# II. Administración de procesos

# 2.3 Comunicación entre procesos

# Comunicación entre procesos

- En una tubería de Shell la salida del primer procesos debe pasarse al segundo…
- InterProcessCommunication – IPC
- Importante:
  - ¿Cómo un proceso le puede pasar información a otro?
  - ¿Cómo se asegura que dos o más procesos no se estorben entre sí?
  - ¿Cómo asegurar un buen manejo de secuencias cuando se presentan dependencias?

# Práctica – Redireccionamiento

- …

# Práctica de memoria compartida

- …

# 2.4 Sincronización entre procesos/hilos

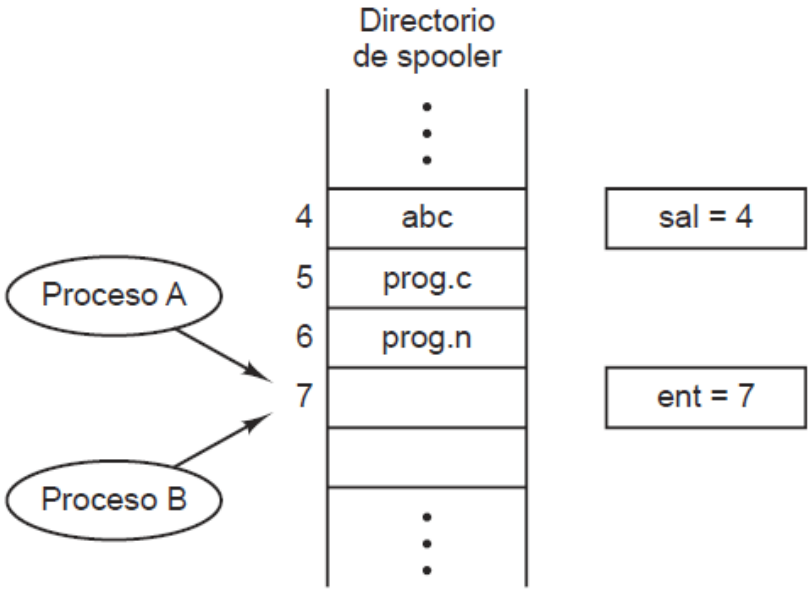
# Condiciones de carrera

- En algunos S.O., los procesos que trabajan en conjunto pueden compartir cierto espacio de almacenamiento en el que pueden leer y escribir datos.

- Ej.: 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.

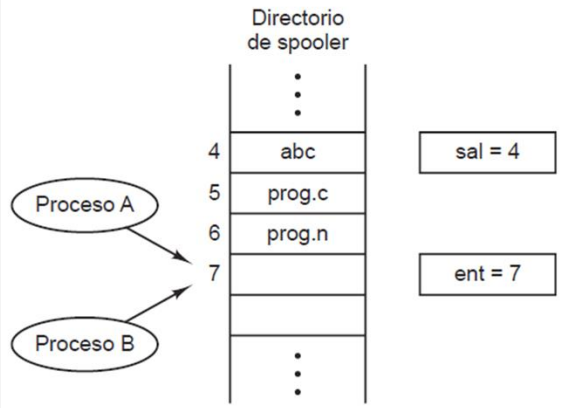
# Condiciones de carrera

- El **directorio de spooler** tiene una cantidad muy grande de ranuras, numeradas como 0, 1, 2, …,
- Cada ranura puede contener el nombre de un archivo.
- Hay dos variables compartidas: **sal**, que apunta al siguiente archivo a imprimir, y **ent**, que apunta a la siguiente ranura libre en el directorio.
- Las dos variables se podrían mantener 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.



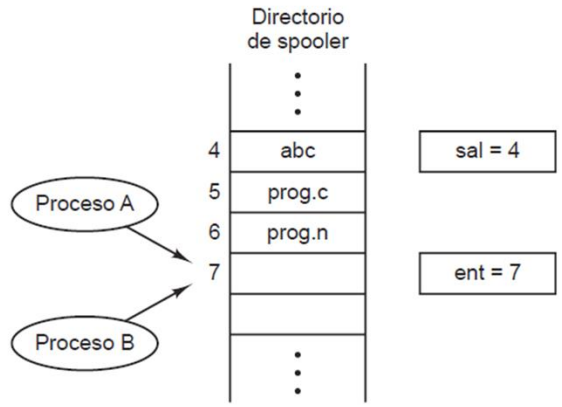
# Condiciones de carrera >> Ley de murphy

- 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**.
- *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.



# Condiciones de carrera >> Ley de murphy

- 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.**
- Condiciones de carrera. Situaciones 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.



# Regiones críticas

- ¿Cómo evitamos las condiciones de carrera?
  - Buscando alguna manera de prohibir que más de un proceso lea y escriba los datos compartidos al mismo tiempo.
  - 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.

