

Concurrencia

Concurrencia: exclusión mutua y sincronización

Concurrencia

El diseño de software está relacionado con la gestión de procesos e hilos

- Multiprogramación
- Multiprocesamiento
- Procesamiento distribuido

En todas estas áreas es fundamental la **concurrencia**

Concurrencia

Se presenta en contextos diferentes

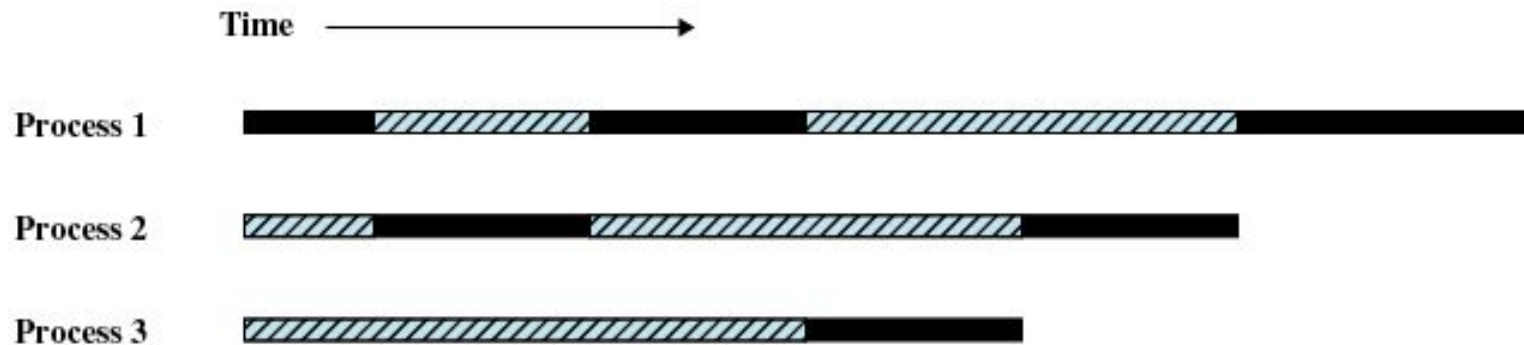
- Múltiples aplicaciones
 - Multiprogramación
- Aplicaciones estructuradas
 - Una aplicación puede ser un conjunto de procesos concurrentes
- Estructura del sistema operativo
 - El sistema operativo es un conjunto de procesos o hilos

Concurrencia

Tabla 5.1. Algunos términos clave relacionados con la concurrencia

sección crítica (<i>critical section</i>)	Sección de código dentro de un proceso que requiere acceso a recursos compartidos y que no puede ser ejecutada mientras otro proceso esté en una sección de código correspondiente.
interbloqueo (<i>deadlock</i>)	Situación en la cual dos o más procesos son incapaces de actuar porque cada uno está esperando que alguno de los otros haga algo.
círculo vicioso (<i>livelock</i>)	Situación en la cual dos o más procesos cambian continuamente su estado en respuesta a cambios en los otros procesos, sin realizar ningún trabajo útil.
exclusión mutua (<i>mutual exclusion</i>)	Requisito de que cuando un proceso esté en una sección crítica que accede a recursos compartidos, ningún otro proceso pueda estar en una sección crítica que acceda a ninguno de esos recursos compartidos.
condición de carrera (<i>race condition</i>)	Situación en la cual múltiples hilos o procesos leen y escriben un dato compartido y el resultado final depende de la coordinación relativa de sus ejecuciones.
inanición (<i>starvation</i>)	Situación en la cual un proceso preparado para avanzar es soslayado indefinidamente por el planificador; aunque es capaz de avanzar, nunca se le escoge.

Multiprogramación y multiprocesamiento



(a) Interleaving (multiprogramming, one processor)



(b) Interleaving and overlapping (multiprocessing; two processors)


 Blocked  Running

Figure 2.12 Multiprogramming and Multiprocessing

Dificultades de la concurrencia

- Compartir recursos globales
- Para el sistema operativo es complicado gestionar la asignación de recursos de manera óptima
- Llega a ser muy complicado localizar errores de programación

Un ejemplo sencillo

```
void echo()  
{  
    cent = getchar();  
    csal = cent;  
    putchar(csal);  
}
```


Un ejemplo sencillo

Proceso P1

```
.  
cent = getchar() ;  
.   
csal = cent ;  
putchar(csal) ;  
.   
.
```

Proceso P2

```
.  
.   
cent = getchar() ;  
csal = cent ;  
.   
putchar(csal) ;  
.
```

Quantum (t)	Proceso P1	Proceso P2
	.	.

Un ejemplo sencillo

Quantum (t)	Proceso P1	Proceso P2
1	.	.
2	<code>cent = getchar() ;</code>	.
3	.	<code>cent = getchar() ;</code>
4	<code>csal = cent ;</code>	<code>csal = cent ;</code>
5	<code>putchar(csal) ;</code>	.
6	.	<code>putchar(csal) ;</code>
7	.	.

Preocupaciones del sistema operativo

- Debe ser capaz de seguir la pista de varios procesos
- Debe ubicar y desubicar varios recursos para cada proceso activo
 - Tiempo de procesador
 - Memoria
 - Archivos
 - Dispositivos de E/S
- Debe proteger los datos y recursos
- **El funcionamiento de un proceso y el resultado que produzca debe ser independiente de la velocidad a la que suceda su ejecución en relación con la velocidad de otros procesos concurrentes.**

Interacción de procesos

- Procesos que no se perciben entre sí
- Procesos que se perciben indirectamente entre sí
- Procesos que se perciben directamente entre sí

Interacción de procesos

Grado de percepción	Relación	Influencia que un proceso tiene sobre el otro	Potenciales problemas de control
Procesos que no se perciben entre sí	Competencia	<ul style="list-style-type: none">• Los resultados de un proceso son independientes de la acción de los otros La temporización del proceso puede verse afectada	<ul style="list-style-type: none">• Exclusión mutua• Interbloqueo (recurso renovable)• Inanición
Procesos que se perciben indirectamente entre sí (por ejemplo, objeto compartido)	Cooperación por compartición	<ul style="list-style-type: none">• Los resultados de un proceso pueden depender de la información obtenida de otros• La temporización del proceso puede verse afectada	<ul style="list-style-type: none">• Exclusión mutua• Interbloqueo (recurso renovable)• Inanición• Coherencia de datos
Procesos que se perciben directamente entre sí (tienen primitivas de comunicación a su disposición)	Cooperación por comunicación	<ul style="list-style-type: none">• Los resultados de un proceso pueden depender de la información obtenida de otros• La temporización del proceso puede verse afectada	<ul style="list-style-type: none">• Interbloqueo (recurso consumible)• Inanición

Requisitos para la exclusión mutua

- Sólo un proceso de entre todos los que poseen secciones críticas por el mismo recurso u objeto compartido, debe tener permiso para entrar en ella en un instante dado (exclusión mutua)
- Un proceso que se interrumpe en su sección no crítica debe hacerlo sin interferir con otros procesos
- Un proceso no puede ser demorado indefinidamente, ni interbloqueo ni inanición
- Si ningún proceso está en su sección crítica, cualquier proceso que solicite entrar en la suya debe poder entrar sin demora
- No se deben hacer suposiciones sobre las velocidades relativas de los procesos ni sobre el número de procesadores
- Un proceso permanece dentro de su sección crítica sólo por un tiempo finito

Exclusión mutua: soporte hardware

- Deshabilitar interrupciones
 - Un proceso continuará ejecutando hasta que invoque un servicio del sistema operativo o hasta que sea interrumpido
 - Deshabilitar interrupciones garantiza la exclusión mutua
 - Se limita la capacidad del procesador de entrelazar programas
 - Multiprocesador
 - Deshabilitar interrupciones no garantiza la exclusión mutua

```
while (true)
{
    /* inhabilitar interrupciones */ ;

    /* sección crítica */;

    /* habilitar interrupciones */ ;

    /* resto */;
}
```


Exclusión mutua: soporte hardware

- Instrucciones máquina especiales
 - Llevan a cabo acciones sobre una única posición de memoria con un único ciclo de búsqueda de instrucción
 - El acceso a la posición de memoria se le bloquea a toda otra instrucción

Exclusión mutua: soporte hardware

- Instrucción *Test and Set* (comprueba y establece)

```
boolean testset (int i)
{
    if (i == 0)
    {
        i = 1;
        return true;
    }
    else
    {
        return false;
    }
}
```

Exclusión mutua: soporte hardware

- Instrucción *Exchange*

```
void exchange(int registro,int memoria)
{
    int temp;
    temp = memoria;
    memoria = registro;
    registro = temp;
}
```

Exclusión mutua

```
/* programa exclusión mutua */
const int n = /* número de procesos */;
int cerrojo;

void P(int i) {
    while (true) {
        while (!testset (cerrojo),
            /* no hacer nada */;
        /* sección crítica */;
        cerrojo = 0;
        /* resto */ }

void main()
{
    cerrojo = 0;
    paralelos (P(1), P(2), . . . ,P(n));
}
```

Pedir y eventualmente esperar

Salida de Sección crítica

Exclusión mutua

```
/* programa exclusión mutua */
const int n = /* número de procesos */;
int cerrojo;
void P(int i)
{
    int llavei = 1;
    while (true)
    {
        do exchange (llavei, cerrojo)
            while (llavei != 0);
        /* sección crítica */;
        exchange (llavei, cerrojo);
        /* resto */    }
    }
void main()
{
    cerrojo = 0;
    paralelos (P(1), P(2), . . ., P(n));
}
```

Entrada a Sección crítica

Salida de Sección crítica

Exclusión mutua: soporte hardware

- Instrucción *Compare_and_swap*

```
int compare_and_swap (int *word, int
    testval, int newval)
{
    int oldval;
    oldval = *word;
    if (oldval == testval) *word = newval;
    return oldval;
}
```

```
/* programa exclusión mutua */
const int n = /* número de procesos */;
int cerrojo;
void P(int i)
{
    while (true)
    {
        while (comp&swap (cerrojo, 1) == 1)
            /* no hacer nada */;
        /* sección crítica */;
        cerrojo = 0;
        /* resto */
    }
}

void main()
{
    cerrojo = 0;
    paralelos (P(1), P(2), ..., P(n));
}
```

Pedir y eventualmente esperar

Salida de Sección crítica

Instrucciones máquina de exclusión mutua

- **Ventajas**

- Es aplicable a cualquier número de procesos sobre un procesador único o multiprocesador de memoria principal compartida
- Es simple y, por tanto, fácil de verificar
- Puede ser utilizado para dar soporte a múltiples secciones críticas

Instrucciones máquina de exclusión mutua

- Desventajas

- La espera activa consume tiempo procesador
- La inanición es posible cuando un proceso abandona su sección crítica y hay más de un proceso esperando
- Interbloqueo
 - Si un proceso de baja prioridad tiene la sección crítica y un proceso de mayor prioridad la necesita, el proceso de más alta prioridad conseguirá el procesador para esperar a la sección crítica.

