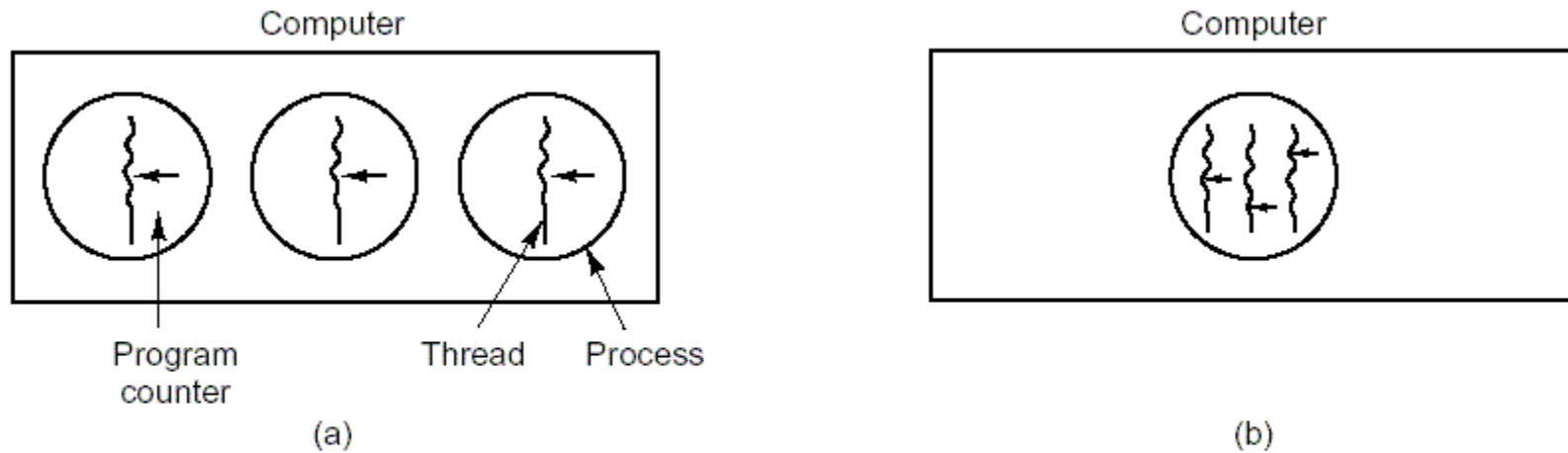


Figure 2-6. (a) Three processes each with one thread. (b) One process with three threads.

En algunos esquemas, el SO no está consciente de la existencia de múltiples hilos por proceso y se manejan en el espacio de usuario (hilos C de Mach).

En los esquemas en los que el SO está consciente de la existencia de múltiples procesos cada hilo se trata como un proceso, con su tabla de procesos y su planificación.

La administración de hilo involucra muchos problemas, considere por ejemplo una llamada al sistema: *fork()*.

2.3 Comunicación entre procesos (IPC).

Los procesos necesitan comunicarse con otros procesos (*shell*).

Esta comunicación es necesaria de preferencia de forma bien estructurada que no utilice interrupciones.

La IPC incurre en problemas que se deben analizar:

1. ¿Cómo puede un proceso pasar información a otro?
2. Dos o más procesos no se deben estorbar al efectuar actividades críticas.
3. Coordinación entre procesos dependientes.

2.3.1 Condiciones de competencia.

Surgen cuando existen recursos compartidos: memoria, archivos, etc.

Ejemplo clásico: spooler de impresión, la variable *out* indica el archivo por imprimir y la variable *in* la siguiente ranura libre, ambas son compartidas.