# Regiones críticas

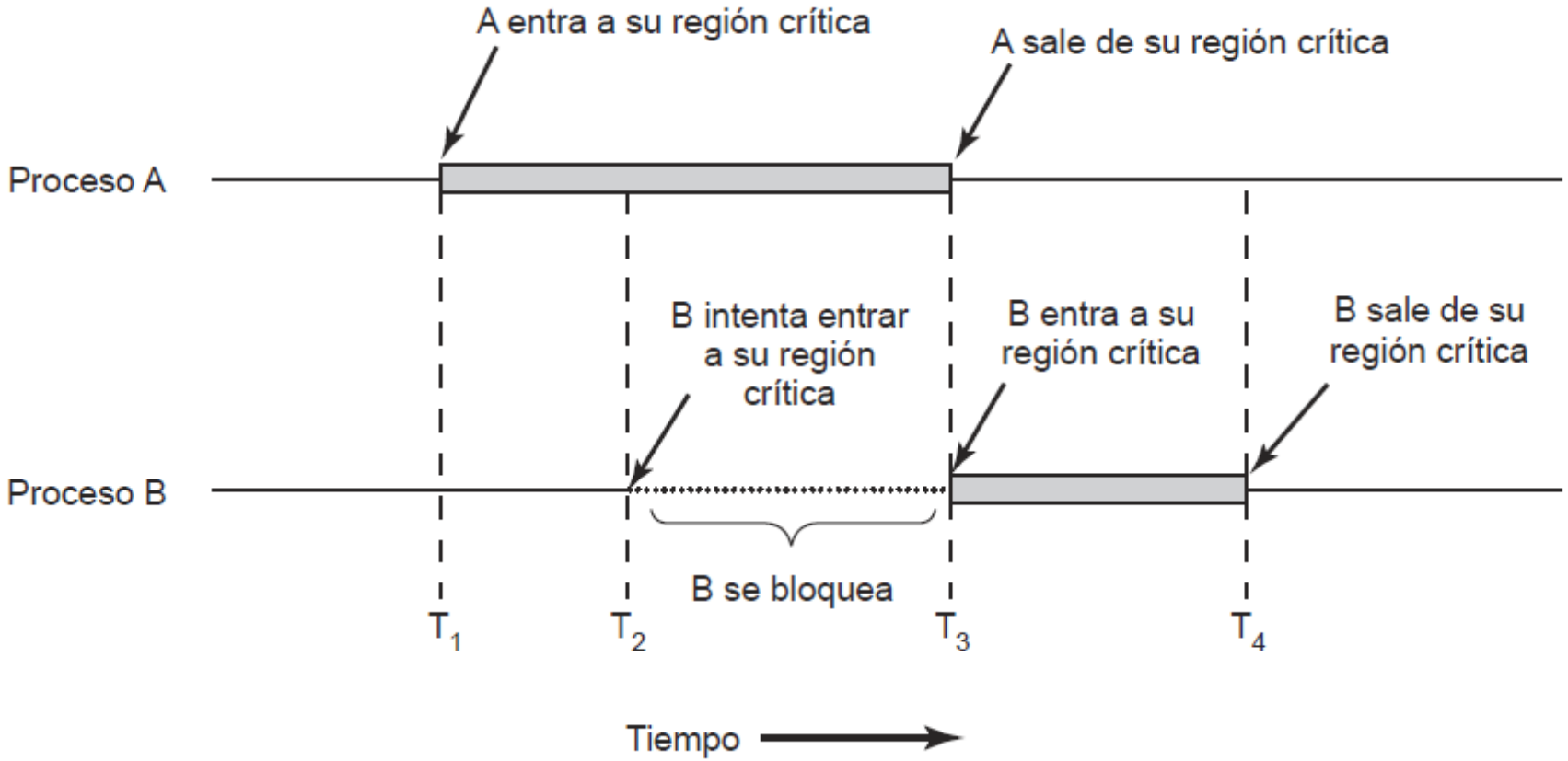
- El **problema de evitar las condiciones de carrera** también se puede formular de una manera abstracta.
- **Parte del tiempo**, un proceso está ocupado realizando cálculos internos y otras cosas que **no producen condiciones de carrera**.
- Sin embargo, **algunas veces** un proceso tiene que acceder 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.

# Regiones críticas

- 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.

# Regiones críticas




Exclusión mutua mediante el uso de regiones críticas

# Exclusión mutua con espera ocupada

- **Deshabilitando interrupciones (1/4)**
  - 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.
  - Después de todo, **la CPU sólo se conmuta de un proceso a otro como resultado de una interrupción del reloj o de otro tipo**, y con las interrupciones desactivadas la CPU no se conmutará a otro proceso.
  - Una vez que un proceso ha deshabilitado las interrupciones, puede examinar y actualizar la memoria compartida sin temor de que algún otro proceso intervenga.

# Exclusión mutua con espera ocupada

- **Deshabilitando interrupciones (2/4)**
  - 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.

# Exclusión mutua con espera ocupada

- **Deshabilitando interrupciones (3/4)**
  - Con frecuencia es conveniente para el mismo kernel deshabilitar las interrupciones por unas cuantas instrucciones mientras actualiza variables o listas.
  - Por ejemplo:
    - Si ocurriera una interrupción mientras la lista de procesos se encuentra en un estado inconsistente, podrían producirse condiciones de carrera.
  - 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.**

# Exclusión mutua con espera ocupada

- **Deshabilitando interrupciones (4/4)**
  - La posibilidad de lograr la exclusión mutua al deshabilitar las interrupciones (incluso dentro del kernel) está disminuyendo día con día debido al creciente número de chips multinúcleo (4 – 8, 16…) que se encuentran hasta en las PCs de bajo rendimiento.
  - En un **multinúcleo** (es decir, sistema con multiprocesadores) al deshabilitar las interrupciones de una CPU no se evita que las demás CPUs interfieran con las operaciones que la primera CPU está realizando.
  - En consecuencia**, se requieren esquemas más sofisticados.**

# Exclusión mutua con espera ocupada

- **Variables de candado (1/3)**
  - 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.

# Exclusión mutua con espera ocupada

- **Variables de candado (2/3)**
  - Por desgracia, esta idea contiene exactamente el mismo error fatal que vimos en el directorio de spooler.
    - Suponga que un proceso lee el candado y ve que es 0.
    - Antes de que pueda fijar el candado a 1, otro proceso se planifica para ejecutarse y fija el candado a 1.
    - Cuando el primer proceso se ejecuta de nuevo, también fija el candado a 1 y por lo tanto dos procesos se encontrarán en sus regiones críticas al mismo tiempo.

# Exclusión mutua con espera ocupada

- **Variables de candado (3/3)**
  - ¿Se podrá resolver con una segunda verificación justo antes de almacenar el nuevo valor?
  - 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.

# Exclusión mutua con espera ocupada

- **Alternancia estricta (1/6)**

```
while (TRUE) {
    while (turno != 0)      /* ciclo */ ;
    region_critica();
    turno = 1;
    region_nocritica();
}
```

(a)

```
while (TRUE) {
    while (turno != 1)      /* ciclo */ ;
    region_critica();
    turno = 0;
    region_nocritica();
}
```

(b)

**Figura 2-23.** Una solución propuesta para el problema de la región crítica. (a) Proceso 0. (b) Proceso 1. En ambos casos, asegúrese de tener en cuenta los signos de punto y coma que terminan las instrucciones `while`.

# Exclusión mutua con espera ocupada

- **Alternancia estricta (2/6)**

Turno -> Lleva la cuenta de a qué proceso le toca entrar a su región crítica y examinar o actualizar la memoria compartida.

```
while (TRUE) {
    while (turno != 0)        /* ciclo */ ;
    region_critica();
    turno = 1;
    region_nocritica();
}
```

(a)

```
while (TRUE) {
    while (turno != 1)        /* ciclo */ ;
    region_critica();
    turno = 0;
    region_nocritica();
}
```

(b)

**Figura 2-23.** Una solución propuesta para el problema de la región crítica. (a) Proceso 0. (b) Proceso 1. En ambos casos, asegúrese de tener en cuenta los signos de punto y coma que terminan las instrucciones `while`.

# Exclusión mutua con espera ocupada

- **Alternancia estricta (3/6)**
  - En un inicio turno = 0
  - Al principio, el proceso 0 (a) inspecciona turno, descubre que es 0 y entra a su región crítica.
  - El proceso 1 también descubre que es 0 y por lo tanto se queda en un ciclo estrecho, evaluando turno en forma continua para ver cuándo se convierte en 1.
  - A la acción de evaluar en forma continua una variable hasta que aparezca cierto valor se le conoce como **espera ocupada**.
    - Por lo general se debe evitar, ya que **desperdicia tiempo de la CPU**.
    - Sólo se utiliza cuando hay una expectativa razonable de que la espera será corta.
  - A un candado que utiliza la espera ocupada se le conoce como **candado de giro**.