Semáforos

- La variable especial denominada semáforo se utiliza para señalar
- Si un proceso está esperando una señal, este se bloquea hasta que la señal haya sido enviada

Semáforos

- Un semáforo es una variable que tiene un valor entero con solamente 3 operaciones definidas
 - Puede ser **inicializado** a un valor no negativo
 - La operación ***semWait*** decrementa el valor del semáforo
 - La operación ***semSignal*** incrementa el valor del semáforo

Primitivas del semáforo

```
struct semaphore {  
    int count;  
    queueType queue; };
```

```
void semWait(semaphore s)  
{  
    s.count--;  
    if (s.count < 0) {  
        /* place this process in s.queue */;  
        /* block this process */;    }    }
```

```
void semSignal(semaphore s)  
{  
    s.count++;  
    if (s.count <= 0) {  
        /* remove a process P from s.queue */;  
        /* place process P on ready list */;    }    }
```

Primitivas del semáforo

```
struct semaphore {
    int count;
    queueType queue;
};
void semWait(semaphore s)
{
    s.count--;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process */;
    }
}
void semSignal(semaphore s)
{
    s.count++;
    if (s.count <= 0) {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}
```

Figure 5.3 A Definition of Semaphore Primitives

Primitivas del semáforo binario

```
struct binary_semaphore {  
    enum {zero, one} value;  
    queueType queue;    };
```

```
void semWaitB(binary_semaphore s)  
{  
    if (s.value == one)  
        s.value = zero;  
    else { /* place this process in s.queue */;  
        /* block this process */;    } }
```

```
void semSignalB(semaphore s)  
{  
    if (s.queue is empty())  
        s.value = one;  
    else { /* remove a process P from s.queue */;  
        /* place process P on ready list */;    } }
```

Primitivas del semáforo binario

```
struct binary_semaphore {
    enum {zero, one} value;
    queueType queue;
};
void semWaitB(binary_semaphore s)
{
    if (s.value == one)
        s.value = zero;
    else {
        /* place this process in s.queue */;
        /* block this process */;
    }
}
void semSignalB(semaphore s)
{
    if (s.queue is empty())
        s.value = one;
    else {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}
```

Figure 5.4 A Definition of Binary Semaphore Primitives

Soluciones a la exclusión mutua con semáforos

```
/* programa exclusión mutua */
const int n = /* numero de procesos */;
semaphore s = 1 ;
void P(int i)
{
    while (true)
    {
        semWait(s);
        /* sección crítica */;

        semSignal(s);

        /* resto */;
    }
}
void main()
{
    parbegin (P(1), P(2), . . . , P(n));
}
```

Soluciones a la exclusión mutua con semáforos

```
/* programa exclusión mutua */
const int n = /* numero de procesos */;
semaphore s = 1 ;
void P(int i)
{
    while (true) {
        entrar sección crítica;
        semWait(s);
        /* sección crítica */;
        semSignal(s);
        salir sección crítica;

        /* resto */;    }
}
void main()
{
    parbegin (P(1), P(2), . . ., P(n));
}
```

El problema productor/consumidor

- Uno o más procesos están generando algún tipo de datos y poniéndolos en un *buffer*
- Un único consumidor está extrayendo datos de dicho *buffer* en un momento dado
- Sólo un agente productor o consumidor puede acceder al *buffer* en un momento dado

El problema productor/consumidor

productor	consumidor
<pre>while (true) { /* producir dato v */; b[entra] = v; entra++; }</pre>	<pre>while (true) { while (entra <= sale) /* no hacer nada */; w = b[sale]; sale++; /* consumir dato w */ }</pre>

El problema productor/consumidor

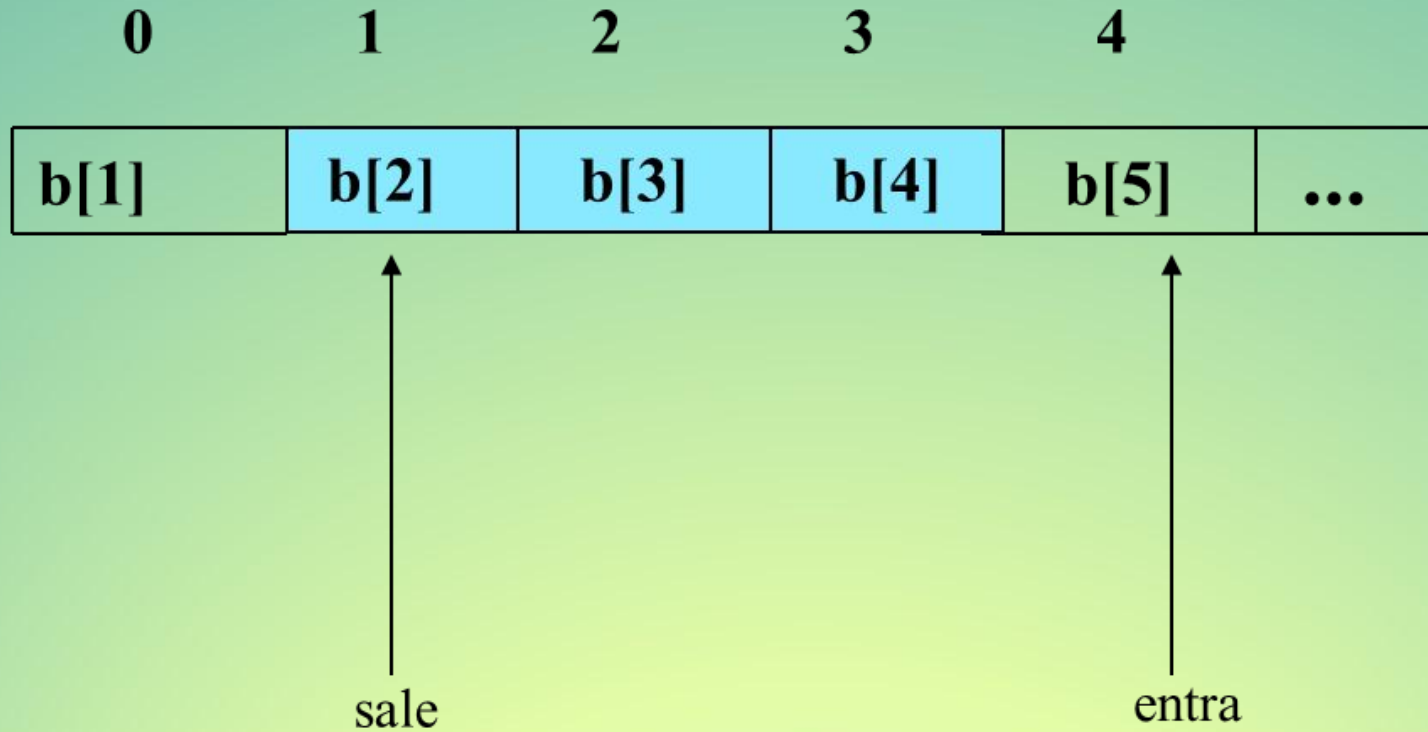


Figura 5.8. *Buffer* infinito para el problema productor/consumidor

Nota: el área sombreada indica la porción del *buffer* que está ocupada

Solución incorrecta

(buffer infinito sem binarios)

```
/* programa productor consumidor */
int n;
binary_semaphore s = 1;
binary_semaphore retardo = 0;

void productor() {
    while (true) {
        producir();
        semWaitB(s);
        añadir();
        n++;
        if (n==1) semSignalB(retardo);
        semSignalB(s);
    }
}
```

Sigue →

```
void consumidor() {
    semWaitB(retardo);
    while (true) {
        semWaitB(s);
        extraer();
        n--;
        semSignalB(s);
        consumir();
        if (n==0) semWaitB(retardo);
    }
}

void main() {
    n = 0;
    paralelos
    (productor, consumidor);
}
```

Solución correcta (buffer infinito sem binarios)

```
/* programa productor consumidor */
int n;
binary_semaphore s = 1;
binary_semaphore retardo = 0;

void productor() {
    while (true) {
        producir();
        semWaitB(s);
        añadir();
        n++;
        if (n==1) semSignalB(retardo);
        semSignalB(s);
    }
}
```

Sigue →

```
void consumidor() {
    int m; /* una variable local */
    semWaitB(retardo);
    while (true) {
        semWaitB(s);
        extraer();
        n--;
        m = n;
        semSignalB(s);
        consumir();
        if (m==0) semWaitB(retardo);
    }
}

void main() {
    n = 0;
    paralelos
    (productor, consumidor);
}
```

Solución correcta (buffer infinito sem generales)

```
/* programa productor
consumidor */
semaphore n = 0;
semaphore s = 1;
void productor()
{
    while (true) {
        producir();
        semWait(s);
        añadir();
        semSignal(s);
        semSignal(n);
    }
}
```

Sigue →

```
void consumidor()
{
    while (true) {
        semWait(n);
        semWait(s);
        extraer();
        semSignal(s);
        consumir();
    }
}

void main()
{
    paralelos (productor, consumidor);
}
```

El problema productor/consumidor

productor	consumidor
<pre>while (true) { /* producir dato v */ while ((entra + 1) % n == sale) /* no hacer nada */; b[entra] = v; entra = (entra + 1) % n }</pre>	<pre>while (true) { while (entra <= sale) /* no hacer nada */; w = b[sale]; sale++; /* consumir dato w */ }</pre>

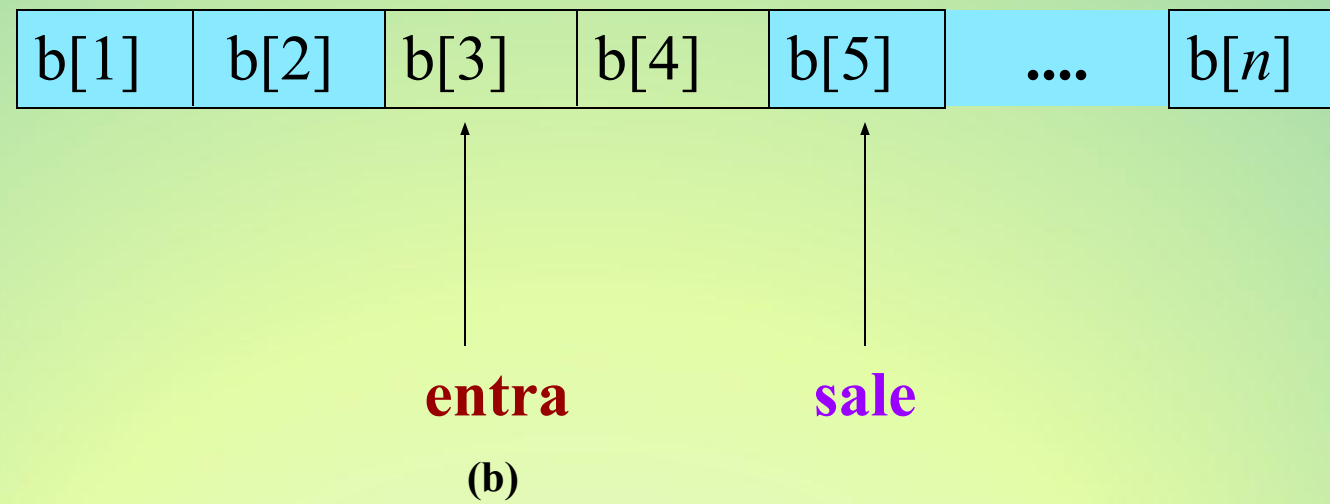
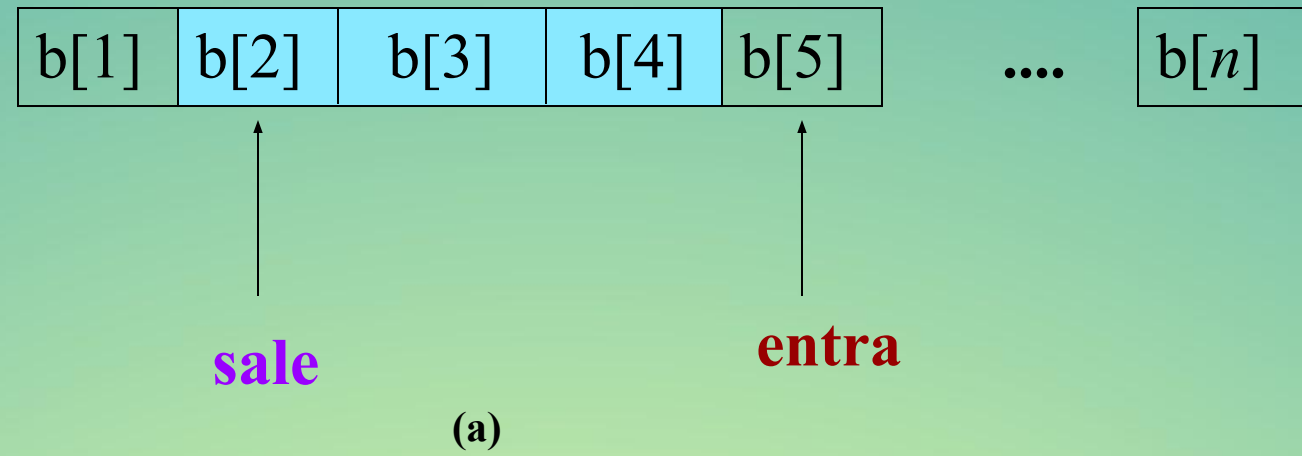


Figura 5.12. *Buffer* finito circular para el problema productor/consumidor

```

/* program productorconsumidor */
semaphore n = 0;
semaphore s = 1;
semaphore e = tamaño_buffer;

void productor()
{
    while (true)
    {
        produce();
        wait(e);
        wait(s);
        agrega();
        signal(s);
        signal(n);
    }
}

```

```

void consumidor()
{
    while (true)
    {
        wait(n);
        wait(s);
        toma();
        signal(s);
        signal(e);
        consume();
    }
}

void main()
{
    parbegin (productor,
consumidor);
}

```

Figura 5.13. Una solución al problema productor/consumidor con *buffer* acotado usando semáforos

Monitores

- Un monitor es un módulo software
- Características principales
 - Las variables locales de datos son sólo accesibles por el monitor
 - Un proceso entra en el monitor invocando uno de sus procedimientos
 - Sólo un proceso puede estar ejecutando dentro del monitor al tiempo