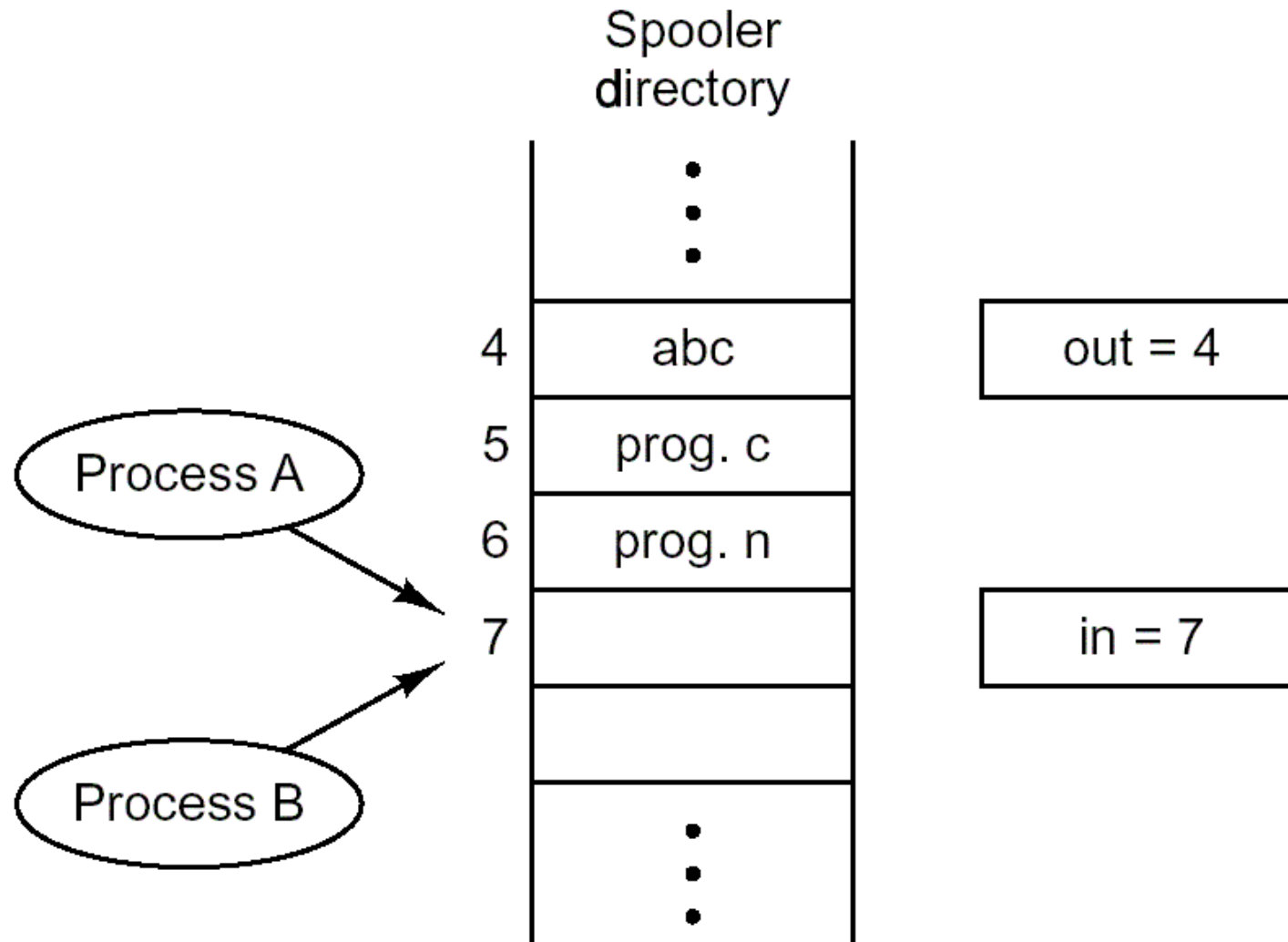


Figure 2-7. Two processes want to access shared memory at the same time.

Una **condición de competencia** se tiene en situaciones en las que dos o más procesos leen o escriben datos compartidos y el resultado final depende de qué proceso se ejecuta, en qué instante y de si termina o no lo que había iniciado.

Es muy difícil determinar en depuración una condición de competencia, si no es que imposible.

¿Cómo evitar las condiciones de competencia?

2.3.2 Secciones críticas.

La **exclusión mutua** es una forma de asegurar que si un proceso está usando un recurso compartido, los otros procesos quedaran excluidos de hacer lo mismo.

La selección de operaciones primitivas apropiadas para lograr la exclusión mutua es un aspecto importante del diseño de cualquier SO.

La parte del programa en la que se hace acceso al recurso compartido se denomina **región crítica** o **sección crítica**.

La idea es entonces, aunque no es lo único que se necesita, que dos o más procesos no puedan estar al mismo tiempo en su región crítica para evitar las condiciones de competencia.