# Exclusión mutua con espera ocupada

- **Alternancia estricta (4/6)**
  - Cuando el proceso 0 sale de la región crítica establece turno a 1, para permitir que el proceso 1 entre a su región crítica.
  - Suponga que el proceso 1 sale rápidamente de su región crítica, de manera que ambos procesos se encuentran en sus **regiones no críticas**, con **turno establecido en 0**.
  - Ahora el proceso 0 ejecuta todo su ciclo rápidamente, saliendo de su región crítica y estableciendo turno a 1.
  - En este punto, turno es 1 y ambos procesos se están ejecutando en sus regiones no críticas.

# Exclusión mutua con espera ocupada

- **Alternancia estricta (5/6)**
  - De repente, el proceso 0 termina en su región no crítica y regresa a la parte superior de su ciclo.
  - Por desgracia no puede entrar a su región crítica ahora, debido a que turno es 1 y el proceso 1 está ocupado con su región no crítica.
  - El proceso 0 espera en su ciclo while hasta que el proceso 1 establezca turno a 0.

- **Tomar turnos no es una buena idea cuando uno de los procesos es mucho más lento que el otro.**

# Exclusión mutua con espera ocupada

- **Alternancia estricta (6/6)**
  - Esta situación viola la condición 3 antes establecida: El proceso 0 está siendo bloqueado por un proceso que no está en su región crítica.
  - Volviendo al directorio de spooler antes mencionado, si ahora asociamos la región crítica con la lectura y escritura del directorio de spooler, el proceso 0 no podría imprimir otro archivo debido a que el proceso 1 está haciendo otra cosa.
  - Esta solución requiere que los dos procesos se alternen de manera estricta al entrar en sus regiones críticas (por ejemplo, al poner archivos en la cola de impresión).
  - Ninguno podría poner dos archivos en la cola al mismo tiempo.

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

# Exclusión mutua con espera ocupada

- **Solución de Peterson (1/5)**
  - Al combinar la idea de tomar turnos con la idea de las variables de candado y las variables de advertencia, un matemático holandés llamado T. Dekker fue el primero en idear una solución de software para el problema de la exclusión mutua que no requiere de una alternancia estricta.
  - En 1981, G.L. Peterson descubrió una manera mucho más simple de lograr la exclusión mutua, con lo cual la solución de Dekker se hizo obsoleta.
  - El algoritmo de Peterson consiste de dos procedimientos escritos en ANSI C, lo cual significa que se deben suministrar prototipos para todas las funciones definidas y utilizadas.

# Exclusión mutua con espera ocupada

- **Solución de Peterson (2/5)**

```
#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 */
}
```

# Exclusión mutua con espera ocupada

- **Solución de Peterson (3/5)**
  - 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

# Exclusión mutua con espera ocupada

- **Solución de Peterson (4/5)**
  - 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.

# Exclusión mutua con espera ocupada

- **Solución de Peterson (5/5)**
  - 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.

# Exclusión mutua con espera ocupada

- **Solución de Dekker (1/3)**
  - Utiliza la variable turno, que sirve para establecer la prioridad relativa de los dos procesos y su actualización se realiza en la sección crítica, lo que evita que pueda haber interferencias entre los procesos.

# Exclusión mutua con espera ocupada

- **Solución de Dekker (2/3)**
  - El programa se inicia con el valor de turno igual a 1, lo que da prioridad al proceso P1.
  - Si ambos procesos piden a la vez el acceso a su sección crítica, ponen en activo sus respectivos indicadores y comprueban si el indicador del otro está activado.
  - Ambos encuentran que sí, por lo que pasan a evaluar el turno.
  - El segundo se encuentra con que no es su turno, desactiva su indicador y se queda en espera de que lo sea.
  - P1 comprueba que sí es el suyo y pasa a valorar el estado del indicador de P2, entrará en su sección crítica y dará el turno a P2 antes de desactivar su indicador.
  - Esto permite que el proceso P2 gane el acceso a su sección crítica aunque el proceso P 1 haga una nueva solicitud de entrar a la región crítica inmediatamente después de desactivar su indicador.

# Exclusión mutua con espera ocupada

- **La instrucción TSL (1/10)**
  - TEST AND SET LOCK (probar y fijar candado)
  - Algunas computadoras, en especial las diseñadas **con varios procesadores** en mente, tienen una instrucción como **TSL REGISTRO, CANDADO** (Evaluar y fijar el candado).

# Exclusión mutua con espera ocupada

- **La instrucción TSL (2/10)**
  - **Lee el contenido** de la palabra de memoria **candado** y lo guarda en el registro RX, y después **almacena** un valor distinto de cero en la **dirección de memoria candado**.
  - Se garantiza que **las operaciones** de **leer** la **palabra** y **almacenar** un **valor** en ella **serán indivisibles**; ningún otro procesador puede acceder a la palabra de memoria sino hasta que termine la instrucción.
  - La CPU que ejecuta la instrucción TSL **bloquea el bus de memoria** para impedir que otras CPUs accedan a la memoria hasta que termine.

# Exclusión mutua con espera ocupada

- **La instrucción TSL (3/10)**
  - Bloquear el bus de memoria es una acción muy distinta de la de deshabilitar las interrupciones.
  - Al deshabilitar las interrupciones y después realizar una operación de lectura en una palabra de memoria, seguida de una operación de escritura, no se evita que un segundo procesador en el bus acceda a la palabra entre la lectura y la escritura.

# Exclusión mutua con espera ocupada

- **La instrucción TSL (4/10)**
  - De hecho, si se deshabilitan las interrupciones en el procesador 1 no se produce efecto alguno en el procesador 2.
  - La única forma de mantener el procesador 2 fuera de la memoria hasta que el procesador 1 termine es bloquear el bus, para lo cual se requiere una herramienta de hardware especial (básicamente, una línea de bus que afirme que el bus está bloqueado y no disponible para los demás procesadores aparte del que lo bloqueó).

# Exclusión mutua con espera ocupada

- **La instrucción TSL (5/10)**
  - Para usar la instrucción TSL necesitamos una variable compartida (*candado*) que coordine el acceso a la memoria compartida.
  - Cuando *candado* es 0, cualquier proceso lo puede fijar en 1 mediante el uso de la instrucción TSL y después una lectura o escritura en la memoria compartida.
  - Cuando termina, el proceso establece *candado* de vuelta a 0 mediante una instrucción **move** ordinaria.