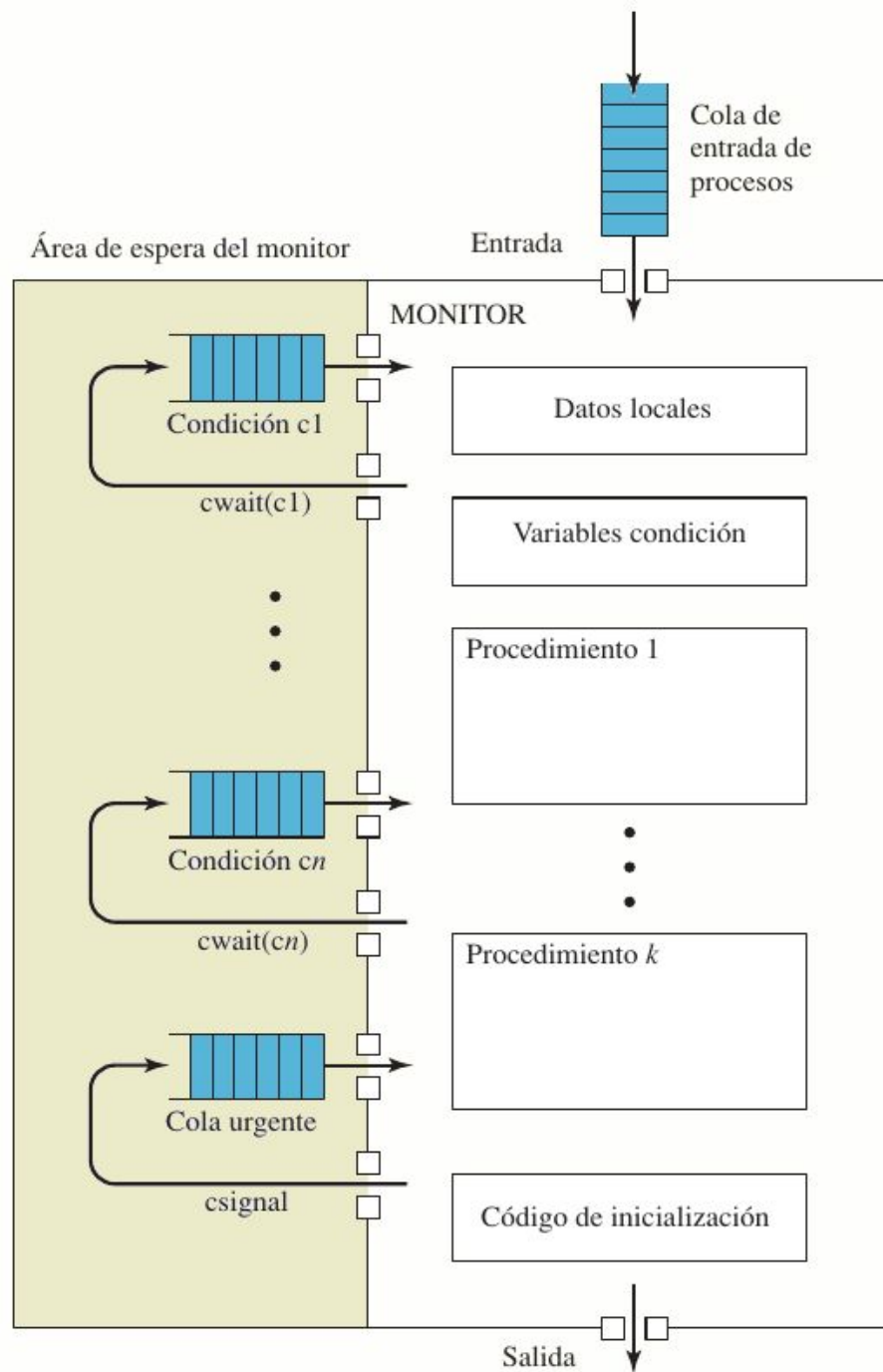


Figura 5.15. Estructura de un Monitor

Una solución al problema productor/consumidor con *buffer* acotado usando un monitor

```
/* program productorconsumidor */
monitor bufferacotado;
char buffer[N];      /* espacio para N items */
int nextin, nextout;  /* punteros buffer */
int cont;             /* número de items en buffer */
cond nolleno, novacío; /* para sincronización*/
void agrega (char x)
{
    if (cont == N)
        cwait(nolleno); /* buffer lleno; espere */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    cont++; /* un item más en buffer */
    csignal(novacío); /* reanudar consumidor que
espera */
}
```

Sigue → .

```
void extrae (char x)
{
    if (cont == 0)
        cwait(novacío); /* buffer vacío; espere */
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    cont--; /* un item menos en buffer */
    csignal(nolleno); /* reanudar productor que
espera */
}

/* cuerpo monitor */
nextin = 0; nextout = 0; cont = 0; /* buffer
inicialmente vacío */
}
```

Una solución al problema productor/consumidor con *buffer* acotado usando un monitor

```
void productor()  
{  char x;  
  while (true)  
  {    produce(x);  
      agrega(x);  }  
}  
  
void consumidor()  
{  
  char x;  
  while (true)  
  {    extrae(x);  
      consume(x);  }  
}  
  
void main()  
{  
  parbegin (productor, consumidor);  
}
```

```

void agrega (char x)
{
    while (cuenta == N)
        cwait(nolleno);          /* buffer lleno, evitar desbordar */
    buffer[dentro] = x;
    dentro = (dentro + 1) % N;
    cuenta++;                     /* un dato más en el buffer */
    cnotify(novacio);             /* notifica a algún consumidor en espera */
}

void extrae (char x)
{
    while (cuenta == 0)
        cwait(novacio);          /* buffer vacío, evitar consumo */
    x = buffer[fuera];
    fuera = (fuera + 1) % N;
    cuenta--;                     /* un dato menos en el buffer */
    cnotify(nolleno);             /* notifica a algún productor en espera */
}}

```

Figura 5.17. Monitor de *buffer* acotado con el monitor Mesa

Paso de mensajes

- Impone la exclusión mutua
- Intercambia información

send (destino, mensaje)

receive (origen, mensaje)

Sincronización

- Emisor y receptor pueden o no bloquearse (esperando al mensaje)
- Envío bloqueante, recepción bloqueante
 - Ambos emisor y receptor se bloquean hasta que el mensaje se entrega
 - Se le conoce como *rendezvous*

Sincronización

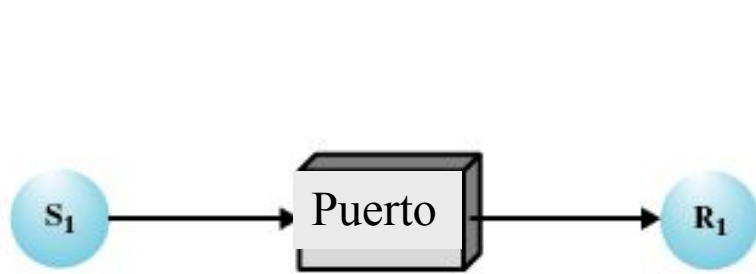
- Envío no bloqueante, recepción bloqueante
 - El emisor puede continuar
 - El receptor se bloquea hasta que el mensaje solicitado llegue
- Envío no bloqueante, recepción no bloqueante
 - Ninguna de las partes tiene que esperar

Direccionamiento

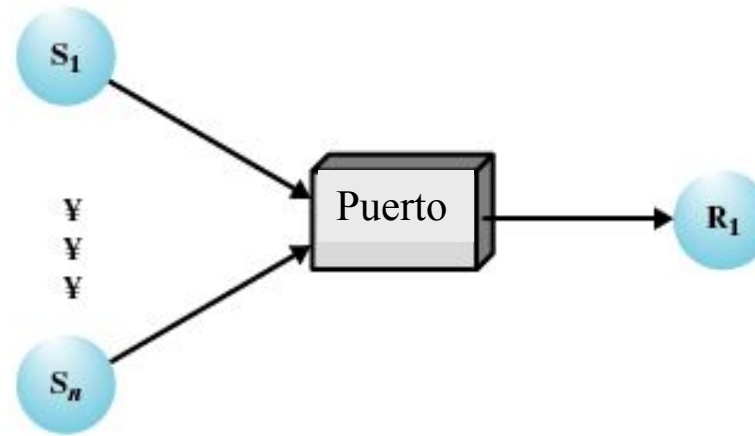
- Direccionamiento directo
 - La primitiva *send* incluye un identificador específico del proceso destinatario
 - La primitiva *receive* debe conocer con anticipación de qué proceso espera el mensaje
 - La primitiva *receive* puede usar el parámetro *origen* para devolver un valor cuando la operación de recepción se completa

Direccionamiento

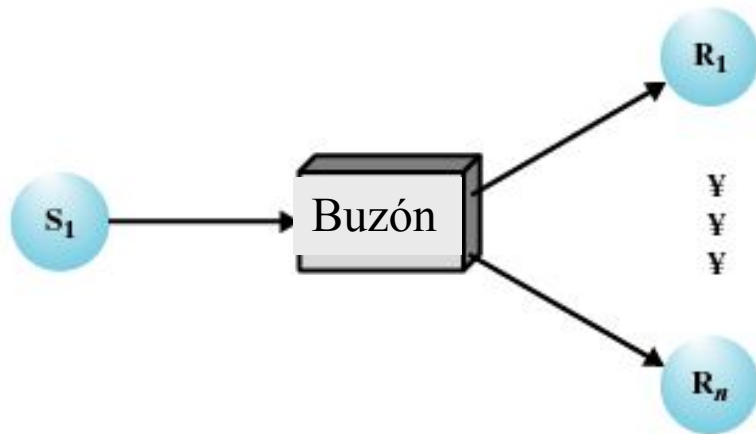
- Direccionamiento indirecto
 - Los mensajes se envían a una estructura de datos compartida, que consiste en colas
 - Las colas se conocen como buzones (*mailboxes*)
 - Un proceso envía un mensaje al buzón apropiado y otro proceso toma el mensaje del buzón



