Sistemas Operativos

Comunicación y Sincronización - I

Sistemas Operativos

Sincronización

Sistemas Operativos

- ☑ Versión: Julio 2019
- ☑ Palabras Claves: Proceso, Comunicación, Mensajes, mailbox, port, send, receive, IPC, Productor, Consumidor

Algunas diapositivas han sido extraídas de las ofrecidas para docentes desde el libro de Stallings (Sistemas Operativos) , el de Silberschatz (Operating Systems Concepts)

Linux Kernel Development - 3er Edicion – Robert Love (Caps. 9 y 10)

Comunicación entre procesos

- ¿Como hacer para pasar información de un proceso a otro?
- ¿Cómo hacer para que no se "superpongan" entre sí?
- ¿Cómo obtener una secuencia apropiada de dependencias? (p.e., cuando uno produce y otro consume)

Comunicación entre Threads

- ☑ Facilidad por compartir el espacio de direcciones
  - ✓ Hilos de un mismo proceso
- ☑ Los problemas de superposición y dependencias siguen ocurriendo.

¿Para qué sirven los procesos cooperativos?

- ☑ Para compartir información (por ejemplo, un archivo)
- ☑ Para acelerar el cómputo (separar una tarea en subtareas que cooperan ejecutándose paralelamente)
- ☑ Para planificar tareas de manera tal que se puedan ejecutar en paralelo.

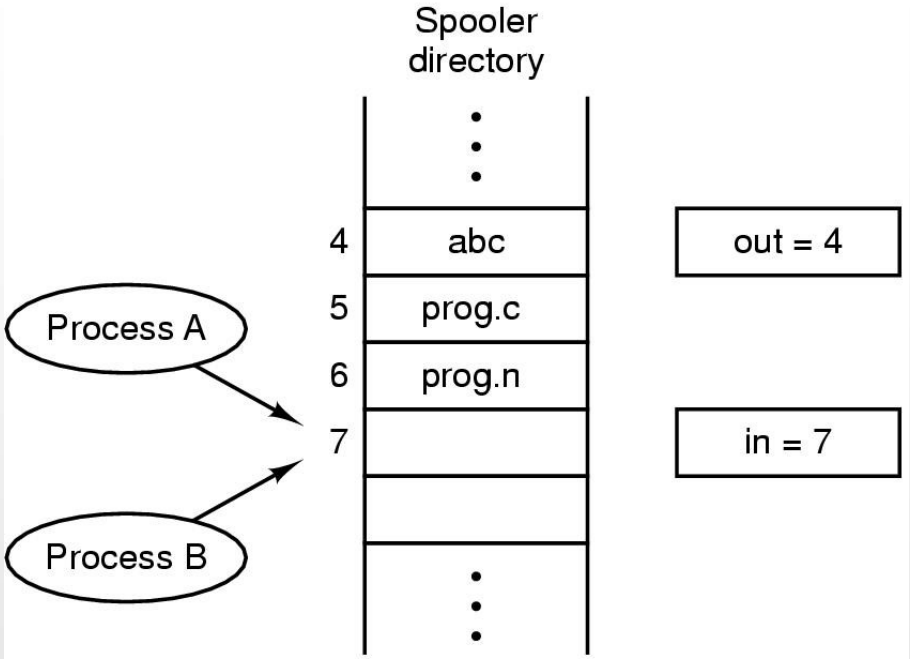
Dificultades de la Concurrencia

- **Compartir recursos globales**: Si dos procesos hacen uso de una variable compartida, el orden de acceso al recurso es crítico
- **Gestión de la asignación óptima de recursos**: Puede dar lugar a bloqueos mutuos (*interbloqueo*)
- **Dificultad de localizar errores de programación**: Los resultados no son ni deterministas ni reproducibles

Condición de carrera

- El resultado final depende del orden en que se ejecuten los procesos.
- Ejemplo:
  - P3 y P4 comparten la variable b y c.
  - Están inicializadas b=1, c=2
  - P3 ejecuta b=b+c
  - P4 ejecuta c=b+c
  - El valor final depende del orden de ejecución