# Exclusión mutua con espera ocupada

- **La instrucción TSL (6/10)**

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


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

# Exclusión mutua con espera ocupada

- **La instrucción TSL (7/10)**
  - ¿Cómo se puede utilizar esta instrucción para evitar que dos procesos entren al mismo tiempo en sus regiones críticas?
    - Utilizando una subrutina de cuatro instrucciones en un lenguaje ensamblador ficticio (pero común).
    - La primera instrucción copia el antiguo valor de *candado* en el registro y después fija el *candado* a 1.
    - Después, el valor anterior se compara con 0.
    - Si es distinto de cero, el candado ya estaba cerrado, por lo que el programa sólo regresa al principio y lo vuelve a evaluar.
    - Tarde o temprano se volverá 0 (cuando el proceso que esté actualmente en su región crítica se salga de ella) y la subrutina regresará, con el bloqueo establecido.
    - Es muy simple quitar el bloqueo. El programa sólo almacena un 0 en *candado*.
    - No se necesitan instrucciones especiales de sincronización.

# Exclusión mutua con espera ocupada

- **La instrucción TSL (8/10)**
  - Es una solución simple y directa para el problema de regiones críticas.
  - Antes de entrar a su región crítica, un proceso llama a entrar_region, que lleva a cabo una espera ocupada hasta que el candado está abierto; después adquiere el candado y regresa.
  - Después de la región crítica, el proceso llama a salir_region, que almacena un 0 en candado.
  - Al igual que con todas las soluciones basadas en regiones críticas, los procesos deben llamar a entrar_region y salir_region en los momentos correctos para que el método funcione.
  - Si un proceso hace trampa, la exclusión mutua fallará.

# Exclusión mutua con espera ocupada

- **La instrucción TSL (9/10)**

```
entrar_region:
        MOVE REGISTRO,#1                |coloca 1 en el registro
        XCHG REGISTRO,CANDADO           |intercambia el contenido del registro y la variable candado
        CMP REGISTRO,#0                 |¿era candado cero?
        JNE entrar_region               |si era distinto de cero, el candado está cerrado, y se repite
        RET                             |regresa al que hizo la llamada; entra a región crítica


salir_region:
    MOVE CANDADO,#0                     |almacena 0 en candado
    RET                                 |regresa al que hizo la llamada
```

**Figura 2-26.** Cómo entrar y salir de una región crítica mediante la instrucción XCHG.

# Exclusión mutua con espera ocupada

- **La instrucción TSL (10/10)**
  - Una instrucción alternativa para TSL es XCHG, que intercambia el contenido de dos ubicaciones en forma atómica.
  - Por ejemplo:
    - Un registro y una palabra de memoria.
    - Es en esencia el mismo código que la solución con TSL.
    - Todas las CPUs Intel x86 utilizan la instrucción XCHG para la sincronización de bajo nivel.

# Dormir y despertar

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

# Dormir y despertar

- Este método no desperdicia tiempo de la CPU, pero puede tener efectos inesperados.
  - Considere una computadora con dos procesos:
    - H con prioridad alta y
    - L con prioridad baja.
  - Las reglas de planificación son tales que H se ejecuta cada vez que se encuentra en el estado listo.

# Dormir y despertar

- En cierto momento, con L en su región crítica, H cambia al estado listo para ejecutarse.
- Entonces H empieza la espera ocupada, pero como L nunca se planifica mientras H está en ejecución, L nunca tiene la oportunidad de salir de su región crítica, por lo que H itera en forma indefinida.

- A esta situación se le conoce algunas veces como el **problema de inversión de prioridades**.

# Dormir y despertar

- Primitivas de comunicación entre procesos que bloquean en vez de desperdiciar tiempo de la CPU cuando no pueden entrar a sus regiones críticas.
  - El par sleep (dormir) y wakeup (despertar).
  - **Sleep** es una llamada al sistema que hace que el proceso que llama se bloquee / desactive / suspenda hasta que otro proceso lo despierte.
  - La llamada **wakeup** tiene un parámetro, el proceso que se va a despertar o activar.
- 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.

# Dormir y despertar

- **Ejemplo: El problema del productor-consumidor**

- 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.

# Dormir y despertar

- **Ejemplo: El problema del productor-consumidor**
  - 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.

# Dormir y despertar

- 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.

# Dormir y despertar

- 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.

- Para expresar llamadas al sistema como sleep y wakeup en C, las mostraremos 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.

# Dormir y despertar

```
#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 */
        }
}
```

# Dormir y despertar

- 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.

- **Ejemplo de condición de carrera:**
  - 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.

# Dormir y despertar

- **Ejemplo de condición de carrera:**
  - 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.**

# Dormir y despertar

- La esencia del problema aquí es que una señal que se envía para despertar a un proceso que no está dormido (todavía) se pierde.

- 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.

# Dormir y despertar

- 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**.
- **Podríamos hacer otra modificación** y agregar un segundo bit de espera de despertar, tal vez hasta 8 o 32 de ellos, pero **en principio el problema sigue ahí**.

# Semáforos

- E. W. Dijkstra (1965) 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.

# Semáforos

- Un semáforo binario es un indicador de condición (S) que registra si un recurso está disponible o no.
- Un semáforo binario sólo puede tomar dos valores: 0 y 1.
- Para un semáforo binario,
  - S=1 → El recurso está disponible y la tarea lo puede utilizar.
  - S=0 → El recurso no está disponible y el proceso debe esperar.

# Semáforos

- Los semáforos se implementan con una cola de tareas o de condición a la cual se añaden los procesos que están en espera del recurso.

- Sólo se permiten tres operaciones sobre un semáforo:
  1) Inicializa.
  2) Espera (wait).
  3) Señal (signal).

# Semáforos

- En algunos textos, se utilizan las notaciones P (espera) y V (señal) para las operaciones, ya que ésta fue la notación empleada originalmente por Dijkstra para referirse a las operaciones.
- P y V provienen de la primera letra de su nombre en holandés: proberen (comprobar) y verhogen (incrementar).
- También algunos autores las denominan up y down respectivamente
- Un semáforo binario es un tipo de datos especial que sólo puede tomar los valores 0 y 1, con una cola de tareas asociada y con sólo tres operaciones para actuar sobre él.

# Semáforos

- **Las operaciones son** procedimientos que se implementan como **acciones indivisibles** y por ello la comprobación y cambio de valor del indicador se efectúa de manera real como una sola operación.
- En sistemas con un **único procesador bastará con inhibir las interrupciones** durante la ejecución de las operaciones del semáforo.
- En los **sistemas multiprocesador**, sin embargo, este método no funciona ya que las instrucciones de los procesadores se pueden entrelazar de cualquier forma.
- La solución está en **utilizar instrucciones especiales hardware**, si se dispone de ellas, o en introducir soluciones software como las vistas anteriormente, que ya indicamos, que servían tanto para sistemas uniprocesador como para sistemas multiprocesador.