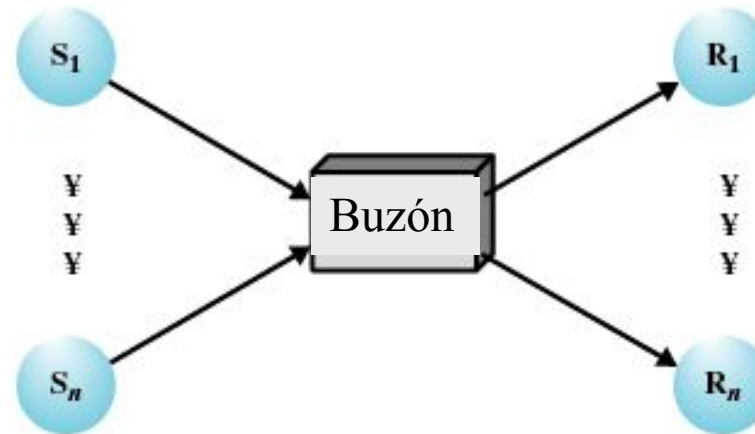
(a) One to one



(b) Many to one



(c) One to many



(d) Many to many

Figure 5.18 Indirect Process Communication

Formato de mensaje

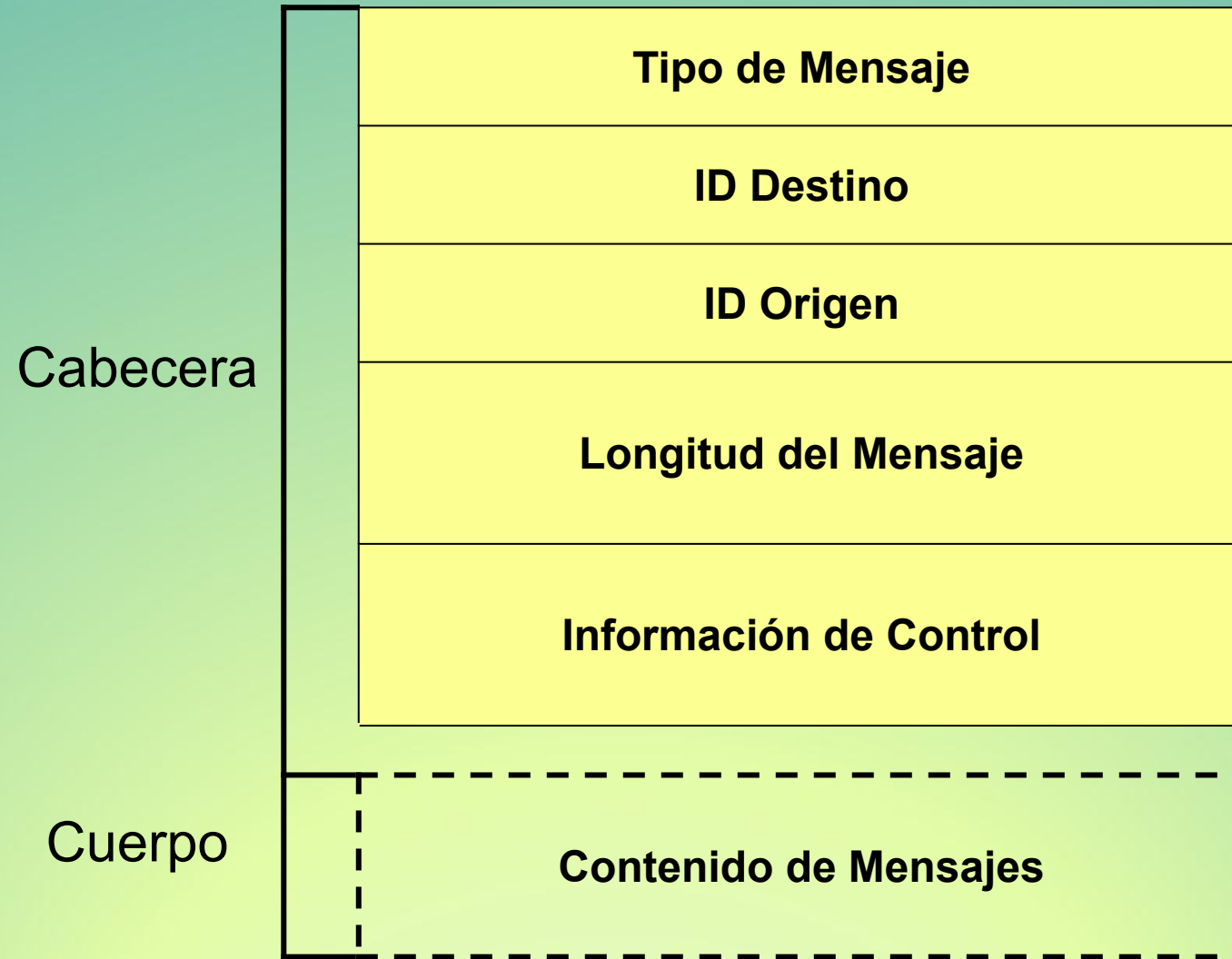


Figura 5.19. Formato general de mensaje

```

/* programa exclusión mutua */
const int n = /* número de procesos */;
void P(int i)
{
    message carta;
    while (true)
    {
        receive (buzon, carta);
        /* sección crítica */;
        send (buzon, carta);
        /* resto */;
    }
}
void main()
{
    create_mailbox (buzon);
    send (buzon, null);
    paralelos (P(1), P(2), ..., P(n));
}

```

Figura 5.20. Exclusión mutua usando mensajes

```

/* programa productor consumidor */
const int
    capacidad = /* capacidad de
almacenamiento */;
    null = /* mensaje vacío */;
int i;
void productor()
{
    message pmsg;
    while (true)
    {
        receive (puedeproducir, pmsg);
        producir();
        send (puedeconsumir, pmsg);
    }
}

```

Sigue →

```

void consumidor()
{
    message cmsg;
    while (true)
    {
        receive (puedeconsumir, cmsg);
        consumir();
        send (puedeproducir, cmsg);
    }
}

void main()
{
    create_mailbox(puedeproducir);
    create_mailbox(puedeconsumir);
    for (int i = 1; i <= capacidad; i++)
        send(puedeproducir, null);
    paralelos (productor, consumidor);
}

```

Figura 5.21. Una solución al problema productor/consumidor con *buffer* acotado usando mensajes

Concurrencia: interbloqueo e inanición

Interbloqueo

- El bloqueo *permanente* de un conjunto de procesos que o bien compiten por recursos del sistema o se comunican entre sí
- No hay una solución eficiente
- Involucran necesidades conflictivas que afectan a los recursos de dos o más procesos