Una buena solución debe cumplir con cuatro condiciones:

1. Dos o más procesos no pueden estar simultáneamente dentro de sus regiones críticas.
2. No debe suponerse nada acerca de la velocidad o el número de CPU.
3. Ningún proceso que se ejecute fuera de su región crítica puede bloquear a otros procesos.

4. Ningún proceso deberá tener que esperar indefinidamente para entrar a su región crítica.

2.3.3 Exclusión mutua con espera activa.

En esta sección se consideran algunas propuestas.

Inhabilitación de interrupciones.

En este esquema, el proceso deshabilita las interrupciones justo después de ingresar a su región crítica, y que justo antes de salir de ella las vuelva a habilitar.

Es un esquema riesgoso y con más de una CPU no funciona.

Variables de candado.

Esta es una solución de software.

Consiste de una variable candado, cuando un proceso intenta ingresar a su región crítica, verifica el valor del candado.

Si el candado es 0 el proceso le asigna 1 y entra a su región crítica.

Si es 1 el proceso espera hasta que el candado vuelve a ser 0.

Esta “solución” incurre en el mismo problema que intenta solucionar (¿Por qué?).

Estricta alternancia.

Esta solución consiste en la prueba continua de una variable hasta que adquiere algún valor y se denomina **espera activa**.

```
while (TRUE) {  
    while (turn != 0) /* wait */ ;  
    critical_region();  
    turn = 1;  
    noncritical_region();  
}
```

(a)

```
while (TRUE) {  
    while (turn != 1) /* wait */ ;  
    critical_region();  
    turn = 0;  
    noncritical_region();  
}
```

(b)

Figure 2-8. A proposed solution to the critical region problem.

Este esquema de alternancia de turnos no es una buena idea cuando un proceso es mucho más lento que el otro (¿Por que?).

Esta solución viola la condición 3. Determine en qué caso.

Solución de Peterson.

Solución propuesta en 1981.

Esta solución cosiste de dos procedimientos.

Analice los procedimientos y determine su comportamiento.

¿Aseguran la exclusión mutua?

¿Qué diferencia hay (si la hubiere) respecto al esquema anterior?

```
#define FALSE      0
#define TRUE      1
#define N    2    /* number of processes */

int turn;          /* whose turn is it? */
int interested[N]; /* all values initially 0 (FALSE) */

void enter_region(int process); /* process is 0 or 1 */
{
    int other;                /* number of the other process */

    other = 1 - process;      /* the opposite of process */
    interested[process] = TRUE; /* show that you are interested */
    turn = process;           /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process) /* process: who is leaving */
{
    interested[process] = FALSE; /* indicate departure from critical region */
}
```

Figure 2-9. Peterson's solution for achieving mutual exclusion.

Debe notarse que esta solución combina la idea de los turnos con la de los candados.

Instrucción TSL (Test and Set Lock).

Esta propuesta requiere ayuda del HW.

La instrucción TSL es una instrucción atómica, y se tiene en computadoras diseñadas para múltiples procesadores.

La idea aquí es usar una variable compartida (*lock*).

TSL lee el contenido de la memoria lo coloca en un registro y luego almacena un valor distinto de cero en esa dirección.

```
enter_region:
    tsl register,lock    | copy lock to register and set lock to 1
    cmp register,#0      | was lock zero?
    jne enter_region     | if it was non zero, lock was set, so loop
    ret                  | return to caller; critical region entered

leave_region:
    move lock,#0         | store a 0 in lock
    ret                  | return to caller
```

Figure 2-10. Setting and clearing locks using TSL.

Para que un proceso pueda entrar a su región crítica, deberá hacer un llamado a la rutina *enter_region*, y para liberar su región crítica, deberá hacer un llamado a la rutina *leave_region*.

2.3.4 Dormir y despertar.

La solución de Peterson y la que usa TSL son correctas, pero ambas tienen el defecto de requerir espera activa.

Este defecto no sólo desperdicia ciclos de CPU, sino que pueden darse casos de inanición. Considere lo siguiente:

Se tienen dos procesos, uno de alta prioridad (H) y otro de baja prioridad (L), la regla es que H se ejecuta siempre que esté listo (aunque L también). Suponga que L se está ejecutando (mientras H está bloqueado por ejemplo) y entra a su región crítica. Después H queda listo para ejecutarse e inicia su espera activa.

