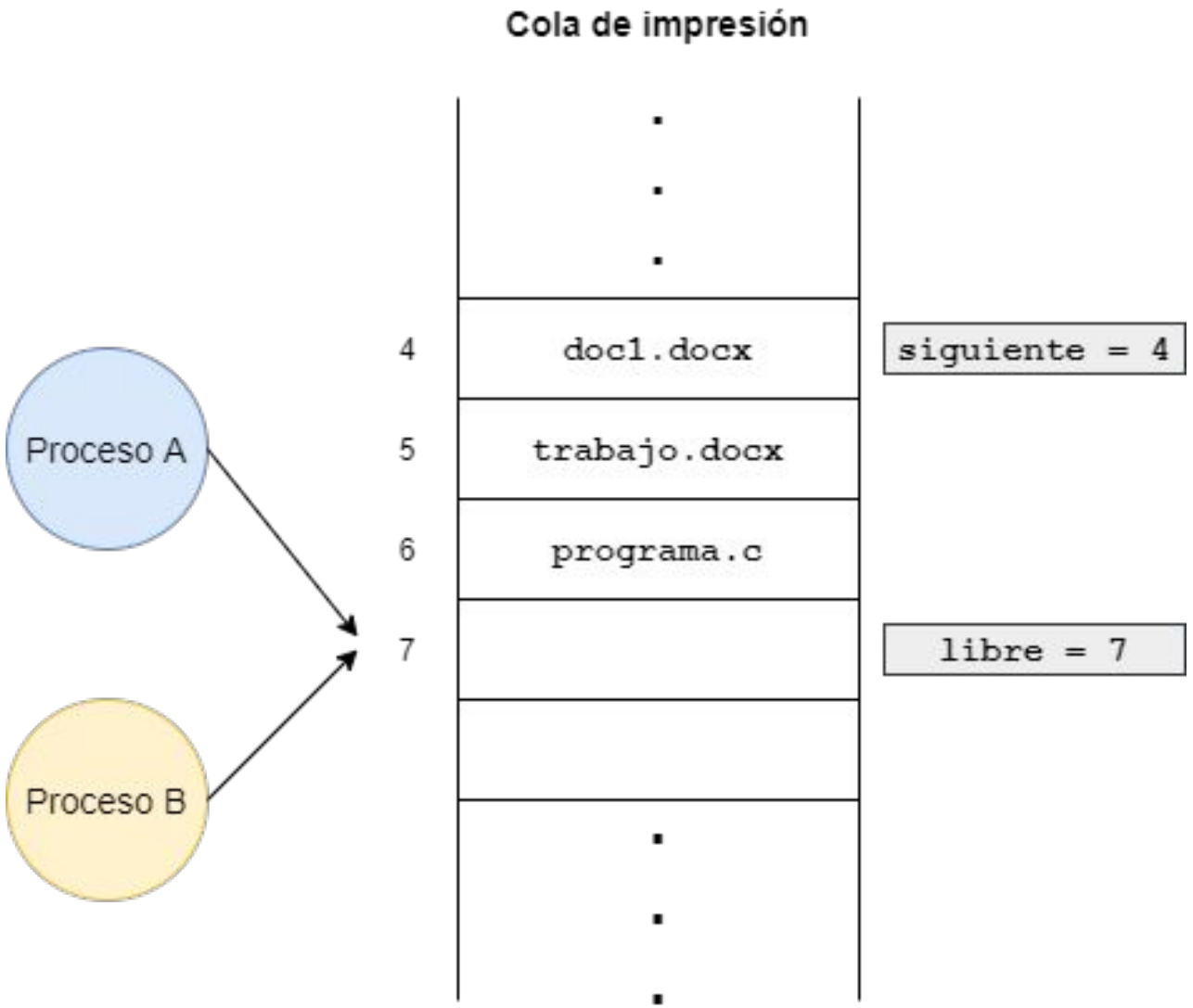
Problemas y soluciones en la sincronización de procesos

Adaptación (ver referencias al final)

Problemas de condiciones de carrera

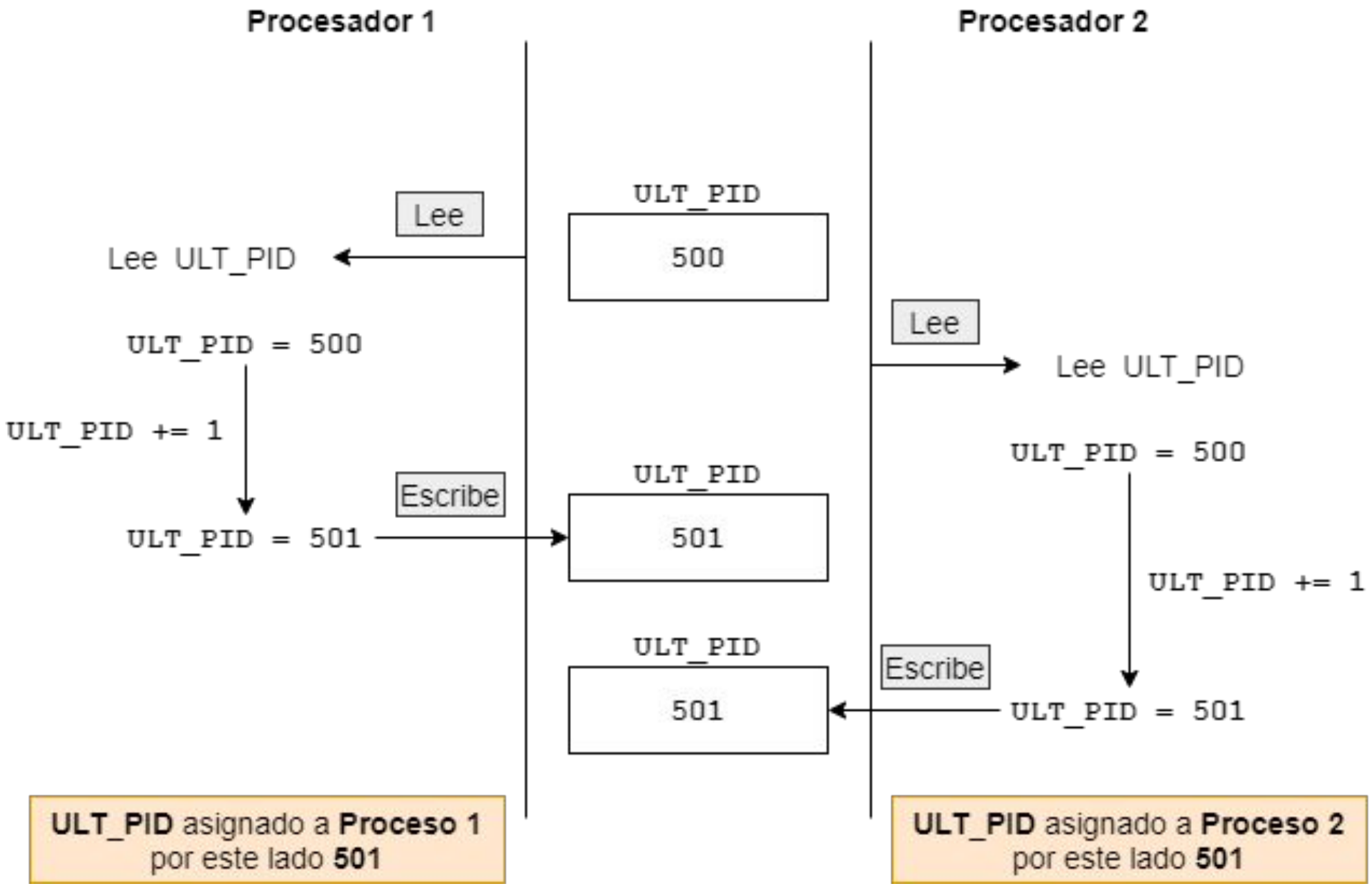
- Dos o más procesos están leyendo/escribiendo datos compartidos.
- Resultado final depende de quién se ejecuta y cuándo lo hace.