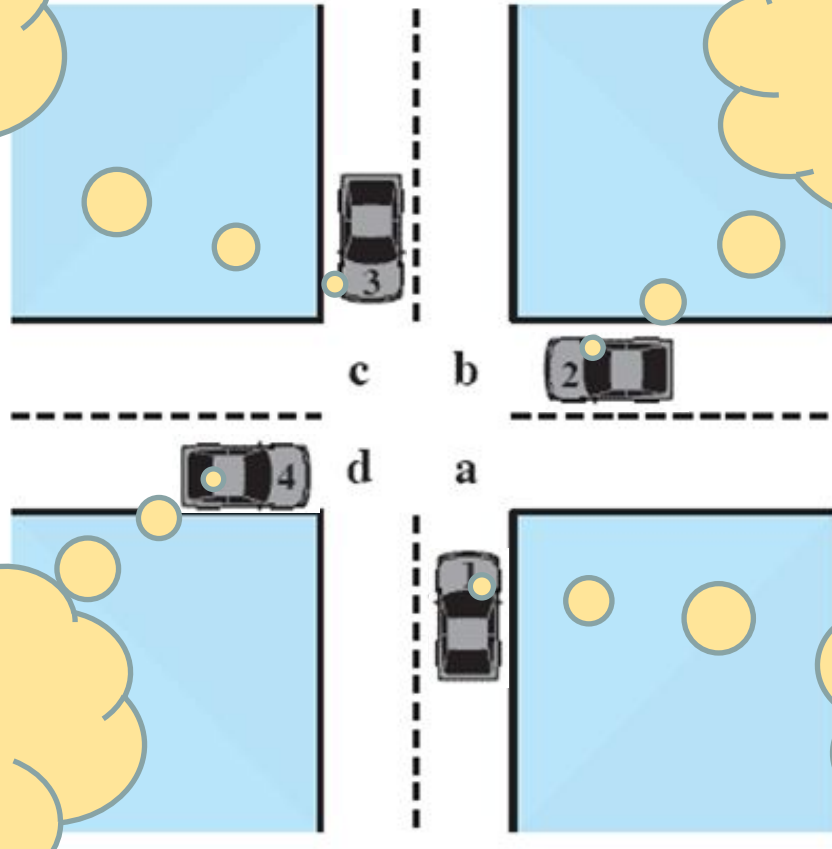
Posible interbloqueo

Yo necesito
cuadrantes
c y d

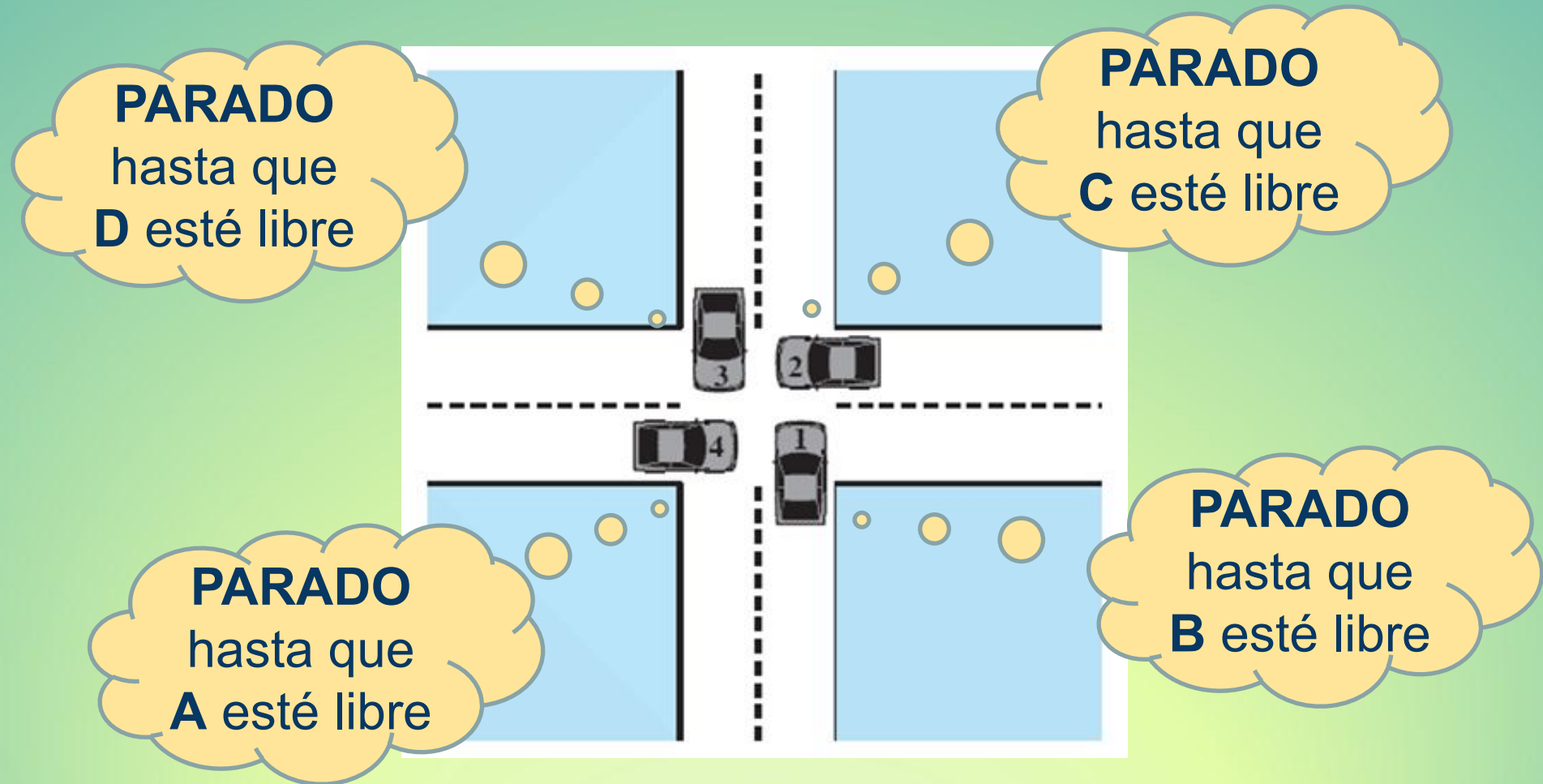
Yo necesito
cuadrantes
b y c

Yo necesito
cuadrantes
d y a

Yo necesito
cuadrantes
a y b



Interbloqueo



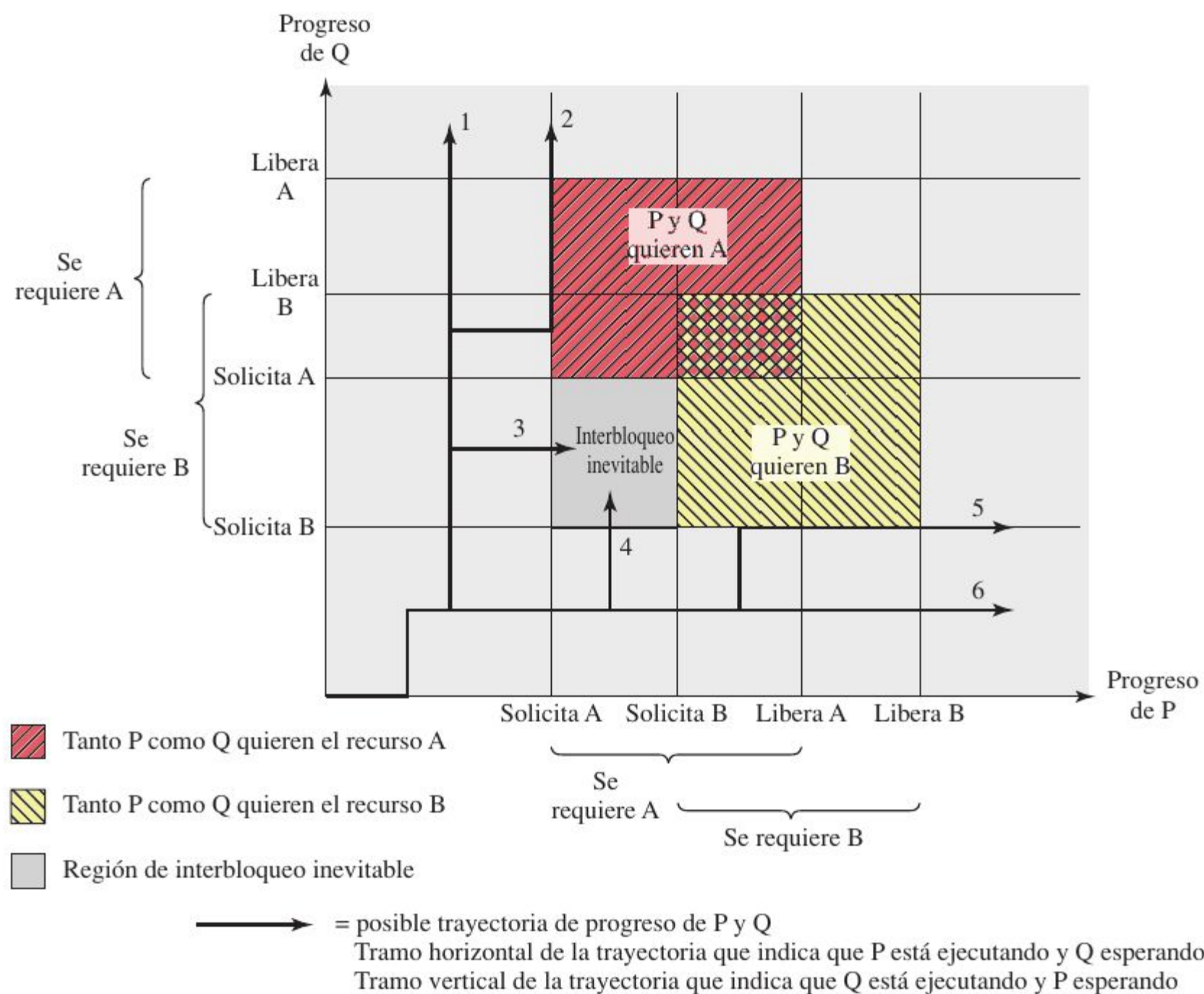


Figura 6.2. Ejemplo de interbloqueo

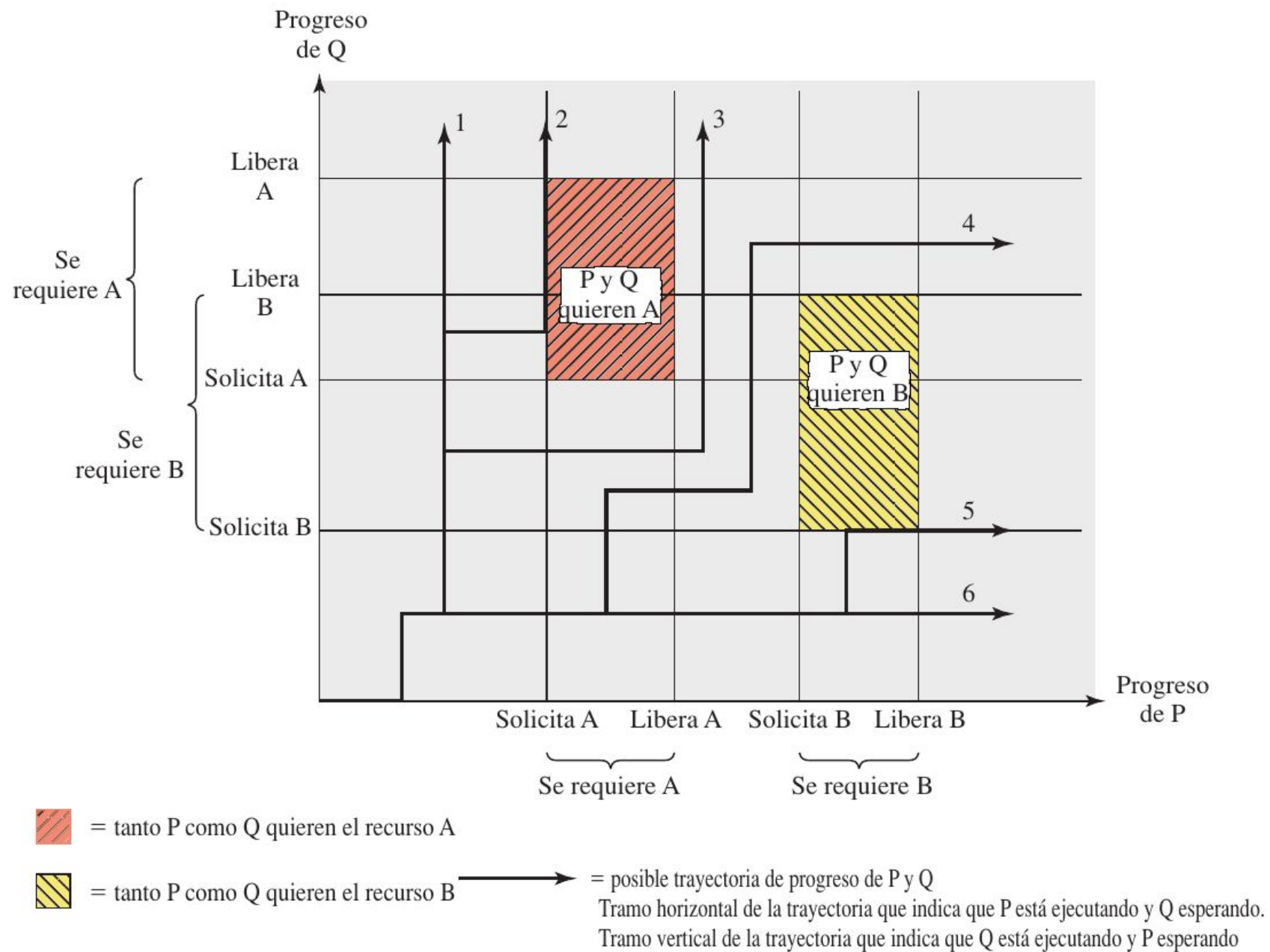


Figura 6.3. Ejemplo donde no hay interbloqueo [BACO03]

Recursos reutilizables

- Usados tan sólo por un proceso en cada momento y no se destruyen después de su uso
- Los procesos obtienen unidades del recurso que más tarde liberarán para que puedan volver a usarlas otros procesos
- Procesadores, canales de E/S, memoria principal y secundaria, dispositivos y estructuras de datos como ficheros, bases de datos y semáforos
- El interbloqueo se produce si cada proceso mantiene un recurso y solicita el otro

Ejemplo de interbloqueo

Proceso P		Proceso Q	
Paso	Acción	Paso	Acción
p_0	Solicita (D)	q_0	Solicita (C)
p_1	Bloquea (D)	q_1	Bloquea (C)
p_2	Solicita (C)	q_2	Solicita (D)
p_3	Bloquea (C)	q_3	Bloquea (D)
p_4	Realiza función	q_4	Realiza función
p_5	Desbloque (D)	q_5	Desbloque (C)
p_6	Desbloquea (C)	q_6	Desbloquea (D)

Figura 6.4. Ejemplo de dos procesos compitiendo por recursos reutilizables

Otro ejemplo de interbloqueo

- El espacio disponible para reservar es de **200 Kbytes** y se produce la secuencia siguiente de peticiones

P1	P2
...	...
Solicita 80 Kbytes;	Solicita 70 Kbytes;
...	...
Solicita 60 Kbytes;	Solicita 80 Kbytes;

- El interbloqueo sucede si ambos procesos progresan hasta su segunda petición

Recursos consumibles

- Creado (producido) y destruido (consumido)
- Interrupciones, señales, mensajes e información en *buffers* de E/S
- EL interbloqueo puede producirse si la función de recepción (Recibe) es bloqueante
- Puede darse una rara combinación de eventos que cause el interbloqueo

Ejemplo de interbloqueo

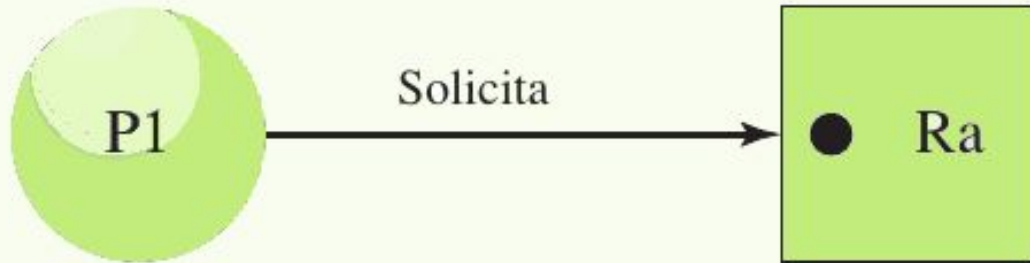
- El interbloqueo se produce si la función de recepción es bloqueante

P1
...
Recibe (P2);
...
Envía (P2, M1);

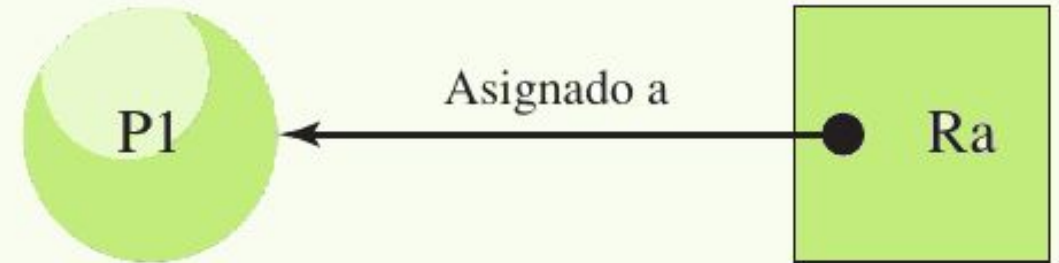
P2
...
Recibe P1;
...
Envía (P1,M2);

Grafos de asignación de recursos

- Grafo dirigido que representa el estado del sistema en lo que se refiere a recursos y procesos