¿Cuál es el problema en la situación anterior?

Existen primitivas de IPC que hacen que los procesos se bloqueen en lugar de gastar innecesariamente tiempo de CPU.

Estas primitivas son SLEEP y WAKEUP.

Como ejemplo de uso de las primitivas, considere:

El problema del productor consumidor (buffer limitado).

Dos procesos comparten un mismo buffer de tamaño fijo. El productor coloca información en el buffer y el consumidor la saca.

¿Qué problemas pueden surgir?

```
#define N 100                /* number of slots in the buffer */
int count = 0;               /* number of items in the buffer */

void producer(void)
{
    while (TRUE) {           /* repeat forever */
        produce_item();      /* generate next item */
        if (count == N) sleep(); /* if buffer is full, go to sleep */
        enter_item();        /* put item in buffer */
        count = count + 1;    /* increment count of items in buffer */
        if (count == 1) wakeup(consumer); /* was buffer empty? */
    }
}

void consumer(void)
{
    while (TRUE) {           /* repeat forever */
        if (count == 0) sleep(); /* if buffer is empty, got to sleep */
        remove_item();        /* take item out of buffer */
        count = count - 1;    /* decrement count of items in buffer */
        if (count == N-1) wakeup(producer); /* was buffer full? */
        consume_item();       /* print item */
    }
}
```

Figure 2-11. The producer-consumer problem with a fatal race condition.

Este esquema ya no hace espera activa pero incurre en condiciones de competencia (¿Por qué?).

2.3.5 Semáforos.

En 1965 E.W. Dijkstra sugirió usar una variable entera para contar el número de señales de despertar guardadas para uso futuro llamada **semáforo**.

Un semáforo puede ser 0, indicando que no hay señales de despertar guardadas o un valor positivo si había señales pendientes.

Dijkstra propuso dos operaciones: DOWN y UP.

DOWN verifica si el valor del semáforo es mayor que 0, y si lo es decrementa el valor y continua, si es 0 el proceso se pone a dormir sin completar la operación DOWN por el momento.

UP incrementa el valor del semáforo direccionado y se despierta a alguno de los procesos (si los hubiera) dormidos.

Todo esto se realiza como una operación atómica, esto es, se garantiza que una vez que una operación de semáforo se ha iniciado, ningún otro proceso podrá acceder al semáforo hasta que la operación se haya completado o bloqueado.

La solución al productor consumidor con semáforos es:

```
#define N 100                /* number of slots in the buffer */
typedef int semaphore;       /* semaphores are a special kind of int */
semaphore mutex = 1;        /* controls access to critical region */
semaphore empty = N;        /* counts empty buffer slots */
semaphore full = 0;         /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {          /* TRUE is the constant 1 */
        produce_item(&item); /* generate something to put in buffer */
        down(&empty);        /* decrement empty count */
        down(&mutex);        /* enter critical region */
        enter_item(item);    /* put new item in buffer */
        up(&mutex);          /* leave critical region */
        up(&full);           /* increment count of full slots */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {          /* infinite loop */
        down(&full);         /* decrement full count */
        down(&mutex);        /* enter critical region */
        remove_item(&item); /* take item from buffer */
        up(&mutex);          /* leave critical region */
        up(&empty);          /* increment count of empty slots */
        consume_item(item); /* do something with the item */
    }
}
```

Figure 2-12. The producer-consumer problem using semaphores.

2.3.6 Monitores.

La solución con semáforos también tiene sus bemoles.

¿Qué pasa si se invierte el orden de los DOWN del productor en la solución anterior al problema del productor consumidor?

La respuesta es que existe un bloqueo mutuo (¿Por qué?).

Hoare y Brinch Hansen propusieron casi simultáneamente, una primitiva de sincronización de más alto nivel llamada monitor.

Un **monitor** es una colección especial de procedimientos, variables y estructuras de datos que se agrupan en un tipo especial de módulo o paquete.

Los procesos pueden invocar los procedimientos de un monitor en el momento que deseen, pero no pueden acceder directamente a las estructuras de datos internas del monitor desde procedimientos declarados fuera del monitor.