- Cola de impresión.
- Área compartida por procesos.
- `siguiente` y `libre` son variables compartidas
- **Proceso A** y **Proceso B** necesitan poner en slot apuntado por `libre` el trabajo a imprimir.

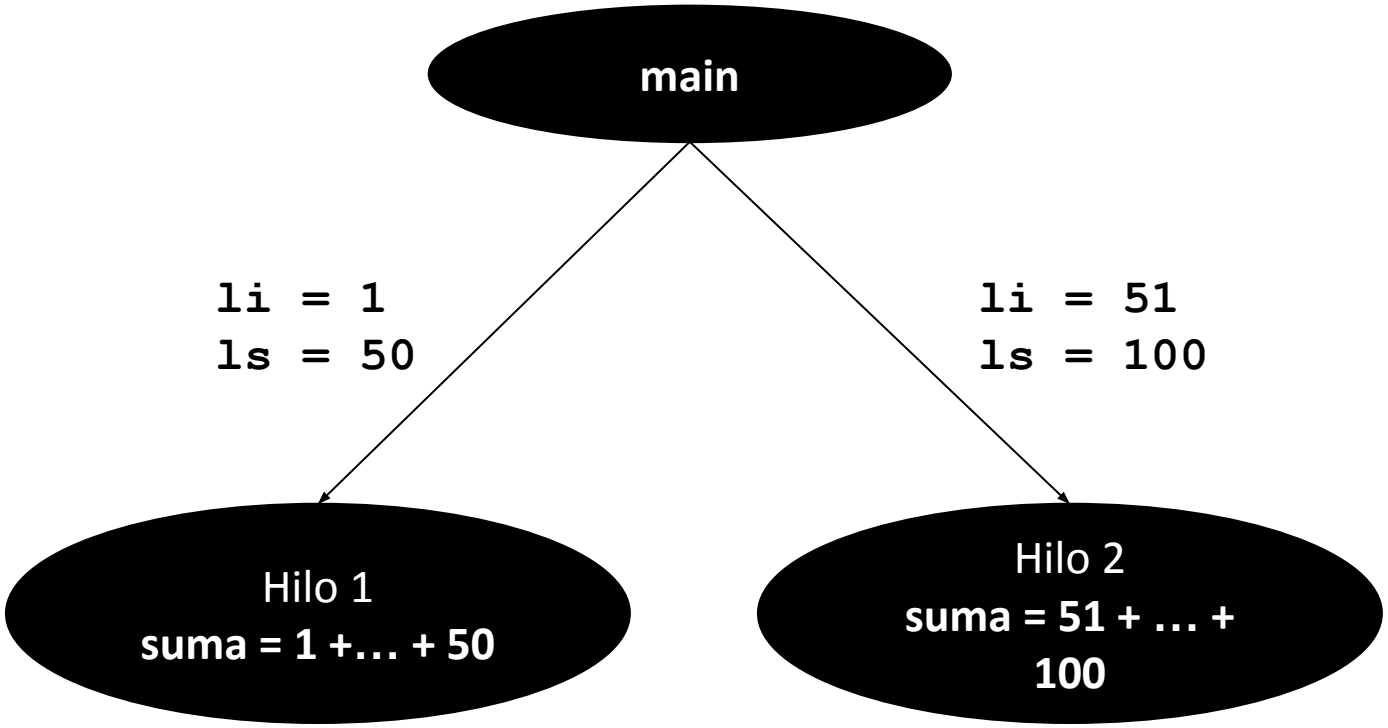
t_1 = **Proceso A** lee `libre = 7`
t_2 = **Proceso A** se interrumpe (por reloj)
t_3 = **Proceso B** lee `libre = 7`
t_4 = **Proceso B** mete en **7** trabajo a imprimir
t_5 = **Proceso A** se desbloquea
t_6 = **Proceso A** mete en **7** trabajo a imprimir y elimina trabajo de **proceso B**
t_8 = **Proceso A** actualiza `libre = 8`
t_9 = **Proceso B** muere esperando su impresión

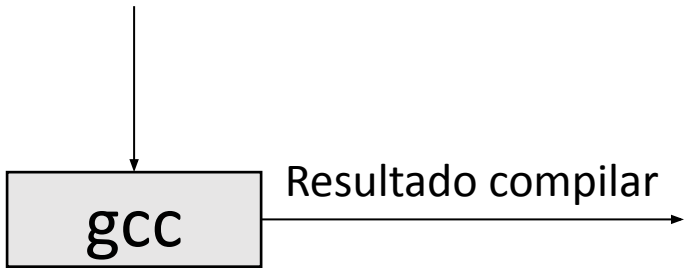
Condición de carrera


Procesador 1
Procesador 2
ULT_PID
500
Lee
Lee ULT_PID
ULT_PID = 500
ULT_PID += 1
ULT_PID = 501
Escribe
ULT_PID
501
Lee
Lee ULT_PID
ULT_PID = 500
ULT_PID += 1
ULT_PID
501
Escribe
ULT_PID = 501
ULT_PID asignado a Proceso 1 por este lado 501
ULT_PID asignado a Proceso 2 por este lado 501