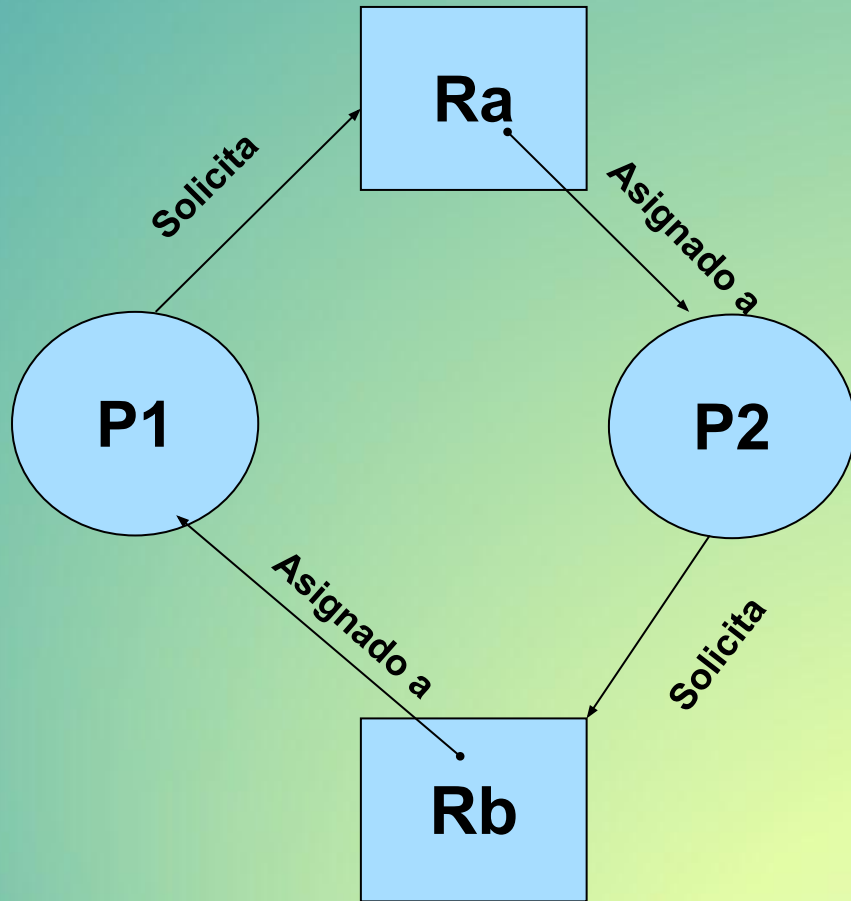


(a) Recurso solicitado

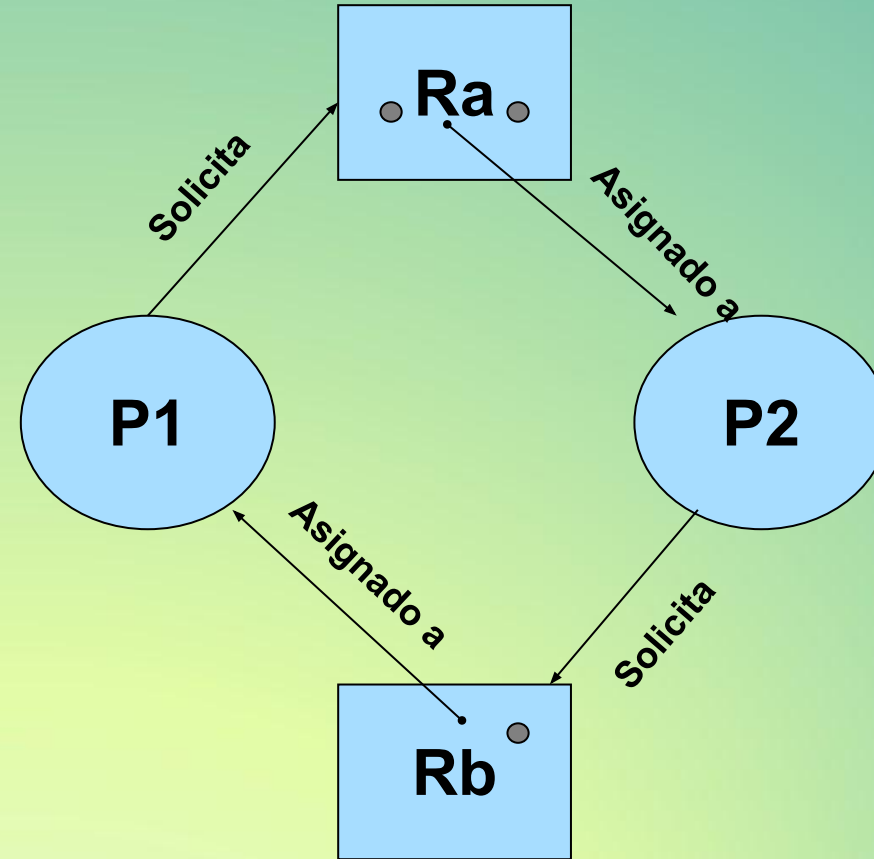


(b) Recurso asignado

Grafos de asignación de recursos



(c) Espera circular



(d) Sin interbloqueo

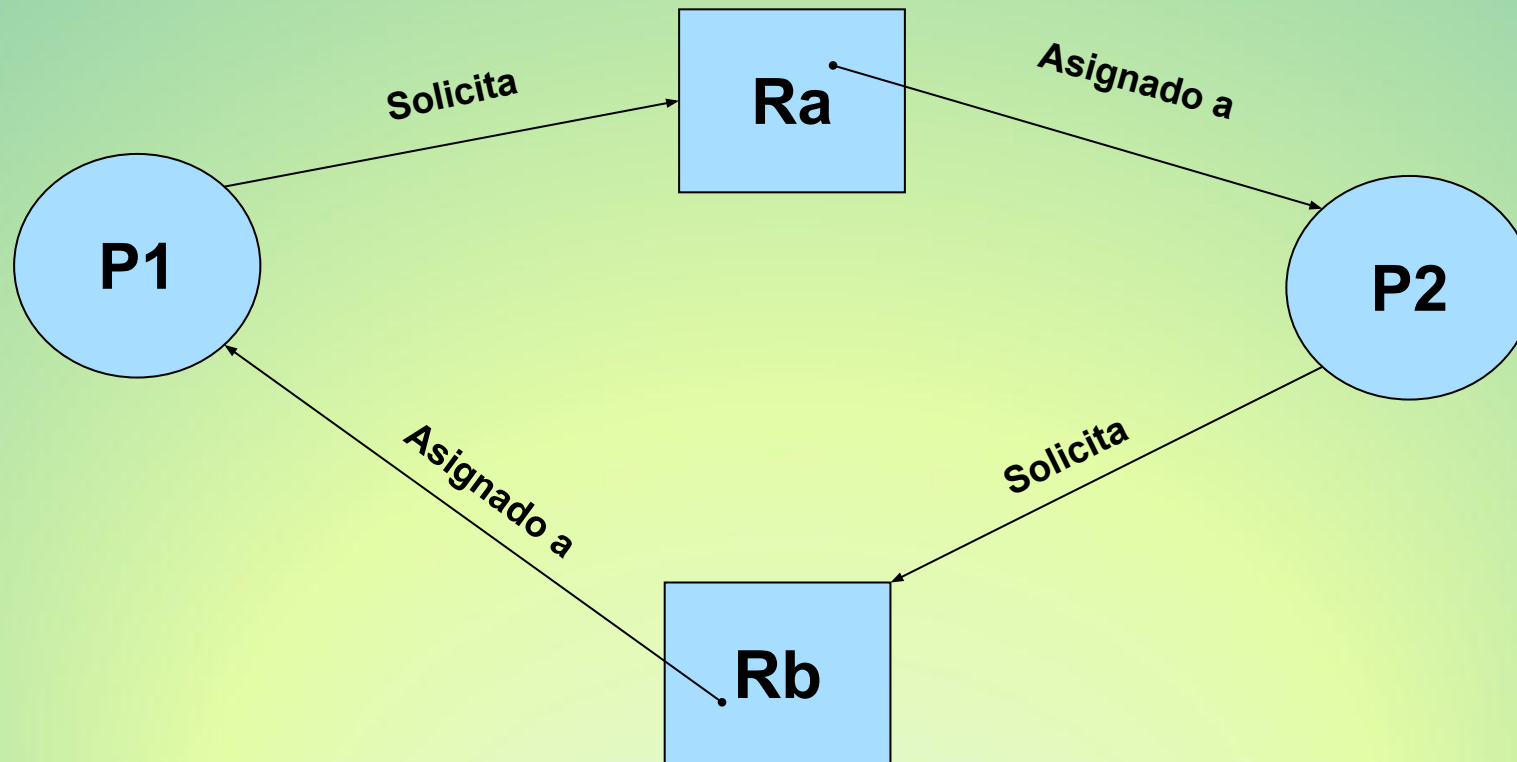
Figura 6.5. Ejemplos de grafos de asignación de recursos

Condiciones para el interbloqueo

- Exclusión mutua
 - Tan sólo un proceso puede usar un recurso en cada momento
- Retención y espera
 - Un proceso puede mantener los recursos asignados mientras espera la asignación de otros procesos
- Sin expropiación
 - No se puede forzar la expropiación de un recurso a un proceso que lo posee

Condiciones para el interbloqueo

- Espera circular
 - Existe una lista cerrada de procesos, de tal manera que cada proceso posee al menos un recurso necesitado por el siguiente proceso de la lista



(c) Espera circular

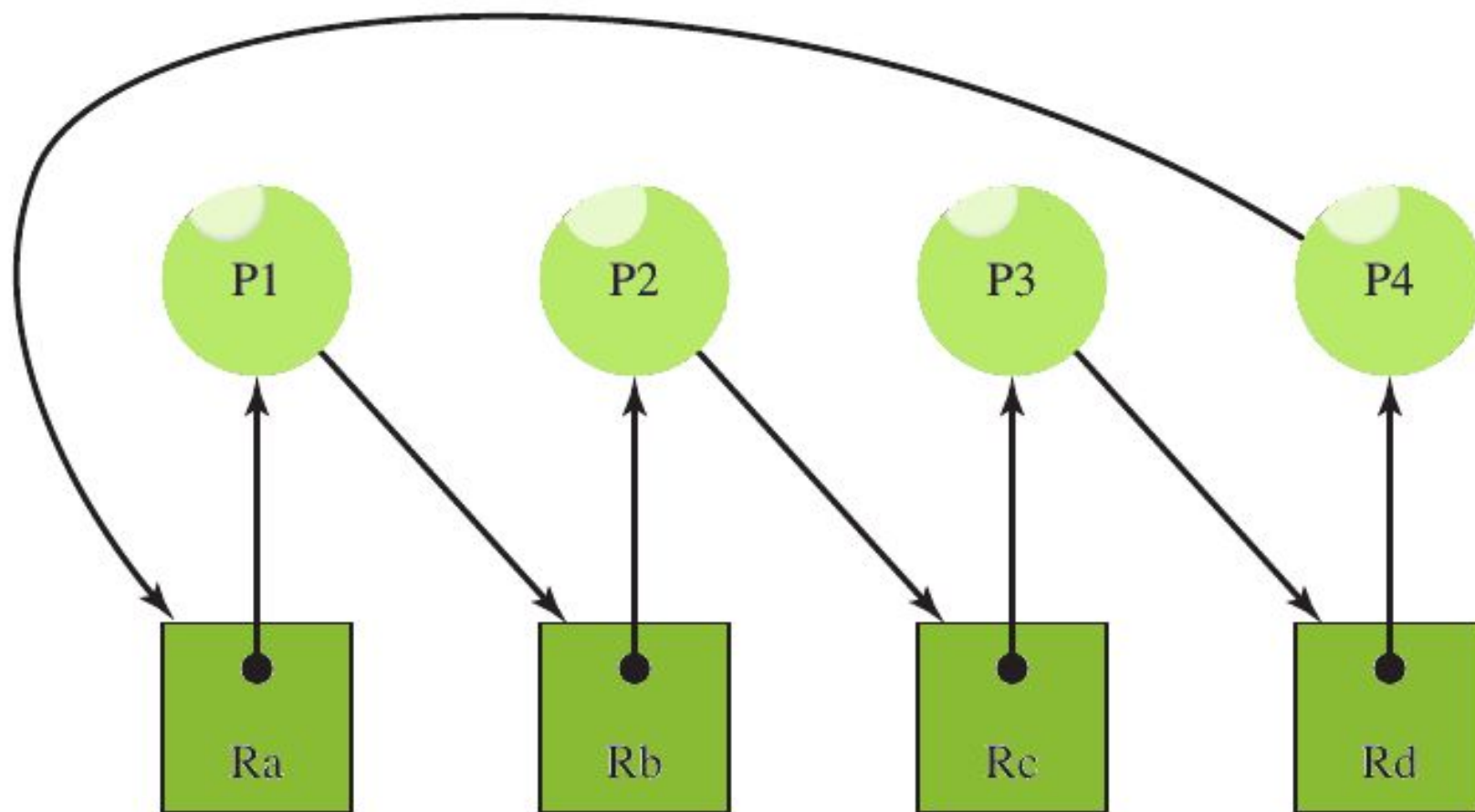


Figura 6.6. Grafo de asignación de recursos correspondiente a la Figura 6.1b.

Posibilidad de interbloqueo

- Exclusión mutua
- Sin expropiación
- Retención y espera

Existencia de interbloqueo

- Exclusión mutua
- Sin expropiación
- Retención y espera
- Espera circular

Prevención del interbloqueo

- Exclusión mutua
 - Debe proporcionarlo el sistema operativo
- Retención y espera
 - Un proceso debe solicitar al mismo tiempo todos sus recursos requeridos

Prevención del interbloqueo

- Sin expropiación
 - El proceso debe liberar sus recursos y solicitarlos de nuevo
 - El sistema operativo puede expropiar a un proceso y obligarle a liberar sus recursos
- Espera circular
 - Define un orden lineal entre los distintos tipos de recursos

Predicción del interbloqueo

- Se decide dinámicamente si la petición actual de reserva de un recurso, si se concede, podrá potencialmente causar un interbloqueo
- Requiere el conocimiento de las futuras solicitudes de recursos del proceso

Dos técnicas para predecir el interbloqueo

- No iniciar un proceso si sus demandas pudieran llevar al interbloqueo
- No conceder una petición adicional de un recurso por parte de un proceso si esta asignación pudiera provocar un interbloqueo

Denegación de la asignación de recursos

- Denominada algoritmo del banquero
- El estado del sistema refleja la asignación actual de recursos a procesos
- Un estado seguro es aquél en el que hay al menos una secuencia de asignación de recursos a los procesos que no implica un interbloqueo
- Un estado inseguro es, evidentemente, un estado que no es seguro

Determinación de un estado **seguro**

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Matriz de necesidad **N**

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Matriz de asignación **A**

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

N-A

R1	R2	R3
9	3	6

Vector de
recursos **R**

R1	R2	R3
1	1	2

Vector de
disponibles **D**

(a) Estado inicial

Determinación de un estado seguro

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Matriz de necesidad N

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Matriz de asignación A

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

N-A

R1	R2	R3
9	3	6

Vector de
recursos **R**

R1	R2	R3
6	2	3

Vector de
disponibles **D**

(b) P2 ejecuta hasta
completarse

Determinación de un estado seguro

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Matriz de necesidad N

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Matriz de asignación A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

N-A

R1	R2	R3
9	3	6

Vector de
recursos **R**

R1	R2	R3
7	2	3

Vector de
disponibles **D**

(c) P1 ejecuta hasta
completarse

Determinación de un estado seguro

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Matriz de necesidad N

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Matriz de asignación A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

N-A

R1	R2	R3
9	3	6

Vector de
recursos **R**

R1	R2	R3
9	3	4

Vector de
disponibles **D**

(d) P3 ejecuta hasta
completarse

Determinación de un estado **inseguro**

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Matriz de necesidad N

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Matriz de asignación A

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

N-A

R1	R2	R3
9	3	6

Vector de
recursos **R**

R1	R2	R3
1	1	2

Vector de
disponibles **D**

(a) Estado inicial

Determinación de un estado **inseguro**

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Matriz de necesidad N

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Matriz de asignación A

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

N-A

R1	R2	R3
9	3	6

Vector de
recursos **R**

R1	R2	R3
0	1	1

Vector de
disponibles **D**

(b) P1 solicita una unidad de
R1 y de R3

Lógica para la predicción del interbloqueo

```
struct state {  
    int resource[m];  
    int available[m];  
    int claim[n][m];  
    int alloc[n][m];  
}
```