Los monitores poseen una propiedad especial que los hace útiles para lograr la exclusión mutua: sólo un proceso puede estar activo en el monitor en un momento dado.

Los monitores son una construcción del lenguaje de programación, por lo que el compilador sabe que hay que manipular a los procedimientos de manera diferente.

```
monitor example
  integer i;
  condition c;

  procedure producer(x);
  :
  end;

  procedure consumer(x);
  :
  end;
end monitor;
```

Figure 2-13. A monitor.

El compilador es el encargado de implementar la exclusión mutua.

2.3.7 Transferencia de mensajes.

Este método de IPC utiliza dos primitivas: SEND y RECEIVE.

- send(destino, &mensaje)
- receive(origen, &mensaje)

La transferencia de mensajes también tiene sus problemas y se deben considerar (¿Podría identificar algunos?)

Revise la propuesta de solución al problema del productor consumidor con transferencia de mensajes.

2.4 Problemas clásicos de IPC.

2.4.1 El problema de la cena de los filósofos.

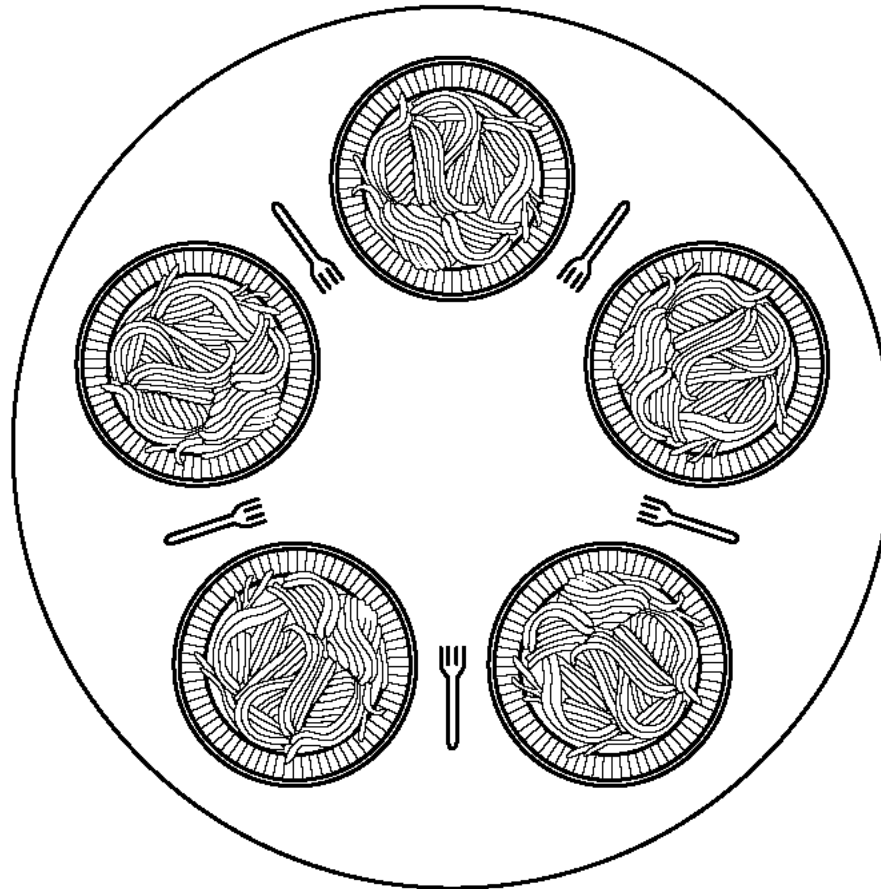


Figure 2-16. Lunch time in the Philosophy Department.

```
#define N 5                                /* number of philosophers */

void philosopher(int i)                    /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think();                          /* philosopher is thinking */
        take_fork(i);                     /* take left fork */
        take_fork((i+1) % N);             /* take right fork; % is modulo operator */
        eat();                            /* yum-yum, spaghetti */
        put_fork(i);                      /* put left fork back on the table */
        put_fork((i+1) % N);             /* put right fork back on the table */
    }
}
```

Figure 2-17. A nonsolution to the dining philosophers problem.

La solución anterior no funciona (¿Por qué?).

Revisar la solución al problema de la cena de los filósofos del libro de Tanenbaum.