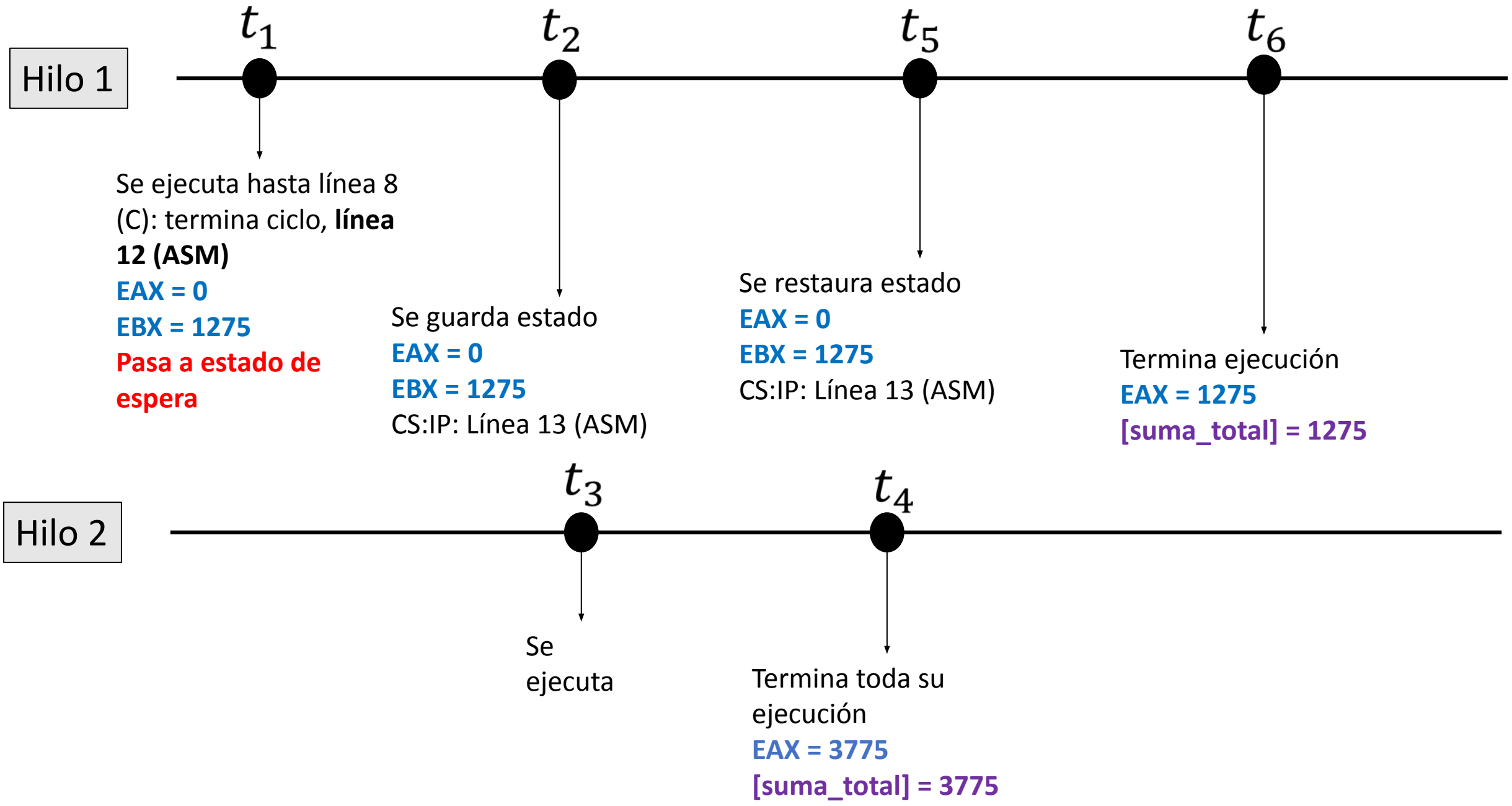
```
1: /* Variable global */
2: int sumatotal = 0;

3: /* Código de cada hilo */
4: void sumar(int li, int ls) {
5:  int j;
6:  int suma = 0;
7:  for (j = li; j <= ls; j++)
8:      suma = suma + j
9:  sumatotal = sumatotal + suma;
10:     pthread_exit(0);
11:}
```






```
7: xor eax, eax              ;eax = 0
6: mov [sumatotal], eax      ;suma_total = 0
...                          ;otras instrucciones
11: mov eax, [sumatotal]     ;Línea 9 en código C
12: mov ebx, [suma]          ;Línea 9 en código C
13: add eax, ebx             ;Línea 9 en código C
44: mov [sumatotal], eax     ;Línea 9 en código C
```


Hilo 1
t_1
t_2
t_5
t_6
Se ejecuta hasta línea 8 (C): termina ciclo, **línea 12 (ASM)**
EAX = 0
EBX = 1275
Pasa a estado de espera
Se guarda estado
EAX = 0
EBX = 1275
CS:IP: Línea 13 (ASM)
Se restaura estado
EAX = 0
EBX = 1275
CS:IP: Línea 13 (ASM)
Termina ejecución
EAX = 1275
[suma_total] = 1275
Hilo 2
t_3
t_4
Se ejecuta
Termina toda su ejecución
EAX = 3775
[suma_total] = 3775

```c
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

int sumatotal = 0;

void * sumar(void *arg) {
        int j;
        int suma = 0;
        int *limites = (int *) arg;

        int li = limites[0];
        int ls = limites[1];

        for (j = li; j <= ls; j++)
                suma = suma + j;
        sumatotal = sumatotal + suma;
        pthread_exit(0);
}
```

¿Qué sucede en un escenario multi procesador? ¿Qué sucede si se agrega un sleep(1) antes de actualizar sumatotal?

¿Qué sucede si solo se define un arreglo (`limites`) para ambos hilos?

```c
int main(int argc, char *argv[]) {
        pthread_t h1, h2;
        int limites_h1[2];
        int limites_h2[2];

        limites_h1[0] = 1;
        limites_h1[1] = 50;
        pthread_create(&h1, NULL, sumar, limites_h1);

        limites_h2[0] = 51;
        limites_h2[1] = 100;
        pthread_create(&h2, NULL, sumar, limites_h2);

        pthread_join(h1, NULL);
        pthread_join(h2, NULL);

        printf("sumatotal = %d\n",sumatotal);
        return 0;
}
```

Regiones | secciones críticas

- **Parte del programa** en la que se accede a memoria compartida.
- Procesos/hilos accediendo y modificando
  - Variables comunes
  - Registros de una BD
  - Archivos comunes
  - En general cualquier recurso compartido
- No puede existir **más de un proceso/hilo** ejecutando código de región crítica
- ¿Código de región crítica en ejemplo anterior?
  - `sumatotal = sumatotal + suma`

```
# ./hilos_sync_suma
li = 51
ls = 100
li = 1
ls = 50
sumatotal = 3775
```

```
# ./hilos_sync_suma
li = 1
ls = 50
li = 51
ls = 100
sumatotal = 5050
```

```
./hilos_sync_suma
li = 1
ls = 50
li = 51
ls = 100
sumatotal = 1275
```

- Diferentes resultados de ejecución
- Se agrega retardo de **un segundo** antes de actualizar código de región crítica
- Escenario multiprocesador
- En el código del hilo se hizo `printf()` de los límites recibidos en el hilo