Condición de carrera



- in: apunta a la siguiente ranura libre
- out: apunta al siguiente archivo a imprimir
- Ranuras 0 a 3, vacías (ya Se imprimieron)
- Ranuras 4 a 6, con archivos a imprimir

1. A lee in (7) y deja el procesador
2. B se ejecuta y lee in (7)
3. B almacena en el lugar 7 el nombre del archivo a imprimir.
4. Cuando le toca a A, sobreescribe la ranura 7

Ejemplo en monoprocesador

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

- ☑ P1 invoca echo y se interrumpe cuando getchar devuelve el valor (p.e., x). Cent se modificó (vale x)
- ☑ P2 invoca echo. Este se ejecuta hasta concluir y muestra el carácter y.
- ☑ P1 se reactiva. Cent ya no tiene el valor x (contiene y). En el *putchar* se muestra y.

Ejemplo en multiprocesador

Proceso 1	Proceso 2
cent= getchar()	**..**
…	**cent= getchar()**
csal=cent	**csal=cent**
putchar(csal)	**…**
…	**putchar(csal)**

Implementación de soluciones

- ☑ Prohibir que más de un proceso lea y escriba datos compartidos al mismo tiempo
  - ✓ Exclusión mutua
- ☑ Delegar responsabilidad en los procesos (modo usuario) (dependen del desarrollo de las aplicaciones)
  - Herramientas del SO
  - Herramientas de los Lenguajes
- ☑ Diseñar el Kernel para garantizar la exclusión mutua (modo kernel)