El problema de la cena de los filósofos es útil para modelar procesos que compiten por tener acceso exclusivo a un número limitado de recursos.

2.4.2 El problema de los lectores y escritores.

Este problema modela el acceso a una base de datos.

Es aceptable tener múltiples procesos leyendo la base de datos al mismo tiempo, pero si un proceso está actualizando la base de datos, ningún otro proceso podrá tener acceso a ella.

```
typedef int semaphore;      /* use your imagination */
semaphore mutex = 1;        /* controls access to 'rc' */
semaphore db = 1;           /* controls access to the data base */
int rc = 0;                 /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {          /* repeat forever */
        down(&mutex);        /* get exclusive access to 'rc' */
        rc = rc + 1;         /* one reader more now */
        if (rc == 1) down(&db); /* if this is the first reader ... */
        up(&mutex);          /* release exclusive access to 'rc' */
        read_data_base();    /* access the data */
        down(&mutex);        /* get exclusive access to 'rc' */
        rc = rc - 1;         /* one reader fewer now */
        if (rc == 0) up(&db); /* if this is the last reader ... */
        up(&mutex);          /* release exclusive access to 'rc' */
        use_data_read();     /* noncritical region */
    }
}

void writer(void)
{
    while (TRUE) {          /* repeat forever */
        think_up_data();     /* noncritical region */
        down(&db);           /* get exclusive access */
        write_data_base();   /* update the data */
        up(&db);             /* release exclusive access */
    }
}
```

Figure 2-19. A solution to the readers and writers problem.

¿Qué pasa con un proceso escritor si los lectores siguen llegando?

2.4.3 El problema del peluquero dormilón.

La idea es programar al peluquero y a sus clientes sin entrar en condiciones de competencia.

Considere lo que pasa en una peluquería.

Revise la solución propuesta por Tanenbaum para el problema del peluquero dormido.