Solución al problema de región crítica

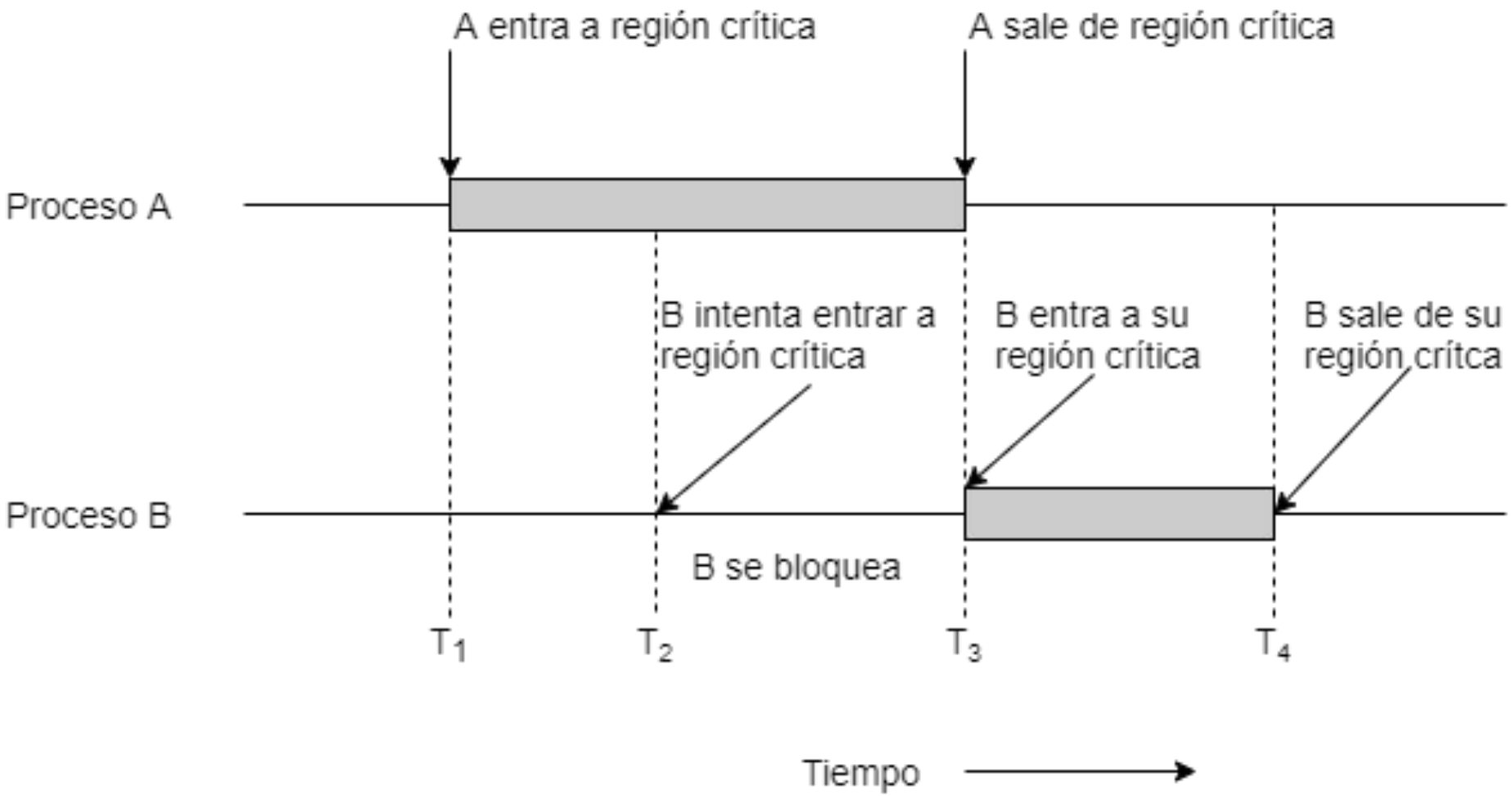
- Cada proceso solicita permiso para entrar en sección crítica
  - Fragmento de código que indica entrada a la sección crítica
- Cada proceso informa que sale de sección crítica
  - Fragmento de código que indica salida de la sección crítica
  - Otros procesos se enteran y pueden entrar en su sección crítica
- Estructura general del mecanismo de solución

```
Entrada en la sección crítica
Código de la sección crítica
Salida de la sección crítica
```

Solución al problema de región crítica

- Condiciones para **una buena solución** al problema de región crítica
  - No pueden existir dos procesos a la vez en sus regiones críticas: **exclusión mutua**.
  - No se pueden hacer suposiciones a cerca de las velocidades de CPU y número de CPUs.
  - Código por fuera de regiones críticas no bloquea a otros procesos.
  - Proceso no puede quedarse esperando para siempre para ejecutar su región crítica.
    - En la decisión solo participan procesos que deseen entrar en región crítica.
    - Decisión debe hacerse en tiempo finito.