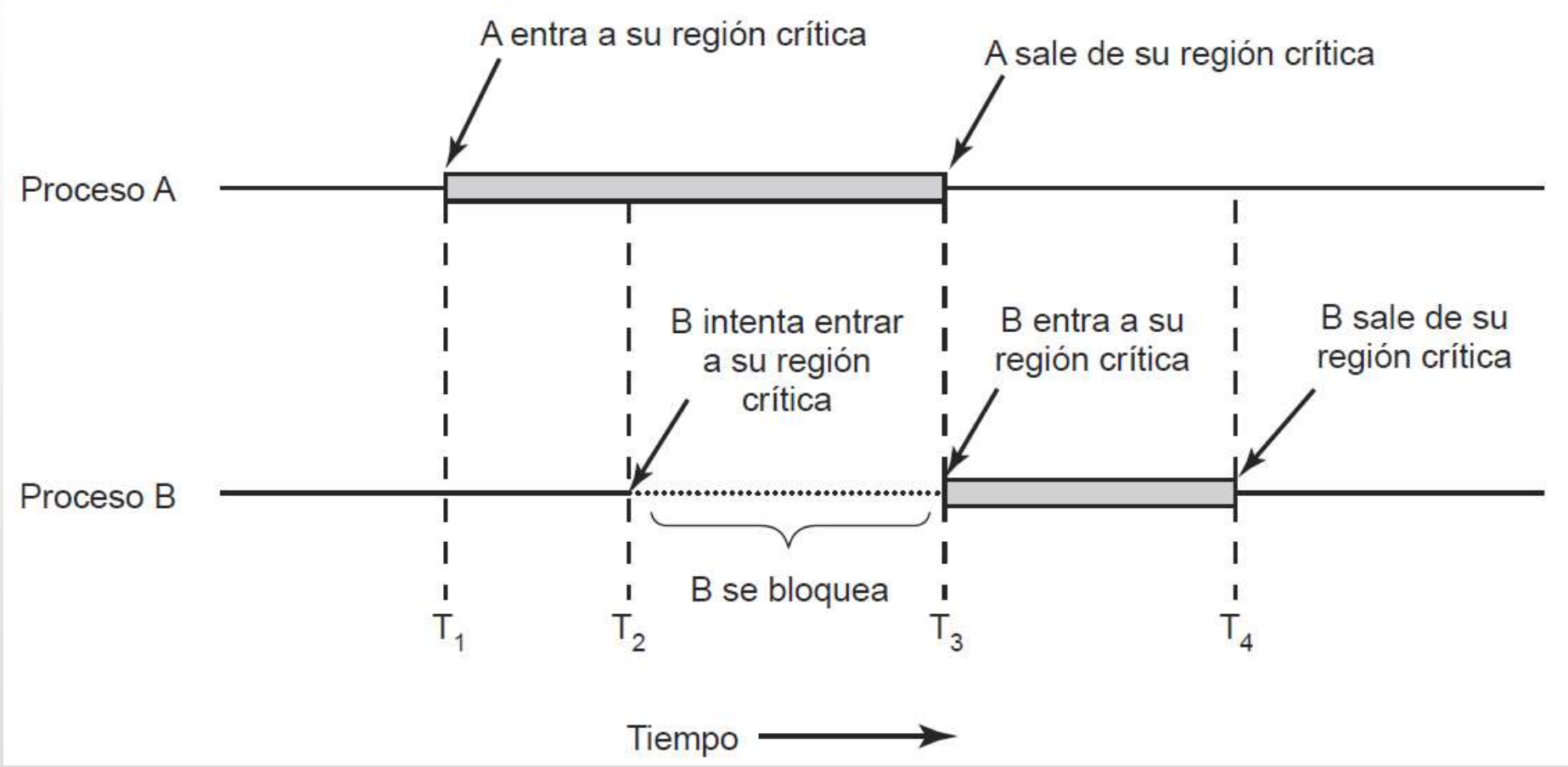
Concepto de sección crítica

- Sección de código en un proceso que accede a recursos compartidos y que no puede ser ejecutada mientras otro proceso esté en esa sección de código
  - Se protegen datos, no código
  - El SO también presenta secciones criticas.

Condiciones para garantizar la Exclusión Mutua

1. Dos procesos no pueden estar simultáneamente dentro de sus regiones críticas
2. No se pueden hacer suposiciones en cuanto a velocidades o cantidad de CPUs
3. Ningún proceso que se ejecute fuera de su SC puede bloquear otros procesos
4. Ningún proceso tiene que esperar "para siempre" para entrar en su SC.

Sección Critica



La solución a SC debe satisfacer

- ☑ **Exclusión mutua:** Si un proceso se está ejecutando en su sección crítica, entonces ningún otro proceso se puede estar ejecutando en la suya. *Recordemos que SC generalmente son líneas de código distintas y propias de cada proceso, pero que acceden a datos comunes.*
- ☑ **Continuidad o Progreso**: Si ningún proceso se está ejecutando en su sección crítica y hay otros que desean entrar, solo aquellos que no se están ejecutando en su sección restante pueden participar en la decisión de cual será el siguiente en entrar en la sección crítica, y esta decisión no puede postergarse indefinidamente.
- ☑ **Espera limitada**: Debe haber un límite en el número de veces que se permite que los demás procesos entren en su sección crítica después de que un proceso haya efectuado una solicitud para entrar en la suya y antes de que se conceda esa solicitud

Estructura General de un Proceso

Repeat

Entry section

Critical section

Exit section

Remainder section

Until false;

Posibles Soluciones

- ☑ Soluciones por Software:
  - ✓ Variables lock
  - ✓ Solución de Peterson
- ☑ Soluciones por Hardware:
  - ✓ Deshabilitar interrupciones
  - ✓ Instrucción TSL/Test and set lock
  - ✓ Instrucción xchg/swap

Espera activa, cíclica u ocupada

- ☑Busy waiting o spin waiting
- ☑El proceso no hace nada hasta obtener permiso para entrar en la SC.
- ☑Continúa ejecutando la/las instrucciones de la entrada a la SC.
- ☑Se usa cuando hay expectativa razonable que la espera será corta, ya que resulta menos costoso desperdiciar ciclos de CPU, que ocuparlos en un posible cambio de contexto

Variables candado (lock)

- Solución por software
- Variable compartida, con valor inicial 0
- Valores posibles: 0, ningún proceso está en la SC; 1, algún proceso está en la SC.
- El proceso lo fija en 1 al entrar a la SC y lo vuelve a 0 cuando sale de la SC.
- Problema: Posible condición de carrera:
  - El proceso, lee la variable con valor = 0
  - Antes de ponerla en 1 (lock), se le da la CPU al otro proceso
  - El otro proceso lee y la variable sigue en 0…

Ejemplo: Alternancia Estricta

Proceso 0

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

Proceso 1

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

- El problema es que ambos procesos manipulan la misma variable (turno) y la misma no está protegida
- La velocidad de ejecución de los procesos puede ser distinta y eso hace que un proceso pueda bloquear al otro
- El funcionamiento depende del orden en el que los procesos se ejecuten → Condición de carrera
- Si los procesos tienen distintos tiempos de ejecución, no es una buena solución
- Se viola la condición de que un proceso en una sección no critica bloquea a otro...