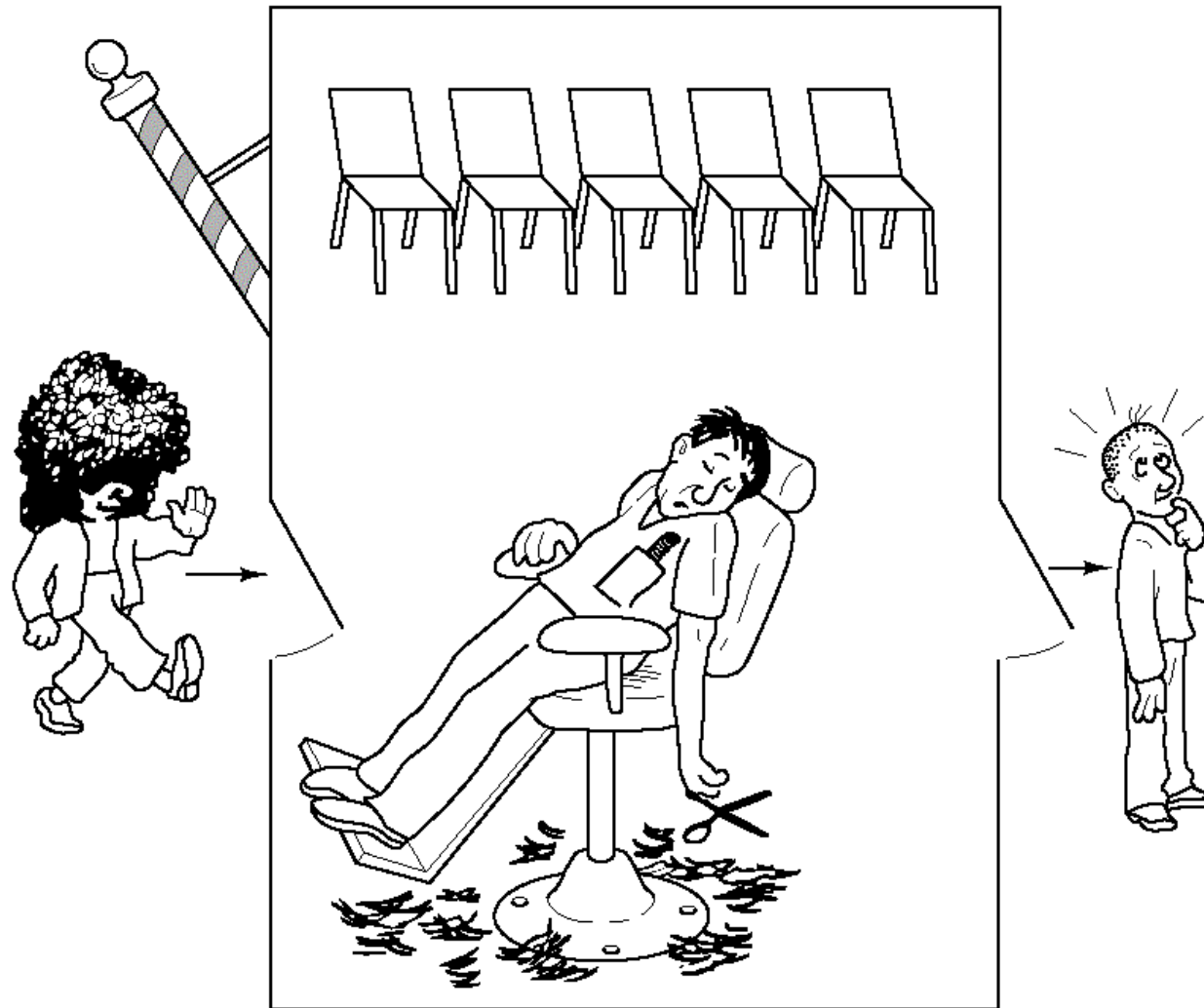


Figure 2-20. The sleeping barber.

No todos los problemas de IPC requieren transferencia de datos, basta que el problema involucre sincronización de procesos para que se consideren problemas de IPC.

2.5 Planificación de procesos.

Cuando hay más de un proceso listo para ejecutarse, la parte del SO que decide cuál se ejecutará primero se llama **planificador** (*scheduler*).

Algunos criterios para determinar un buen algoritmo de planificación son:

1. Equitatividad: asegurarse de que cada proceso reciba una parte justa del tiempo de CPU.
2. Eficiencia: mantener a la CPU ocupada el mayor tiempo posible.
3. Tiempo de respuesta: minimizar el tiempo de respuesta para usuarios interactivos.
4. Volumen de producción: maximizar el número de trabajos procesados por hora.

Es difícil cumplir o conciliar completamente estos objetivos.

Cualquier algoritmo de planificación que dé preferencia a una clase de trabajos, perjudicará a los de otras clases.

Para asegurarse de que ningún proceso se ejecute durante demasiado tiempo, casi todas las computadoras tienen incorporado un reloj que genera interrupciones periódicamente.

En cada interrupción de reloj, el SO se ejecuta y decide si debe permitirse que el proceso que se está ejecutando actualmente continúe o si ya disfrutó de suficiente tiempo de CPU por el momento y debe suspenderse para darle oportunidad a otro proceso de usar la CPU.

Existen dos tipos de planificación:

1. Planificación expropiativa: es la estrategia de permitir que procesos lógicamente ejecutables se suspendan temporalmente.
2. Planificación no expropiativa: aquí los procesos no pueden ser interrumpidos sino hasta que terminan.

2.5.1 Planificación round robin.

En este esquema, a cada proceso se le asigna un intervalo de tiempo llamado **cuanto** durante el que se le permite ejecutarse.

Este algoritmo es uno de los más sencillos, equitativos y ampliamente utilizados.



Figure 2-22. Round robin scheduling. (a) The list of runnable processes. (b) The list of runnable processes after *B* uses up its quantum.

Escoger un cuanto corto causa demasiadas conmutaciones de procesos y reduce la eficiencia de la CPU, pero escogerlo

demasiado largo puede dar pie a una respuesta deficiente a solicitudes interactivas cortas.

2.5.2 Planificación por prioridad.

La idea básica es sencilla: a cada proceso se le asigna una prioridad y se ejecutará primero el proceso con la prioridad más alta.

A fin de evitar que los procesos de alta prioridad se ejecuten indefinidamente, el planificador puede reducir la prioridad de los procesos que actualmente se ejecutan en cada interrupción.

En ocasiones conviene combinar la idea de los procesos con prioridad y la de round robin.

Las prioridades más altas se ejecutan primero.