Región crítica: solución deseada



Análisis de soluciones por software

- Revisión de soluciones por software al problema de región crítica

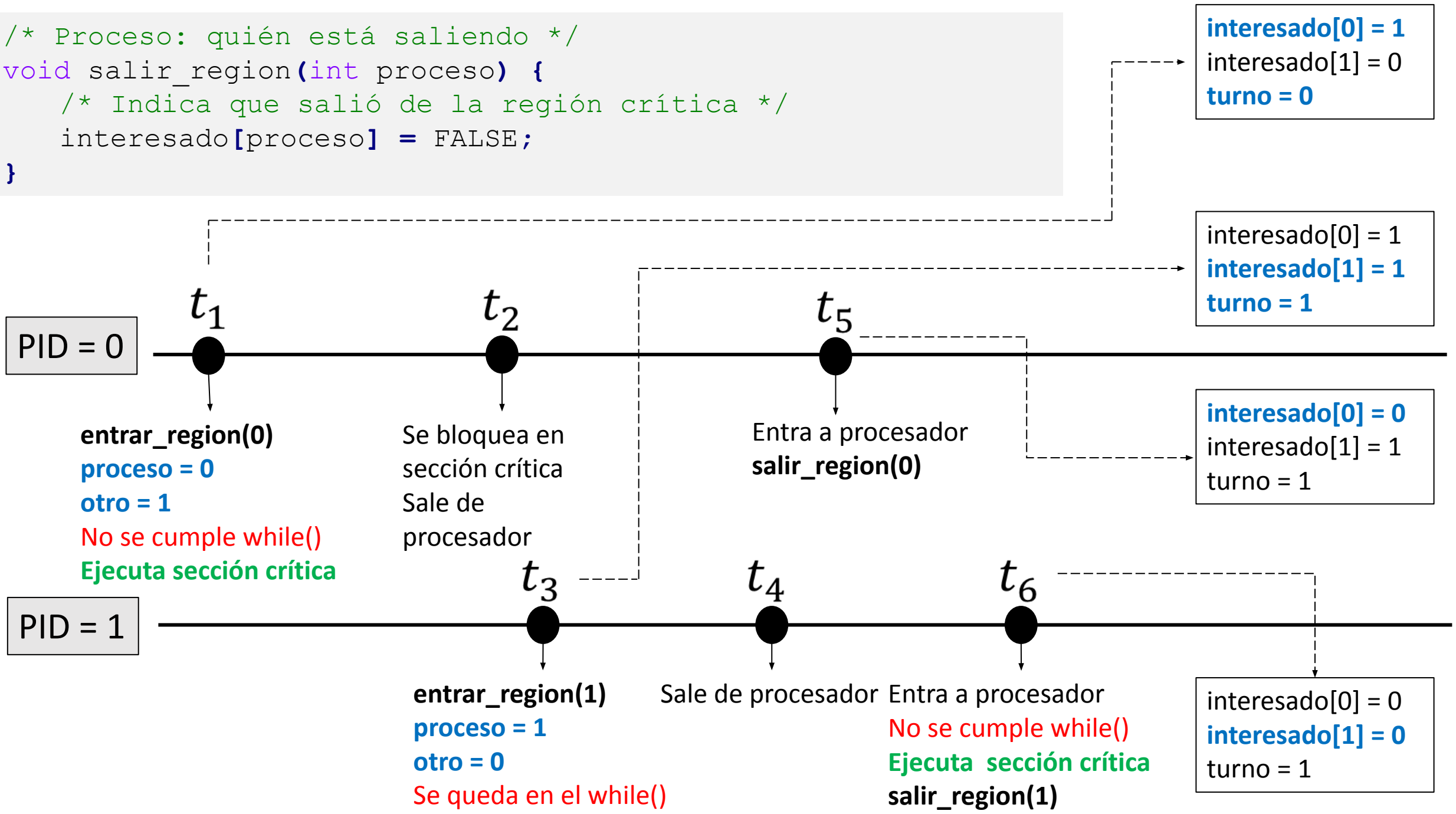
Solución de Peterson

- Restringida a dos procesos
  - PID = 0
  - PID = 1
- Alternan ejecución entre regiones críticas y regiones no críticas
- Se requiere que ambos procesos **compartan dos variables**
  - Variable entera **`turno`** en el código
  - Arreglo **`interesado[N]`** en el código
- Solución **por software** que **no** se garantiza dadas ciertas instrucciones en lenguaje ensamblador: LOAD, STORE

```c
#define FALSE 0
#define TRUE 1
/* Número de procesos */
#define N 2
/* ¿De quién es el turno? */
int turno;
/* Al principio todos los valores son 0 (FALSE) */
int interesado[N];
```

```c
/* Proceso es 0 o 1 */
void entrar_region(int proceso) {
    /* Número del otro proceso */
    int otro;
    /* El opuesto de este proceso */
    otro = 1 - proceso;
    /* Este proceso indica que está interesado */
    interesado[proceso] = TRUE;
    /* Establece bandera */
    turno = proceso;
    /* Instrucción nula */;
    while (turno == proceso && interesado[otro] == TRUE)
}
```

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



Soluciones de dormir y despertar

- Solución de Peterson y otras soluciones **desperdician tiempo de CPU**
- Procesos se ponen a consumir CPU en ciclos
  - **Espera ocupada** mientras esperan por entrada a región crítica.
- Existen primitivas que **bloquean/suspenden** al proceso en lugar de ponerlo en espera ocupada
- **sleep()**
  - Proceso que llama se bloquea hasta que otro lo despierte.
- **wakeup(pid)**
  - despierta el proceso que se indica como parámetro.

Problema del productor – consumidor

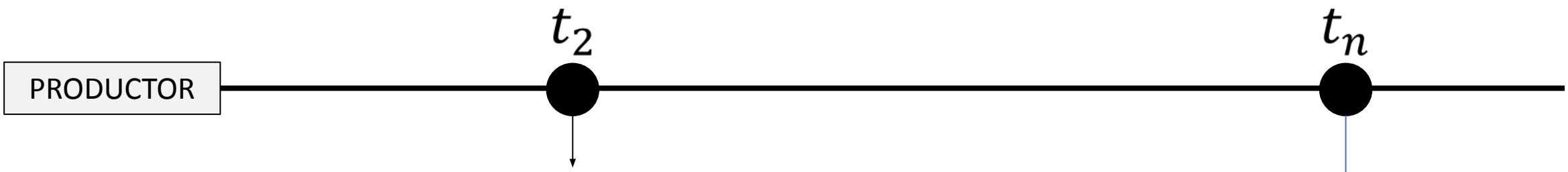
- Dos procesos comparten **búfer** común de **tamaño fijo**
- Un proceso coloca información en el búfer: **productor**
- Un proceso consumidor saca información del búfer: **consumidor**
- SI **productor** va a colocar elemento en búfer pero búfer está lleno.
  - Productor debe dormir hasta que productor lo despierte.
  - Consumidor despierta productor cuando haya quitado elementos del búfer.
- SI **consumidor** va a quitar elemento pero búfer vacío
  - Consumidor debe dormir hasta que productor lo despierte.
  - Productor despierta a consumidor cuando haya metido elementos en el búfer.

Productor con dormir y despertar

```
/* número de ranuras en el búfer */
#define N 100
/* número de elementos en el búfer */
int cuenta = 0;
void productor(void)
{
    int elemento;
    while (TRUE) {
        /* genera el siguiente elemento */
        elemento = producir_elemento();
        /* si el búfer está lleno, pasa a inactivo */
        if (cuenta == N)
            sleep();
        /* coloca elemento en búfer */
        insertar_elemento(elemento);
        /* incrementa cuenta de elementos en búfer */
        cuenta = cuenta + 1;
        if (cuenta == 1) /* ¿estaba vacío el búfer? */
            wakeup(consumidor);
    }
}
```

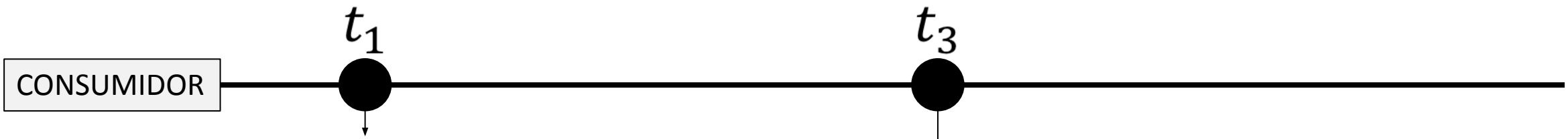
Consumidor con dormir y despertar

```
void consumidor(void)
{
    int elemento;
    * se repite en forma indefinida */
    while (TRUE) { /
        /* si búfer está vacío, pasa a inactivo */
        if (cuenta == 0)
            sleep();
        /* saca el elemento del búfer */
        elemento = quitar_elemento();
        /* disminuye cuenta de elementos en búfer */
        cuenta = cuenta - 1;
        /* ¿estaba lleno el búfer? */
        if (cuenta==N-1)
            wakeup(productor);
        /* consume el elemento del búfer */
        consumir_elemento(elemento);
    }
}
```



Mete un elemento en búfer
Incrementa cuenta
`if (cuenta == 1)`
Se envía **`wakeup(consumidor)`**
pero consumidor no está **realmente** dormido: señal se pierde.