# Semáforos

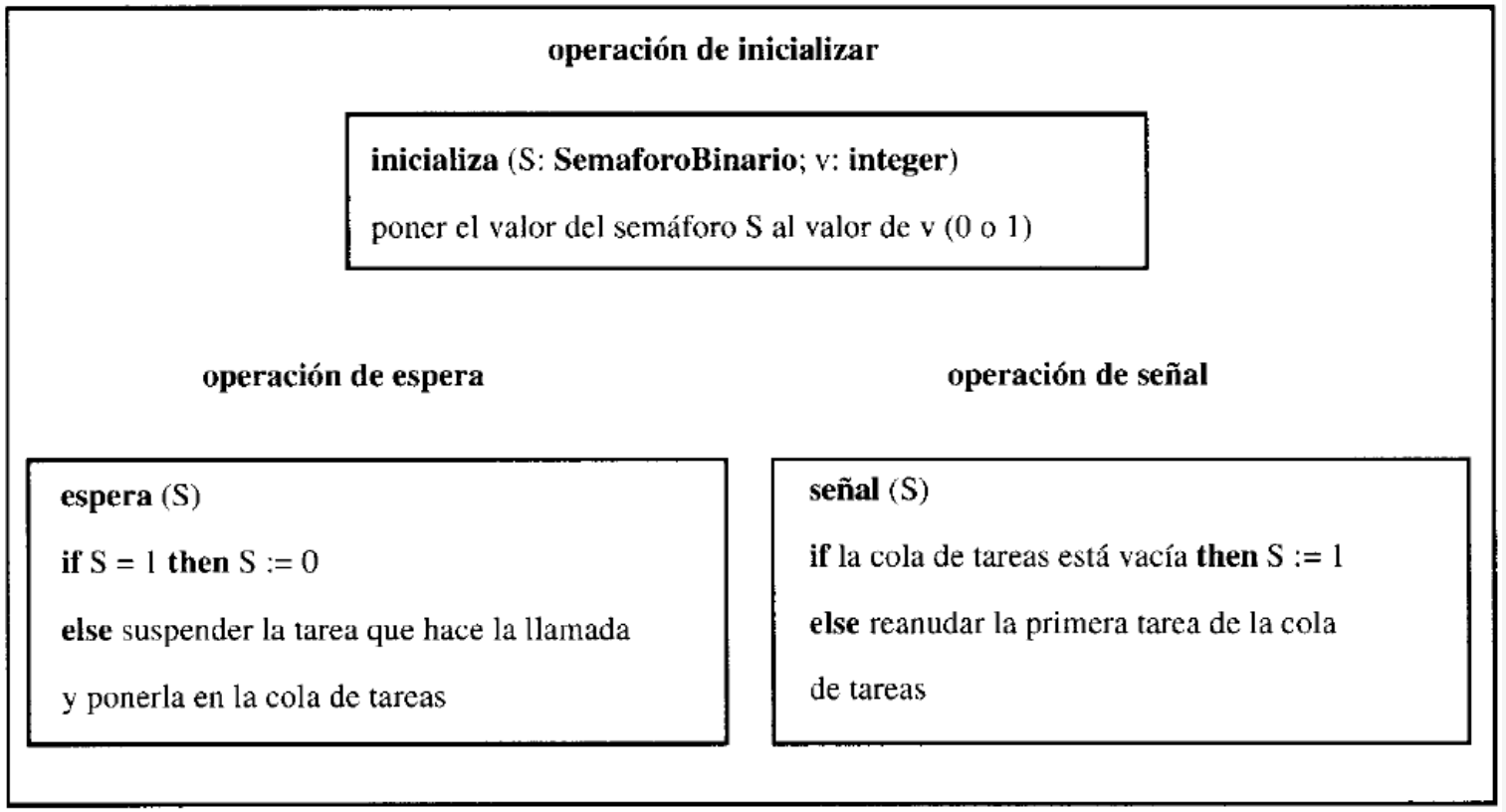



**Figura 3.3:** Operaciones primitivas sobre un semáforo

# Semáforos

- Se ejecutan antes de comience la ejecución concurrente de los procesos.
- Da el valor inicial al semáforo

**operación de inicializar**

**inicializa** (S: **SemaforoBinario**; v: **integer**)

poner el valor del semáforo S al valor de v (0 o 1)

**operación de espera**

**espera** (S)

**if** S = 1 **then** S := 0

**else** suspender la tarea que hace la llamada

y ponerla en la cola de tareas

**operación de señal**

**señal** (S)

**if** la cola de tareas está vacía **then** S := 1

**else** reanudar la primera tarea de la cola

de tareas

**Figura 3.3:** Operaciones primitivas sobre un semáforo

# Semáforos

- Si el semáforo es 1 lo cambia a 0 y continua su ejecución.
- Si el semáforo es 0 el proceso queda en estado de *espera* hasta que se libera el semáforo.