(a) global data structures

```
if (alloc [i,*] + request [*] > claim [i,*])  
    < error >;                                /* total request > claim*/  
else if (request [*] > available [*])  
    < suspend process >;  
else {                                         /* simulate alloc */  
    < define newstate by:  
        alloc [i,*] = alloc [i,*] + request [*];  
        available [*] = available [*] - request [*] >;  
    }  
    if (safe (newstate))  
        < carry out allocation >;  
    else {  
        < restore original state >;  
        < suspend process >;  
    }  
}
```

(b) resource alloc algorithm

Lógica para la predicción del interbloqueo

```
boolean safe (state S) {
    int currentavail[m];
    process rest[<number of processes>];
    currentavail = available;
    rest = {all processes};
    possible = true;
    while (possible) {
        <find a process  $P_k$  in rest such that
            claim  $[k,*] - \text{alloc } [k,*] \leq \text{currentavail};$ >
        if (found) {                                /* simulate execution of  $P_k$  */
            currentavail = currentavail + alloc  $[k,*]$ ;
            rest = rest -  $\{P_k\}$ ;
        }
        else possible = false;
    }
    return (rest == null);
}
```

(c) test for safety algorithm (banker's algorithm)

Figure 6.9 Deadlock Avoidance Logic

Predicción del interbloqueo

Ventajas

- No es necesario expropiar a los procesos ni retroceder su ejecución como ocurre en la detección del interbloqueo
- Es menos restrictivo que la prevención del interbloqueo

Restricciones

- Deben establecerse por anticipado los requisitos máximos de recursos de cada proceso
- Los procesos involucrados deben ser independientes, es decir, el orden en el que se ejecutan no debe estar restringido por ningún requisito de sincronización
- Debe haber un número fijo de recursos que asignar
- Ningún proceso puede terminar mientras mantenga recursos

Detección de Interbloqueo

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Matriz de solicitud S

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Matriz de asignación A

R1	R2	R3	R4	R5
2	1	1	2	1

Vector de recursos

R1	R2	R3	R4	R5
0	0	0	0	1

Vector de disponibles

Figura 6.10. Ejemplo de detección del interbloqueo

Estrategias una vez que se ha detectado el interbloqueo

- Abortar todos los procesos involucrados en el interbloqueo
- Retroceder cada proceso en interbloqueo a algún punto de control (*checkpoint*) previamente definido y reiniciar todos los procesos
 - El interbloqueo original puede ocurrir
- Abortar sucesivamente los procesos en el interbloqueo hasta que éste deje de existir
- Expropiar sucesivamente los recursos hasta que el interbloqueo deje de existir

Criterios de selección de procesos de interbloqueo

- La menor cantidad de tiempo de procesador consumida hasta ahora
- La menor cantidad de salida producida hasta ahora
- El mayor tiempo restante estimado
- El menor número total de recursos asignados hasta ahora
- La menor prioridad

Ventajas y desventajas de las estrategias

Estrategia	Política de reserva de recursos	Esquemas alternativos	Principales ventajas	Principales desventajas
Prevención	Conservadora; infrautiliza recursos	Solicitud simultánea de todos los recursos	<ul style="list-style-type: none"> • Adecuada para procesos que realizan una sola ráfaga de actividad • No es necesaria la expropiación 	<ul style="list-style-type: none"> • Ineficiente • Retrasa la iniciación del proceso • Los procesos deben conocer sus futuros requisitos de recursos
		Expropiación	<ul style="list-style-type: none"> • Conveniente cuando se aplica a recursos cuyo estado se puede guardar y restaurar fácilmente 	<ul style="list-style-type: none"> • Expropia con más frecuencia de lo necesario
		Ordenamiento de recursos	<ul style="list-style-type: none"> • Es posible asegurarlo mediante comprobaciones en tiempo de compilación • No necesita cálculos en tiempo de ejecución ya que el problema se resuelve en el diseño del sistema 	<ul style="list-style-type: none"> • Impide solicitudes graduales de recursos
Predicción	A medio camino entre la detección y la prevención	Asegura que existe al menos un camino seguro	<ul style="list-style-type: none"> • No es necesaria la expropiación 	<ul style="list-style-type: none"> • El SO debe conocer los futuros requisitos de recursos de los procesos • Los procesos se pueden bloquear durante largos periodos
Detección	Muy liberal; los recursos solicitados se conceden en caso de que sea posible	Se invoca periódicamente para comprobar si hay interbloqueo	<ul style="list-style-type: none"> • Nunca retrasa la iniciación del proceso • Facilita la gestión en línea 	<ul style="list-style-type: none"> • Pérdidas inherentes por expropiación

El problema de los filósofos comensales

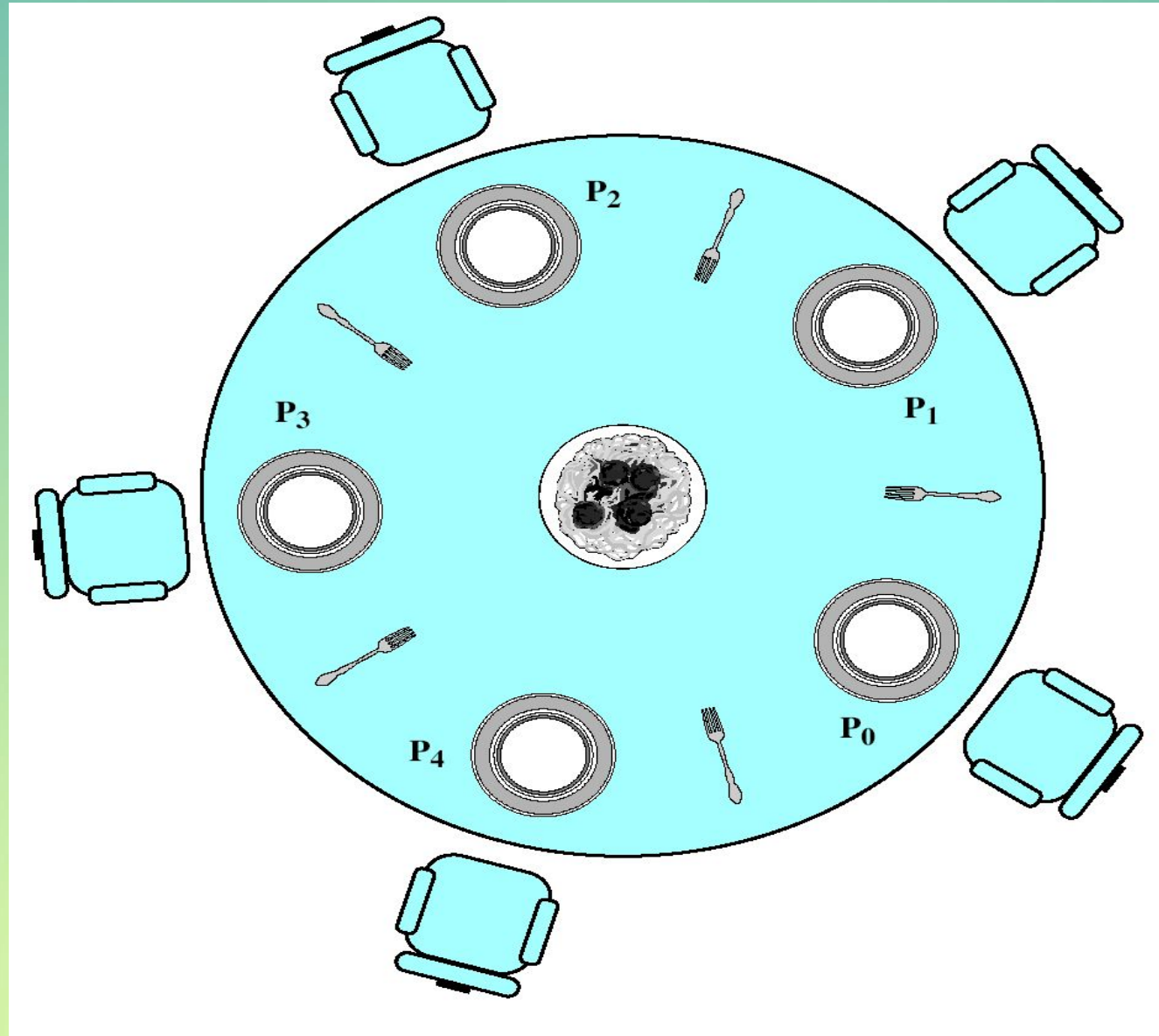


Figura 6.11. Disposición para los filósofos comensales

El problema de los filósofos comensales

```
/* programa filosofos_comensales */
semaforo tenedor [5] = {1};
int i;
void filosofo (int i)
{
    while (verdadero)
    {
        piensa ();
        wait (tenedor[i]);
        wait (tenedor[(i+1) mod 5]);
        come ();
        signal (tenedor[(i+1) mod 5]);
        signal (tenedor[i]);
    }
}
void main ()
{
    paralelos (filosofo (0), filosofo (1), filosofo (2),
filosofo (3), filosofo (4));
}
```

Figura 6.12. Una primera solución al problema de los filósofos comensales

El problema de los filósofos comensales

```
/* programa filosofos_comensales */
semaforo tenedor [5] = {1};
semaforo comedor = {4};
int i;
void filosofo (int i)
{
    while (verdadero)
    {
        piensa ();
        wait (comedor);
        wait (tenedor [i]);
        wait (tenedor [(i+1) mod 5]);
        come ();
        signal (tenedor [(i+1) mod 5]);
        signal (tenedor [i]);
        signal (comedor);
    }
}
void main ()
{
    paralelos (filosofo (0), filosofo (1), filosofo (2),
filosofo (3), filosofo (4));
}
```

Figura 6.13. Una segunda solución al problema de los filósofos comensales

El problema de los filósofos comensales

```
monitor controlador_de_comensales;
cond TenedorListo[5]           /* variable de condición para sincronizar */
boolean tenedor [5] = {verdadero}; /* disponibilidad de cada tenedor */
void obtiene_tenedores (int id_pr) /* id_pr es el número de ident. del filósofo */
{
    int izquierdo = id_pr;
    int derecho = (id_pr++) % 5;
    /* concede el tenedor izquierdo */
    if (!tenedor (izquierdo))
        cwait(TenedorListo[izquierdo]); /* encola en variable de condición */
    tenedor(izquierdo) = falso;
    /* concede el tenedor derecho */
    if (!tenedor(derecho))
        cwait(TenedorListo[derecho]); /* encola en variable de condición */
    tenedor(derecho) = falso;
}
void libera_tenedores (int id_pr)
{
    int izquierdo = id_pr;
    int derecho = (id_pr++) % 5;
    /* libera el tenedor izquierdo */
    if (empty(TenedorListo[izquierdo])); /* nadie espera por ese tenedor */
        tenedor(izquierdo) = verdadero;
    else /* despierta a un proceso que espera por este tenedor */
        csignal(TenedorListo[izquierdo]);
    /*libera el tenedor derecho */
    if (empty(TenedorListo[derecho])) /* nadie espera por este tenedor */
        tenedor(derecho) = verdadero;
    else /* despierta a un proceso que espera por este tenedor */
        csignal(TenedorListo[derecho]);
}
```

```
void filosofo [k=0 hasta 4] /* los cinco clientes filósofos */
{
    while (verdadero)
    {
        <piensa>;
        obtiene_tenedores(k); /* cliente solicita dos tenedores vía el monitor*/
        <come espaguetis>;
        libera_tenedores(k); /*cliente libera tenedores vía el monitor*/
    }
}
```

Figura 6.14. Una solución al problema de los filósofos comensales usando un monitor

El problema de los filósofos comensales

```
monitor controlador_de_comensales;
enum estados {pensando, hambriento, comiendo} estado[5];
cond requiereTenedor [5];          /* variable de condición */
void obtiene_tenedores (int id_pr)  /* id_pr es el número de ident. del filósofo */
{
    estado[id_pr] = hambriento; /* anuncia que tiene hambre */
    if ((estado[(id_pr+1) % 5] == comiendo)
        || (estado[(id_pr-1) % 5] == comiendo))
        cwait(requiereTenedor [id_pr]); /* espera si algún vecino come */
    estado[id_pr] = comiendo; /* continúa si ningún vecino come */
}
void libera_tenedores (int id_pr)
{
    estado[id_pr] = pensando;
    /* le da una oportunidad de comer al siguiente vecino (número mayor) */
    if ((estado[(id_pr+1) % 5] == hambriento)
        &&(estado[(id_pr+2)%5] != comiendo))
        csignal(requiereTenedor [id_pr+1]);
    /* le da una oportunidad de comer al anterior vecino (número menor) */
    else if ((estado[id_pr-1] % 5] == hambriento)
        &&(estado[(id_pr-2) % 5] != comiendo))
        csignal(requiereTenedor [id_pr-1]);
}
```

```
void filosofo[k=0 hasta 4] /*los cinco clientes filosofos*/
{
    while (verdadero)
    {
        <piensa>;
        obtiene_tenedores(k); /*cliente solicita dos tenedores vía el monitor*/
        <come espaguetis>;
        libera_tenedores(); /*cliente libera tenedores vía el monitor*/
    }
}
```