Ejemplo: Alternancia Estricta (cont.)

Proceso 0

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

Proceso 1

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

- La instrucción while (turno…..) implementa la espera activa. Permite determinar quien va a entrar a la sección crítica y quien debe esperar
- Inicialmente turno=0, por lo tanto P0 ejecuta su SC, mientras que P1 espera en forma activa
- Cuando P0 termina de ejecutar su SC, pone turno en 1 y le permite entrar a P1
- Supongamos que P1 ejecuta rápido su SC y sale del lock antes de que P0 termine su sección no critica.
- Si la ejecución de la región no critica de P0 tarda mucho, P1 no podrá volver a ejecutarse hasta que P0 termine y vuelva a desbloquear el candado
- Esta solución viola la condición 3: *"Ningún proceso que se ejecute fuera de su SC puede bloquear otros procesos"*, ya que P0 en una region no critica bloquea a P1

Solución de Peterson

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

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

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

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

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

Solución de Peterson

```
#define FALSE 0
#define TRUE 1
#define N   2

int turno;
int interesado[N];

void entrar_region(int proceso);
{
    int otro;

    otro = 1 – proceso;
    interesado[proceso] = TRUE;
    turno = proceso;
    while (turno == proceso && interesado[otro] == TRUE)
}

void salir_region(int proceso)
{
    interesado[proceso] = FALSE;
}
```

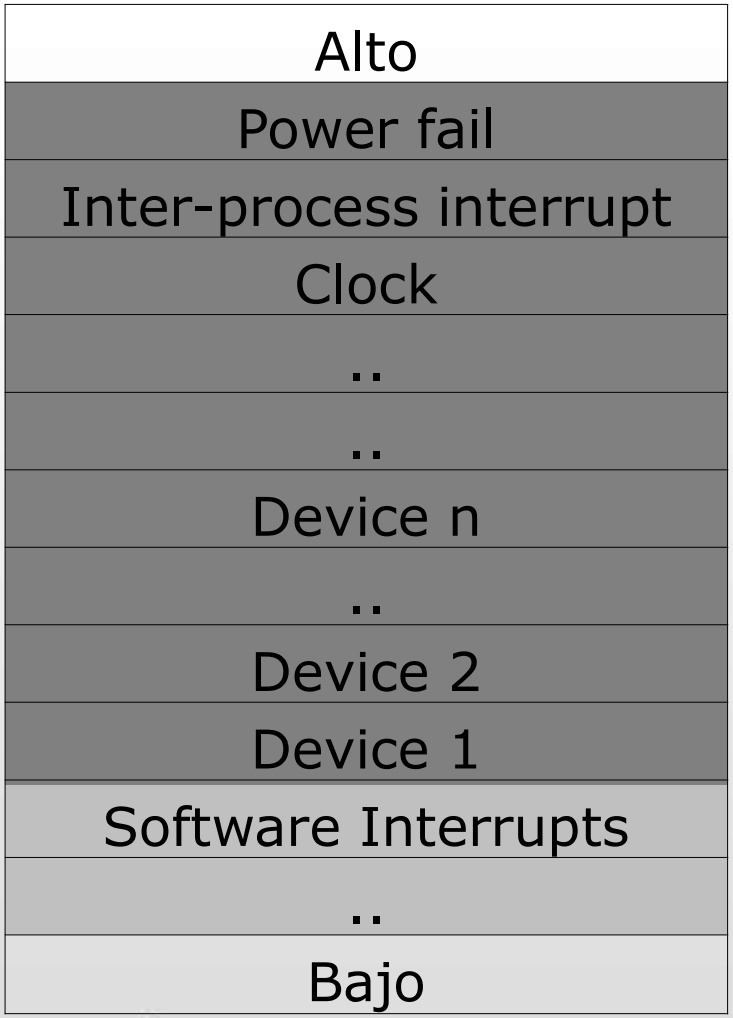
- Antes de entrar a la SC cada proceso llama a *entrar_region* con su propio número de proceso (0 o 1) como parámetro
- Si es necesario, cada proceso espera en esta llamada(while)
- Cuando se sale de la SC, el proceso invoca a *salir_region* para desbloquear el acceso a la SC
- **Soluciona el problema planteado anteriormente de requerir una alternancia estricta**