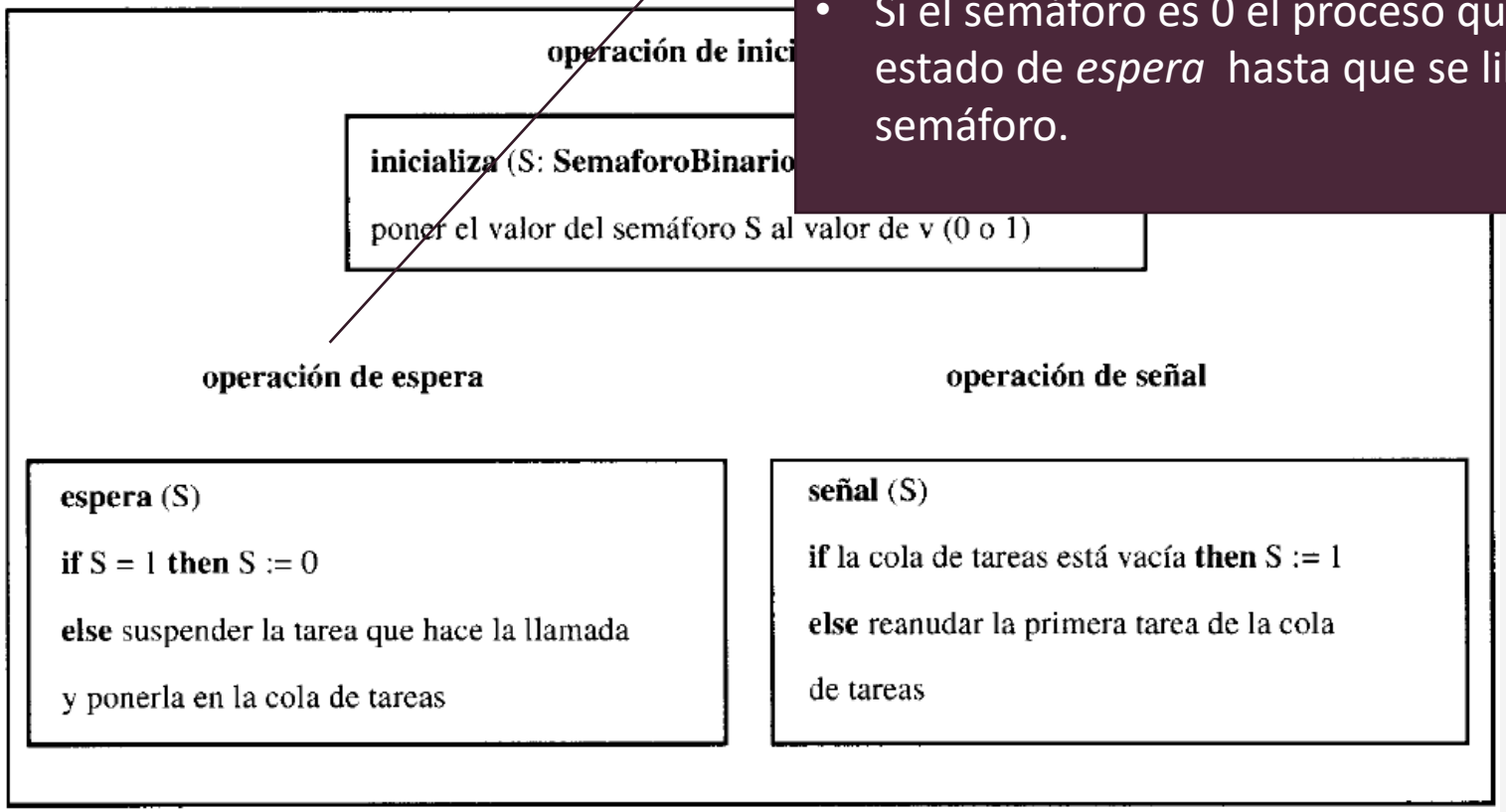



**Figura 3.3:** Operaciones primitivas sobre un semáforo

# Semáforos

- Cuando se ejecuta señal puede haber varios procesos en la lista/cola.
- El proceso que sigue depende del esquema de gestión de la cola de tareas suspendidas que haya sido implementado.
- Si no hay ningún proceso en espera del semáforo este se deja libre (S:= 1)

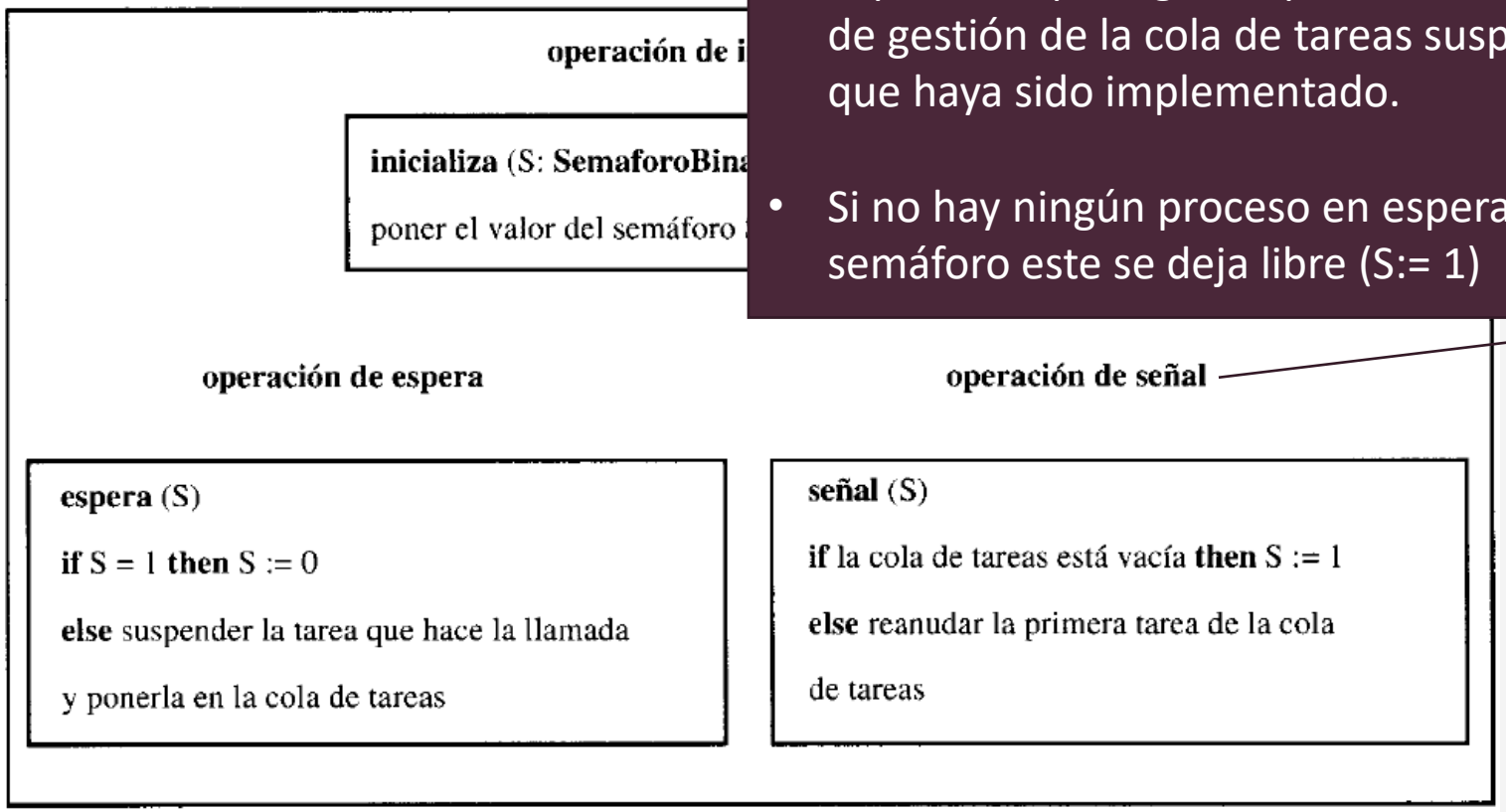
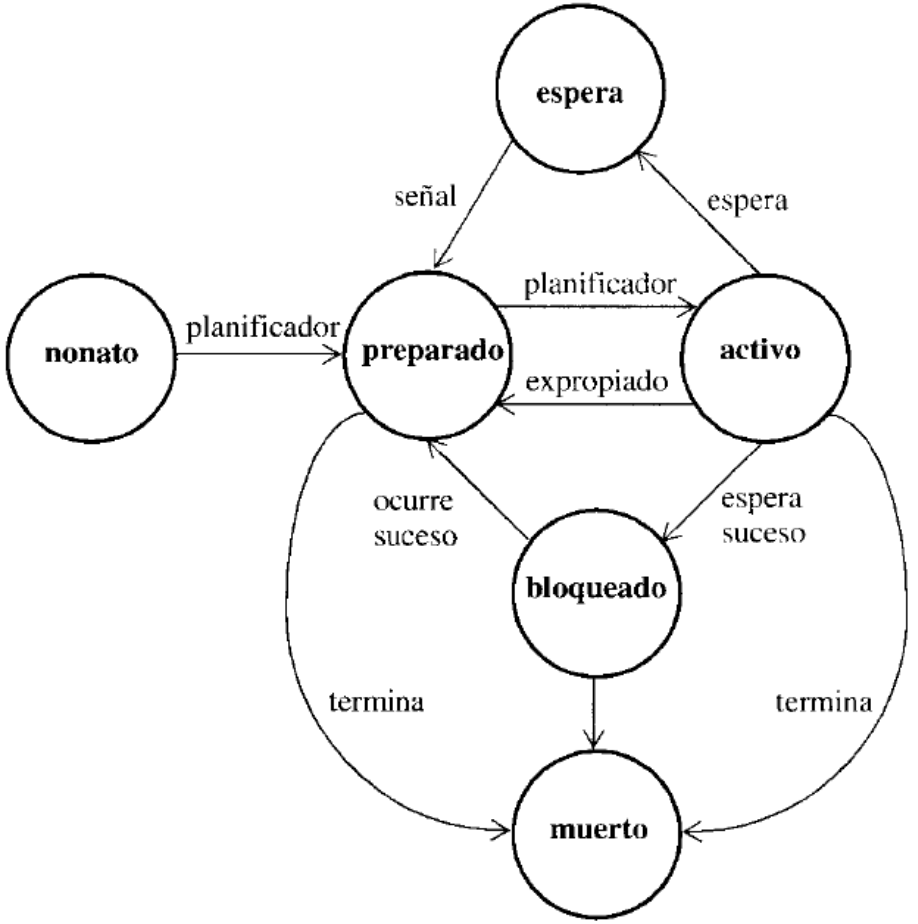