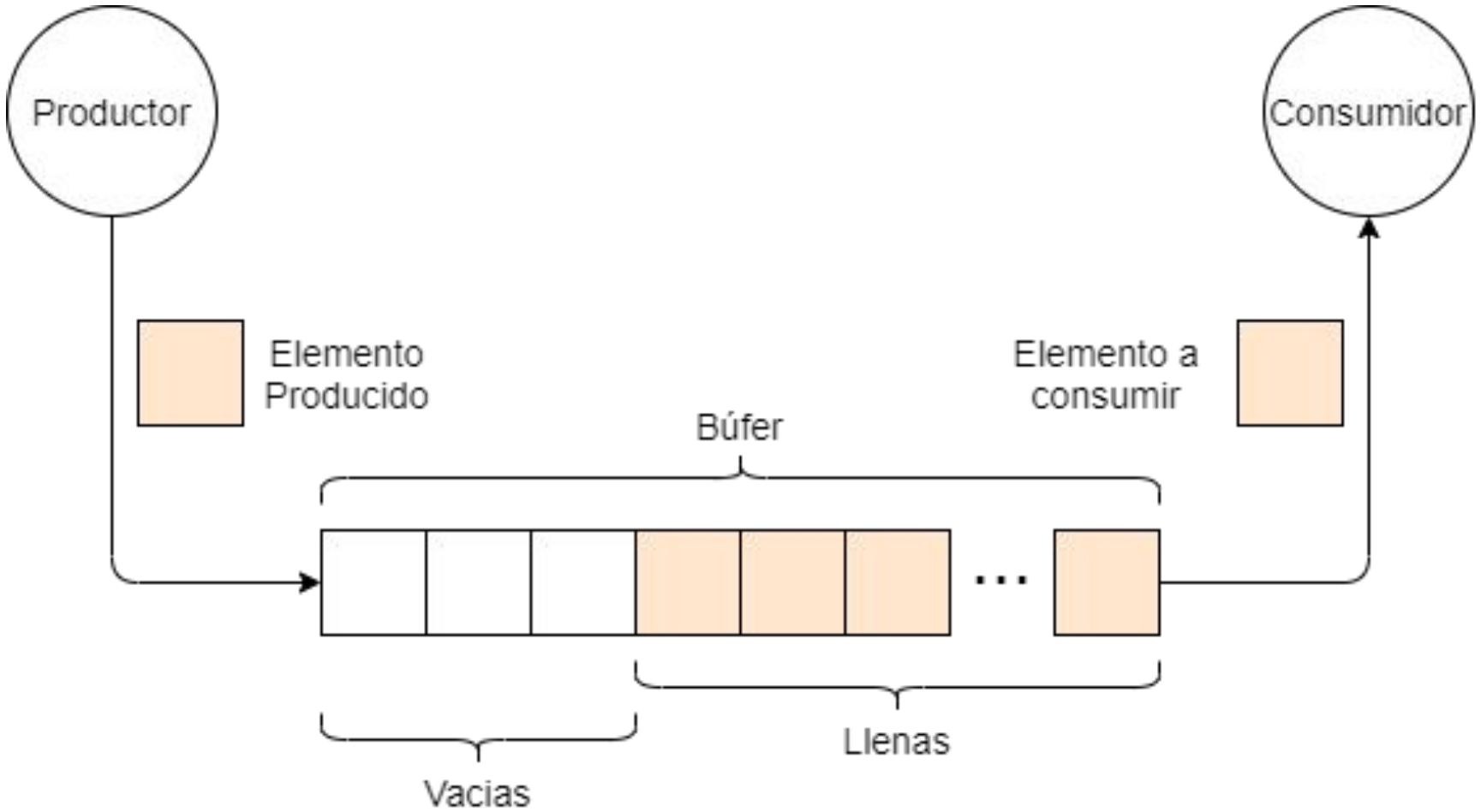
En algún momento llena el búfer y se duerme



Búfer vacío
Ejecuta validación variable cuenta
`if (cuenta == 0)`
Planificador lo saca de ejecución
Siguiente línea: **`sleep();`**

Se ejecuta y se duerme

Productor – consumidor



Problema del productor – consumidor

- ¿Qué elementos son compartidos por productor y consumidor?
  - El búfer
  - Número de ranuras vacías en el búfer
  - Número de ranuras llenas en el búfer
- ¿Cuál es la región crítica?
  - Cuando se ponen o se sacan elementos del búfer
  - Cuando se actualizan el número de ranuras vacías/llenas en el búfer

Semáforos

- Propuesto por **Edsger Dijkstra**
- Mecanismo de sincronización de procesos **disponible por el S.O**
- Un semáforo es un objeto con un valor entero
  - Se le puede asignar un valor inicial **NO** negativo
  - Solo se puede acceder al semáforo mediante dos operaciones **atómicas**: **wait** y **signal**.
- Operación atómica

Instrucción o conjunto de instrucciones que son tratadas como una sola operación que **NO puede ser interrumpida**

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

```
void wait(semaphore *s)
{
    s.count--;
    if (s.count < 0) {
    /* poner al proceso en la cola s.queue */;
    /* bloquear este proceso */;
    }
}
```