Esta línea garantiza la exclusión. Por mas que sea llamada al mismo tiempo por 2 procesos, solo se queda con el último resultado y le da acceso a uno u otro proceso

Deshabilitar interrupciones – Solución por Hardware

- ☑ Conocida también como "elevar el nivel de procesador"
- ☑ Sabemos que un proceso se ejecuta hasta que se invoca una System Call o es interrumpido
  - ✓ Lo que define esta técnica es que no pueden ocurrir interrupciones de reloj mientras se esté ejecutando en la SC → Podría ser adecuada en ambientes monoprocesador
- ☑ Multiprocesadores
  - ✓ Deshabilitar las interrupciones en un procesador no garantiza Exclusión Mutua
- ☑ Peligroso su uso por parte de procesos de usuario → no queremos que cualquier proceso pueda deshabilitarlas
- ☑ No es una técnica escalable

Deshabilitar interrupciones – Solución por Hardware




```
While (true)
   {
    /* deshabilitar interrupciones
    */;
    /* sección crítica */;
    /* habilitar interrupciones */;
    /* resto */;

    }
```

Instrucción TSL/Test and set Lock – Solución por Hardware

- ☑ Es una solución que requiere asistencia del Hardware
- ☑ Algunas computadoras (principalmente multiprocesadores) poseen una instrucción especial

TSL REGISTRO, CANDADO

- ☑ La instrucción lee el contenido de una variable ***CANDADO*** de la memoria y lo almacena en un registro. Luego almacena en ***CANDADO*** un valor distinto a cero.
- ☑ Si CANDADO = 0 entonces cualquier proceso puede acceder para bloquearla.
- ☑ Si CANDADO != 0, se hace una espera activa hasta que su valor sea 0
- ☑ Cuando el proceso termina de ejecutar la SC, vuelve a poner su valor en 0

```
entrar_region:
        TSL REGISTRO,CANDADO
        CMP REGISTRO,#0
        JNE entrar_region
        RET
```

```
salir_region:
        MOVE CANDADO,#0
        RET
```