**Figura 3.3:** Operaciones primitivas sobre un semáforo

# Semáforos

**Transiciones para el estado de espera**



# Semáforos
# >> Exclusión mutua con semáforos

- La **operación de espera** se usará como procedimiento de bloqueo antes de acceder a una sección crítica.
- La **operación señal** como procedimiento de desbloqueo después de la sección.
- Se emplearán tantos semáforos como clases de secciones críticas se establezcan.

```
process P1
begin
        loop
        espera (S) ;
        /*Sección Crítica*/
        señal (S) ;
        /* resto del proceso */
    end
end P1;
```

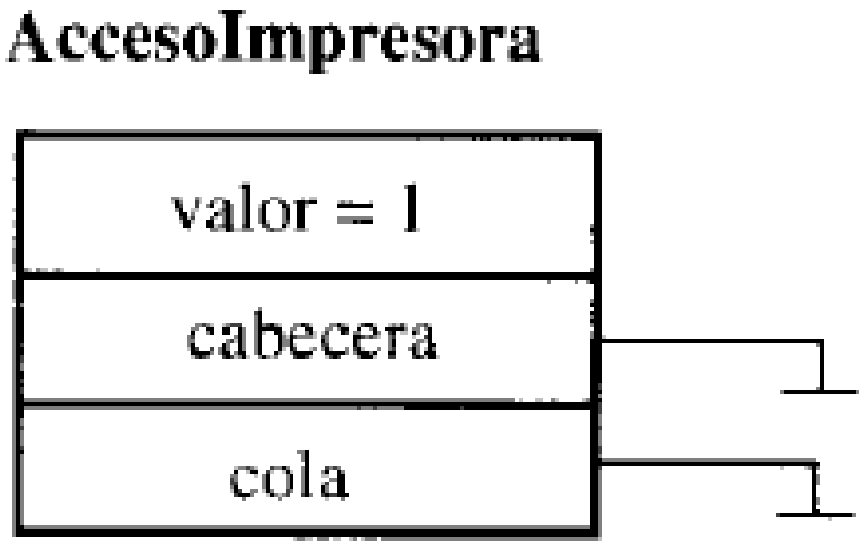
# Semáforos

- Ejemplo: Muestra un **esquema** del comportamiento de una **cola ligada a un semáforo** utilizado para compartir un recurso entre tres procesos concurrentes (P1, P2, P3).
- En la **implementación** de la cola se usa **una estrategia FIFO**.
- Se emplea el nombre **AccesoImpresora** para designar al semáforo.

# Semáforos

**1. Inicializa (AccesoImpresora, 1)**

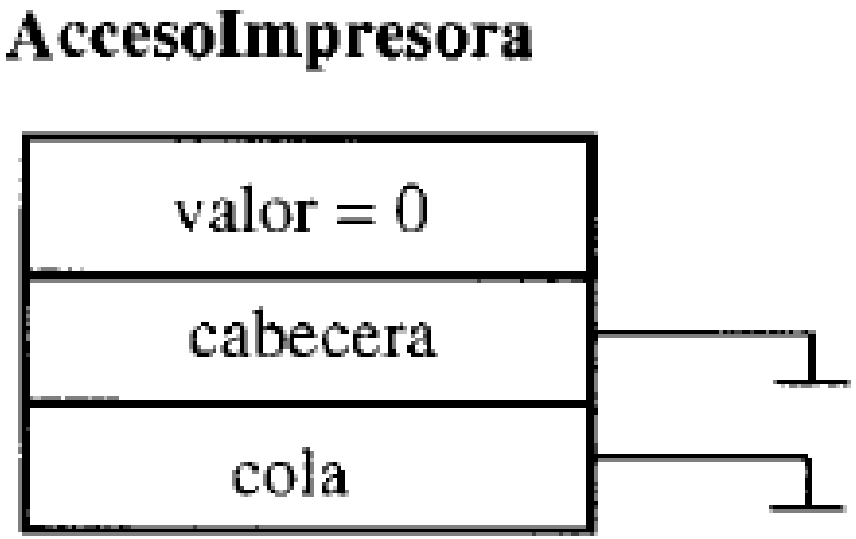
Se **inicializa** el semáforo **AccesoImpresora** con el **valor 1** de modo que deja **libre** el **acceso** a la **impresora**.