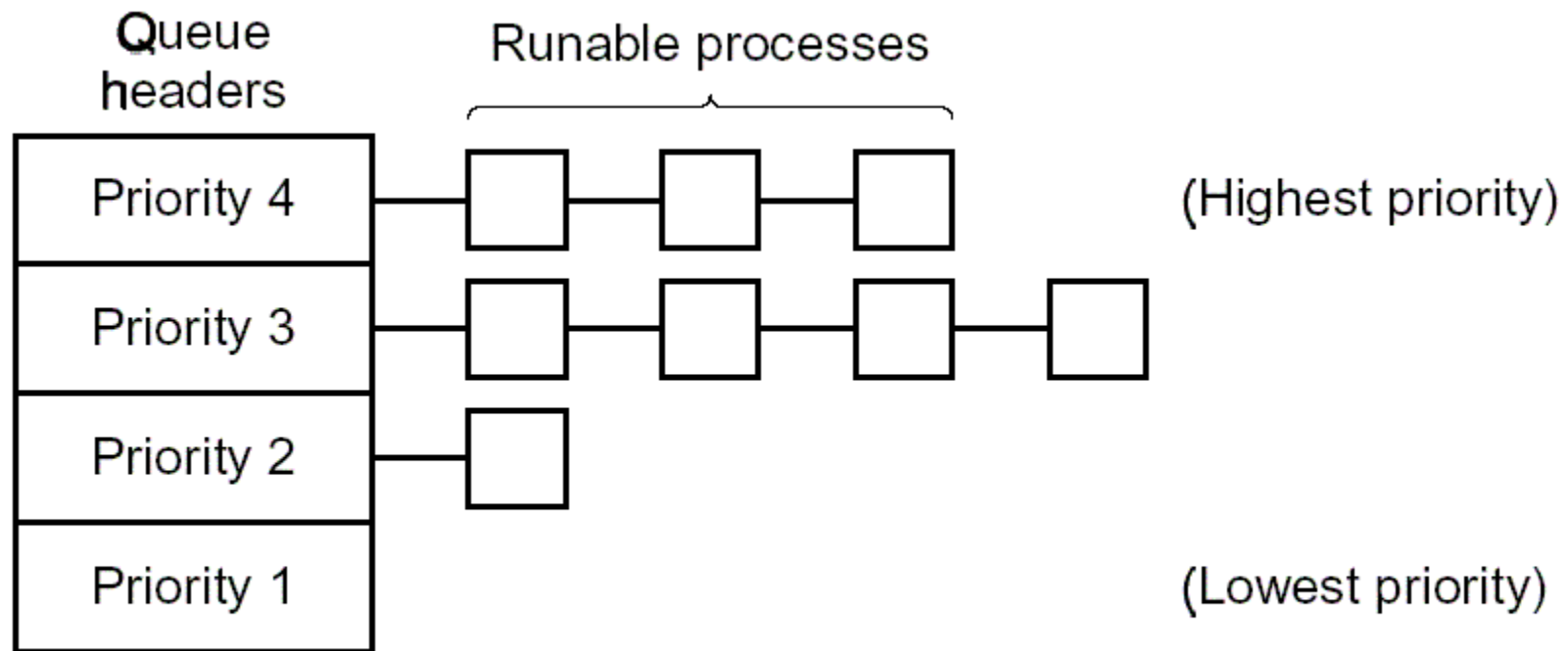


Figure 2-23. A scheduling algorithm with four priority classes.

Sólo se desciende de prioridad cuando ya no hay procesos por ejecutar en la prioridad actual.

Si las prioridades no se ajustan, los procesos de baja prioridad podrían morir de inanición.

2.5.3 El trabajo más corto.

Este algoritmo resulta especialmente apropiado para los trabajos por lotes disponibles simultáneamente y cuyos tiempos de ejecución se conocen de antemano. Esto es posible predecir, en donde se efectúan trabajos parecidos todos los días.

La idea fundamental consiste en calcular el tiempo de retorno de cada proceso. La mejora se verá reflejada en el tiempo de espera promedio de cada proceso.



Figure 2-24. An example of shortest job first scheduling.

También existen:

- Planificación garantizada
- Planificación por lotería
- Planificación en tiempo real
- planificación de dos niveles, etc.