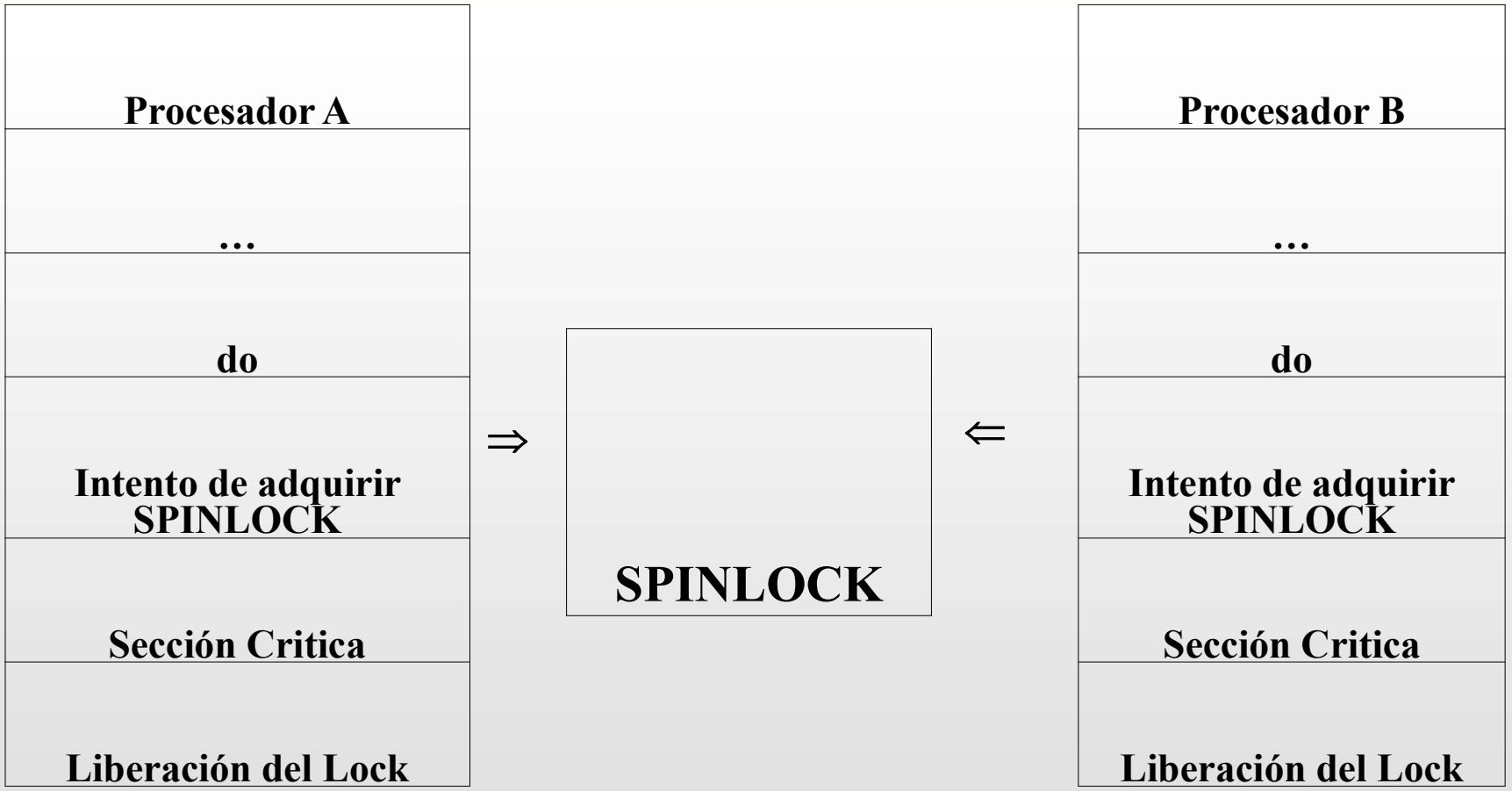
Instrucción TSL/Test and set Lock

- ☑ **Lo importante es garantizar que las operaciones leer y almacenar sean indivisibles**. Ninguna otra CPU deberá acceder a la memoria hasta que se ejecuten ambas operaciones.
- ☑ La CPU que ejecuta la instrucción TSL **bloquea el bus de memoria** para impedir que otras CPU accedan a la memoria hasta que esta termine.
- ☑ Al bloquear el acceso a la memoria se evita que otra CPU modifique los valores durante la operación. **Se garantiza atomicidad**
- ☑ Esta solución es mejor que la de deshabilitar las interrupciones, pero sigue dejando el control del lado de los procesos, lo cual es peligroso

Uso de instrucciones tipo lock

- Se "bloquea" el uso del bus multiprocesador para proteger la ubicación de memoria que se está accediendo.
- SPINLOCK: mecanismo que asocia una primitiva de locking a la estructura de datos que se quiere proteger
- El spinlock se asocia a la estructura a proteger
- Los procesos deben "adquirir" el spinlock

Spinlock



- ☑ Se puede implementar por test-and-set
- ☑ Cuando está tratando de adquirir el spinlock se eleva nivel de procesador.

Instrucción xchg/swap

- ☑ Presenta una solución al inconveniente de atomicidad planteado sobre las operaciones test & set.
- ☑ En vez de leer un valor y sobreescribirlo, la operación xchg intercambia el contenido de 2 ubicaciones de memoria en forma atómica.

```
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
```