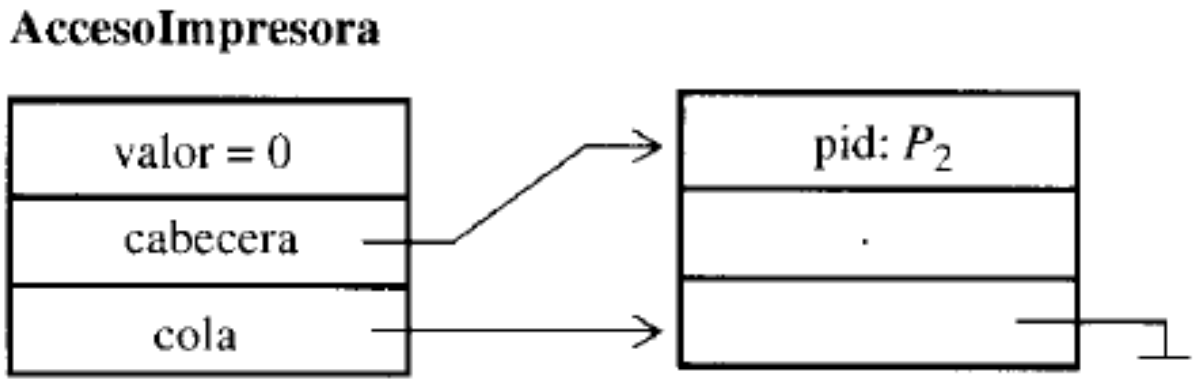
# Semáforos

**2. Proceso P1 ejecuta: espera (AccesoImpresora)**

El proceso **P1 toma la impresora** e impide el acceso de otro proceso al poner a 0 el valor del semáforo.



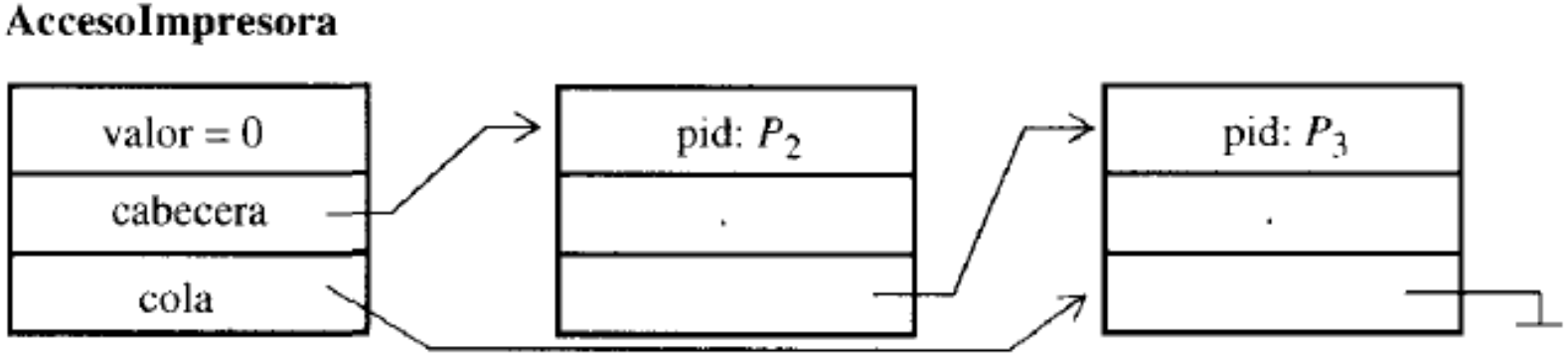
# Semáforos




**3. Proceso P1 bloqueado, Proceso P2 ejecuta: espera (AccesoImpresora)**

- El proceso P2 intenta acceder al recurso y se suspende, en espera de que éste quede libre, colocándose en la cola del semáforo.

# Semáforos

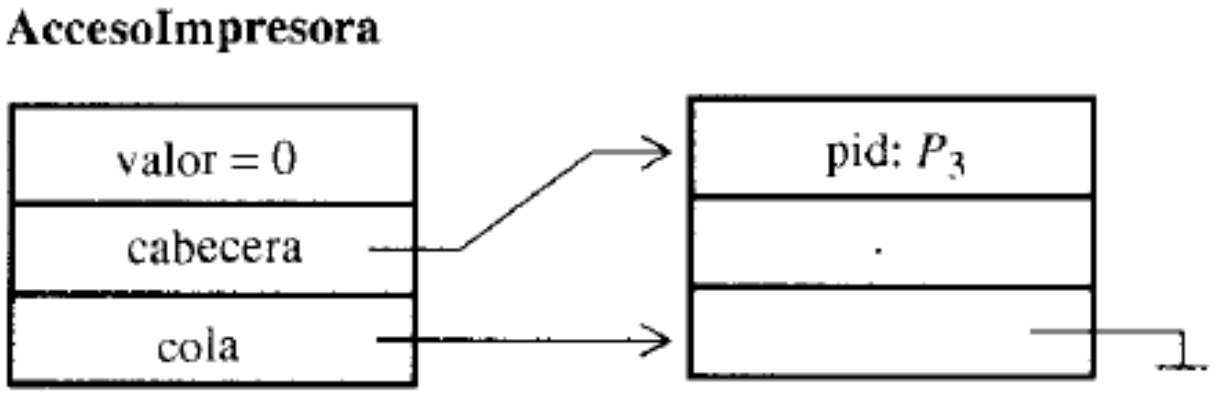



**4.**

**Proceso P3 ejecuta: espera (AccesoImpresora)**

- El proceso P3 intenta acceder al recurso, se suspende y se añade a la cola de espera.

# Semáforos

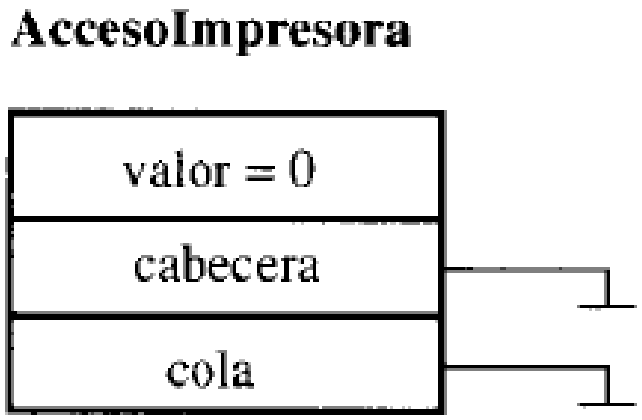



**5.**

**Proceso P1 ejecuta: señal (AccesoImpresora)**

- El proceso P1 deja el recurso; el proceso P2 pasa al estado preparado y tomará el recurso cuando pase a ejecución.

# Semáforos




**6.**

**Proceso P2 ejecuta: señal (AccesoImpresora)**

- El proceso P2 deja el recurso; el proceso P3 pasa al estado preparado.

# Gracias (>‿◠)✌