Figura 6.17. Otra solución al problema de los filósofos comensales usando un monitor

Mecanismos de concurrencia de UNIX

- Tuberías (*pipes*)
- Mensajes
- Memoria compartida
- Semáforos
- Señales

Valor	Nombre	Descripción
01	SIGHUP	Desconexión; enviada al proceso cuando el núcleo asume que el usuario de ese proceso no está haciendo trabajo útil
02	SIGINT	Interrupción
03	SIGQUIT	Abandonar; enviada por el usuario para provocar la parada del proceso y la generación de un volcado de su memoria (<i>core dump</i>)
04	SIGILL	Instrucción ilegal
05	SIGTRAP	<i>Trap</i> de traza; activa la ejecución de un código para realizar un seguimiento de la ejecución del proceso
06	SIGIOT	Instrucción IOT
07	SIGEMT	Instrucción EMT
08	SIGFPE	Excepción de coma flotante
09	SIGKILL	Matar; terminar el proceso
10	SIGBUS	Error de bus
11	SIGSEGV	Violación de segmento; el proceso intenta acceder a una posición fuera de su espacio de direcciones
12	SIGSYS	Argumento erróneo en una llamada al sistema
13	SIGPIPE	Escritura sobre una tubería que no tiene lectores asociados
14	SIGALRM	Alarma; emitida cuando un proceso desea recibir una señal cuando transcurra un determinado periodo de tiempo
15	SIGTERM	Terminación por software
16	SIGUSR1	Señal 1 definida por el usuario
17	SIGUSR2	Señal 2 definida por el usuario
18	SIGCHLD	Muerte de un proceso hijo
19	SIGPWR	Interrupción en el suministro de energía

Tabla 6.2. Señales UNIX

Mecanismos de concurrencia del núcleo de LINUX

- Incluye todos los mecanismos de concurrencia presentes en otros sistemas UNIX
- Las operaciones atómicas se ejecutan sin interrupción y sin interferencia

Operaciones atómicas en LINUX

Operaciones atómicas con enteros	
ATOMIC_INT (int i)	En una declaración: inicia un atomic_t con el valor i
int atomic_read(atomic_t *v)	Lee el valor entero de v
void atomic_set(atomic_t *v,int i)	Asigna el valor entero i a v
void atomic_add(int i,atomic_t *v)	Suma i a v
void atomic_sub(int i,atomic_t *v)	Resta i de v
void atomic_inc(atomic_t *v)	Suma 1 a v
void atomic_dec(atomic_t *v)	Resta 1 de v
int atomic_sub_and_test(int i,atomic_t *v)	Resta i de v; devuelve 1 si el resultado es cero; y 0 en caso contrario
int atomic_dec_and_test(atomic_t *v)	Resta 1 de v; devuelve 1 si el resultado es cero; y 0 en caso contrario
Operaciones atómicas con mapas de bits	
void set_bit(int n,void *dir)	Pone a 1 el bit n del mapa de bits apuntado por dir
void clear_bit(int n,void *dir)	Pone a 0 el bit n del mapa de bits apuntado por dir
void change_bit(int n,void *dir)	Invierte el valor del bit n del mapa de bits apuntado por dir; devuelve su valor previo
int test_and_set_bit(int n,void *dir)	Pone a 1 el bit n del mapa de bits apuntado por dir; devuelve su valor previo
int test_and_clear_bit(int n,void *dir)	Pone a 0 el bit n del mapa de bits apuntado por dir; devuelve su valor previo
int test_and_change_bit(int n,void *dir)	Invierte el valor del bit n del mapa de bits apuntado por dir; devuelve su valor previo
int test_bit(int n,void *dir)	Devuelve el valor del bit n del mapa de bits apuntado por dir

Mecanismos de concurrencia del núcleo de LINUX

- Cerrojos cíclicos
 - Se utilizan para proteger una sección crítica

<code>void spin_lock(spinlock_t *cerrojo)</code>	Adquiere el cerrojo especificado, comprobando cíclicamente el valor hasta que esté disponible
<code>void spin_lock_irq(spinlock_t *cerrojo)</code>	Igual que <code>spin_lock</code> , pero prohibiendo las interrupciones en el procesador local
<code>void spin_lock_irqsave(spinlock_t *cerrojo, unsigned long indicadores)</code>	Igual que <code>spin_lock_irq</code> , pero guardando el estado actual de las interrupciones en indicadores
<code>void spin_lock_bh(spinlock_t *cerrojo)</code>	Igual que <code>spin_lock</code> , pero prohibiendo la ejecución de mitades inferiores
<code>void spin_unlock(spinlock_t *cerrojo)</code>	Libera el cerrojo especificado
<code>void spin_unlock_irq(spinlock_t *cerrojo)</code>	Libera el cerrojo especificado y habilita las interrupciones locales
<code>void spin_unlock_irqrestore(spinlock_t *cerrojo, unsigned long indicadores)</code>	Libera el cerrojo especificado y restaura el estado de las interrupciones locales
<code>void spin_unlock_bh(spinlock_t cerrojo)</code>	Libera el cerrojo especificado y habilita la ejecución de mitades inferiores
<code>void spin_lock_init(spinlock_t *cerrojo)</code>	Inicia el cerrojo especificado
<code>int spin_trylock(spinlock_t *cerrojo)</code>	Intenta adquirir el cerrojo especificado, devolviendo un valor distinto de cero si lo posee si lo posee otro proceso y cero en caso contrario
<code>int spin_is_locked(spinlock_t *cerrojo)</code>	Devuelve un valor distinto de cero si el cerrojo lo posee otro proceso y cero en caso contrario

Mecanismos de concurrencia del núcleo de LINUX

Semáforos tradicionales	
void sema_init(struct semaphore *sem,int cont)	Inicia el semáforo creado dinámicamente con el valor cont
void init_MUTEX(struct semaphore *sem)	Inicia el semáforo creado dinámicamente con el valor 1 (inicialmente abierto)
void init_MUTEX_LOCKED(struct semaphore *sem)	Inicia el semáforo creado dinámicamente con el valor 0 (inicialmente cerrado)
void down(struct semaphore *sem)	Intenta adquirir el semáforo especificado, entrando en un bloqueo ininterrumpible si el semáforo no está disponible
int down_interruptible(struct semaphore *sem)	Intenta adquirir el semáforo especificado, entrando en un bloqueo interrumpible si el semáforo no está disponible; devuelve el valor -EINTR si se recibe una señal
int down_trylock(struct semaphore *sem)	Intenta adquirir el semáforo especificado, y devuelve un valor distinto de cero si el semáforo no está disponible
void up(struct semaphore *sem)	Libera el semáforo especificado
Semáforos de lectura-escritura	
void init_rwsem(struct rw_semaphore, *sem_le)	Inicia el semáforo creado dinámicamente con el valor 1
void down_read(struct rw_semaphore, *sem_le)	Operación down de los lectores
void up_read(struct rw_semaphore, *sem_le)	Operación up de los lectores
void down_write(struct rw_semaphore, *sem_le)	Operación down de los escritores
void up_write(struct rw_semaphore, *sem_le)	Operación up de los escritores

Tabla 6.5. Semáforos en Linux

Mecanismos de concurrencia del núcleo de Linux

<code>rmb()</code>	Impide que se cambie el orden de las lecturas evitando que se realicen después de la barrera
<code>wmb()</code>	Impide que se cambie el orden de las escrituras evitando que se realicen después de la barrera
<code>mb</code>	Impide que se cambie el orden de las lecturas y escrituras evitando que se realicen después de la barrera
<code>barrier()</code>	Impide al compilador cambiar el orden de las lecturas y escrituras evitando que se realicen después de la barrera
<code>smp_rmb()</code>	En SMP proporciona un <code>rmb()</code> y en UP proporciona un <code>barrier()</code>
<code>smp_wmb()</code>	En SMP proporciona un <code>wmb()</code> y en UP proporciona un <code>barrier()</code>
<code>smp_mb()</code>	En SMP proporciona un <code>mb()</code> y en UP proporciona un <code>barrier()</code>

SMP = multiprocesador simétrico

UP = uniprocador

Tabla 6.6. Operaciones de barrera de memoria en Linux

Funciones de sincronización de hilos de Solaris

- Cerrojos de exclusión mutua (*mutex*)
- Semáforos
- Cerrojos de múltiples lectores y un único escritor (lectores/escritor)
- Variables de condición

Funciones de sincronización de hilos de Solaris

propietario (3 octetos)
cerrojo (1 octeto)
en espera (2 octetos)
información específica del tipo (4 octetos) (posiblemente un ident. de <i>turnstile</i> tipo de relleno del cerrojo, o puntero a estadística)

(a) Cerrojo MUTEX

Tipo (1 octeto)
cerrojo de escritura (1 octeto)
en espera (2 octetos)
unión (4 octetos) (puntero a estadísticas o número de peticiones de escritura)
hilo propietario (4 octetos)

(c) Cerrojo de lectura/escritura

Tipo (1 octeto)
cerrojo de escritura (1 octeto)
en espera (2 octetos)
contador (4 octetos)

(b) Semáforo

en espera (2 octetos)

(d) Variable de condición

Figura 6.15. Estructura de datos de sincronización de Solaris

Mecanismos de concurrencia en windows

- Windows proporciona sincronización entre hilos como parte de su arquitectura.
- Métodos de sincronización:
 - Objetos despachador del ejecutivo (Executive dispatcher objects) usan funciones de espera
 - secciones críticas modo usuario
 - cerrojos lector-escritor delgados (slim reader-writer locks)
 - variables de condición

Funciones de espera

- Las funciones de espera permiten que un hilo bloquee su propia ejecución.
 - Las funciones de espera no retornan hasta que se cumplen los criterios especificados.
 - El tipo de función de espera determina el conjunto de criterios utilizado.

Objetos despachador

Tipo de objeto	Definición	Pasa al estado de señalado cuando	Efecto sobre los hilos en espera
Notificación de evento	Un aviso de que ha ocurrido un evento del sistema	Un hilo genera un evento	Desbloquea a todos
Sincronización de evento	Un aviso de que ha ocurrido un evento del sistema	Un hilo genera un evento	Se desbloquea un hilo
Mutex	Un mecanismo que proporciona exclusión mutua; equivalente a un semáforo binario	El hilo propietario u otro hilo libera el mutex	Se desbloquea un hilo
Semáforo	Un contador que regula el número de hilos que pueden usar un recurso	El contador del semáforo llega a cero	Desbloquea a todos
Temporizador con espera	Un contador que registra el paso del tiempo	Se cumple el tiempo especificado o expira el intervalo de tiempo	Desbloquea a todos
Fichero	Una instancia de un fichero abierto o un dispositivo de E/S	Se completa una operación de E/ <u>S</u>	Desbloquea a todos
Proceso	Una invocación de un programa, incluyendo el espacio de direcciones y los recursos requeridos para ejecutar el programa	El último hilo termina	Desbloquea a todos
Hilo	Una entidad ejecutable dentro de un proceso	El hilo termina	Desbloquea a todos

Tabla 6.7. Objetos de sincronización de Windows

Nota: las filas que no están coloreadas corresponden a objetos que sólo existen para sincronización

Secciones críticas

- Mecanismo de sincronización similar a mutex
 - Excepto que los objetos de sección crítica sólo los hilos del mismo proceso pueden usarlos
- Si el sistema es multiprocesador, el código intentará conseguir un “spin-lock”
 - Como último recurso, si no se consigue el spin-lock, el objeto despachador se usa para bloquear el hilo de manera de permitir al kernel despachar otro hilo al procesador

Cerrojos lector-escritor delgados

- Windows Vista agrega un lector-escritor modo usuario
- El cerrojo lector-escritor entra al kernel para bloquear sólo después de intentar usar un spin-lock
- Se les dice delgados (slim) porque normalmente requieren asignación de memoria muy pequeña

Variables de condición

- Windows Vista también agrega variables de condición
- El proceso debe declarar e inicializar una `CONDITION_VARIABLE`
- Se usan tanto con secciones críticas como con cerrojos lector escritor delgados (SRW locks)