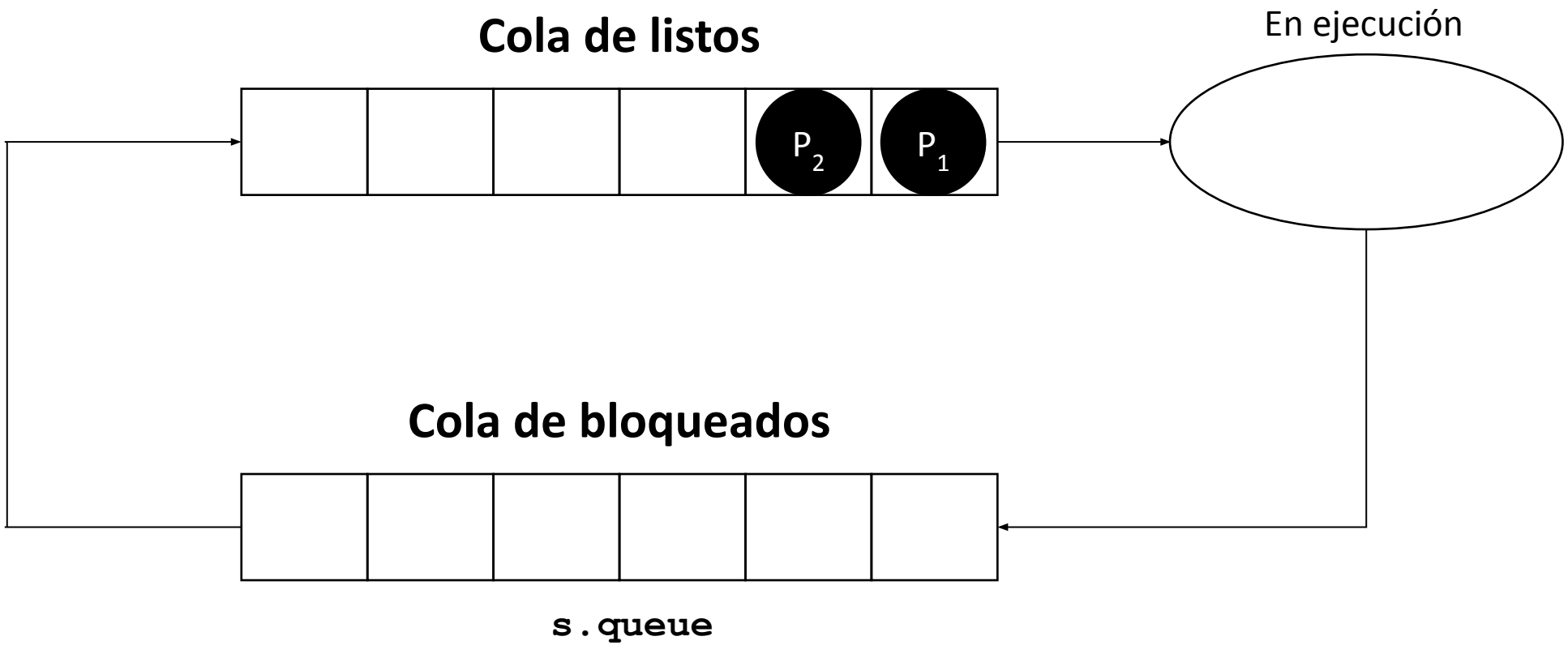
Otros nombres para esta función:

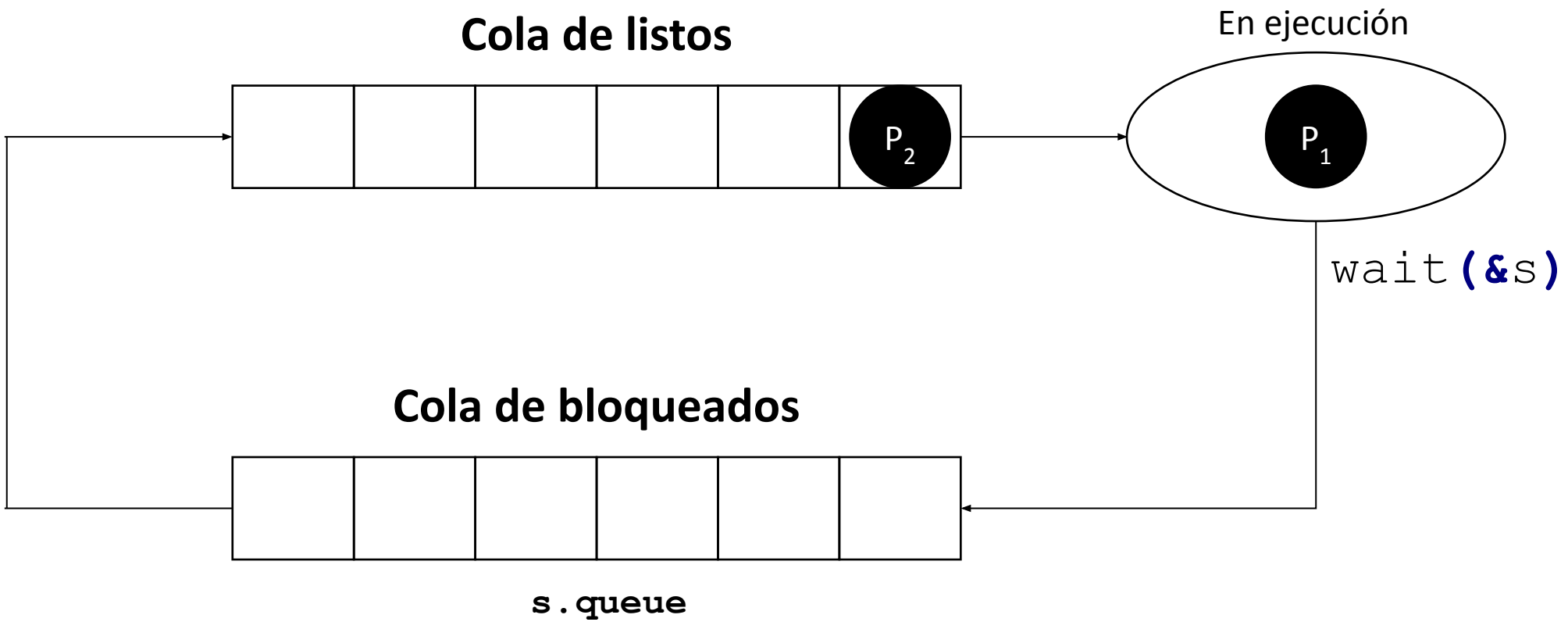
- `p(semaphore *s)`
- `down(semaphore *s)`

```
void signal(semaphore *s)
{
    s.count++;
    if (s.count <= 0) {
    /* sacar proceso de cola: s.queue */;
    /* poner proceso en cola de listo */;
    }
}
```

Otros nombres para esta función:

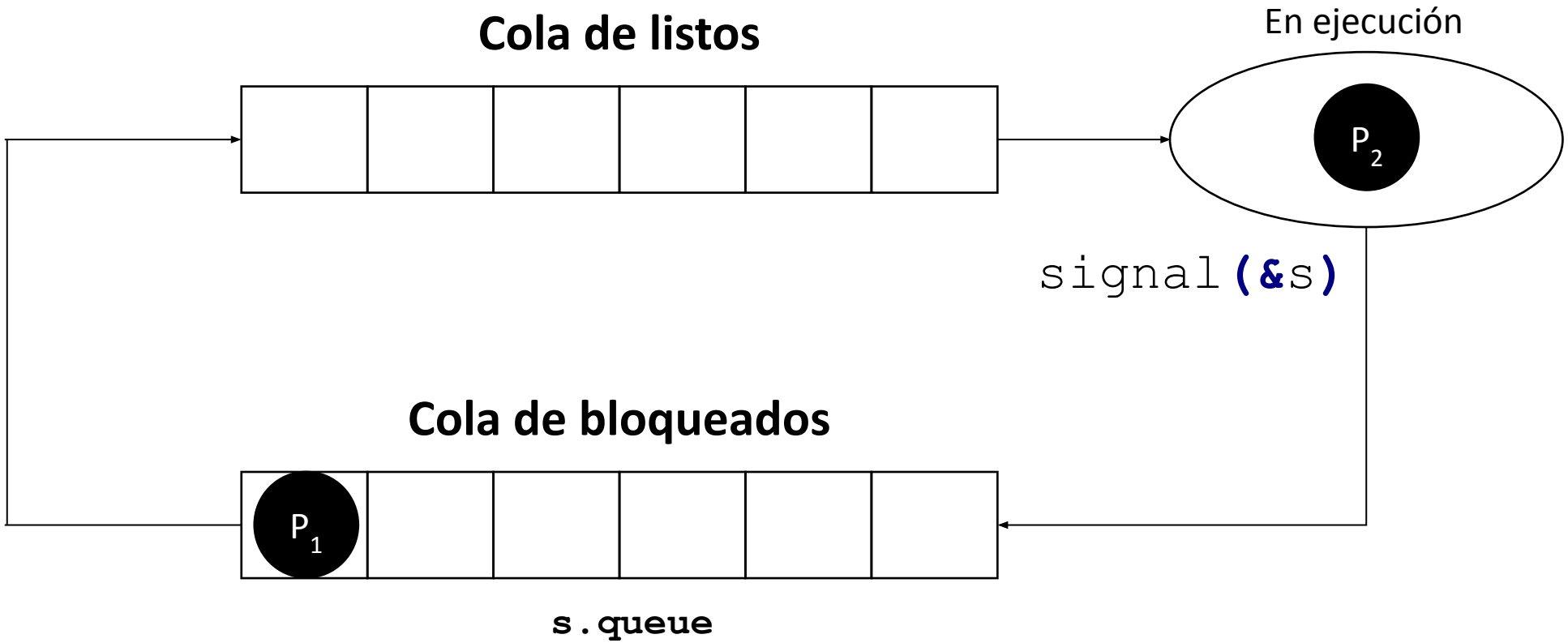
- `v(semaphore *s)`
- `up(semaphore *s)`


Cola de listos
En ejecución
P2
P1
Cola de bloqueados
s.queue



Llamada a `wait()` hace que el proceso se bloquee

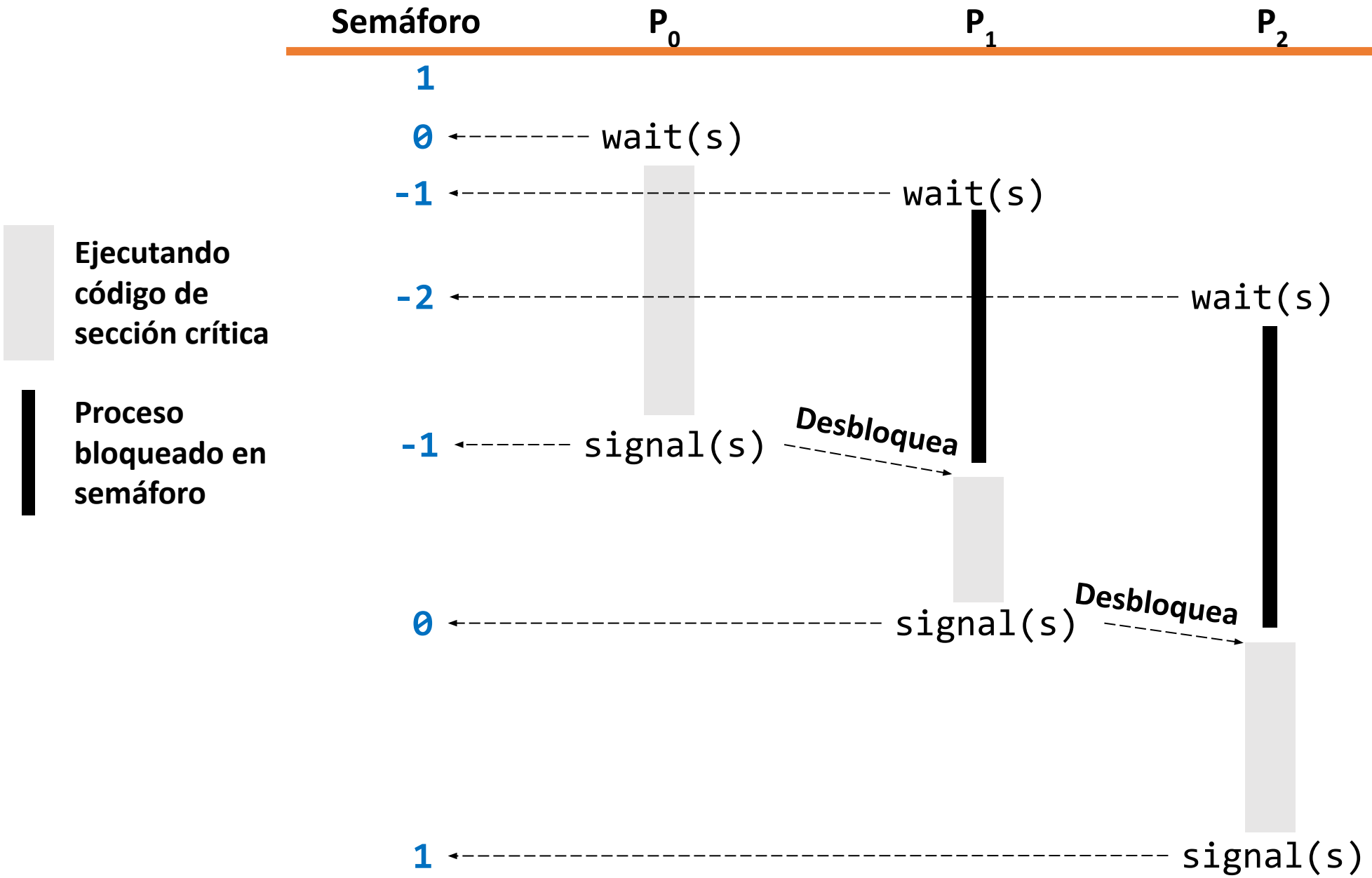
Llamada a `signal()` hace que el proceso bloqueado se desbloquee



Región crítica con semáforos

- El código de sección crítica se protege de la siguiente manera
- Semáforo inicial con valor en 1

```
wait(s)
Código de la sección crítica
signal(s)
```


Semáforo
P0
P1
P2
1
0
wait(s)
-1
wait(s)
-2
wait(s)
-1
signal(s)
Desbloquea
0
signal(s)
Desbloquea
1
signal(s)
Ejecutando código de sección crítica
Proceso bloqueado en semáforo

PRODUCTOR CON SEMÁFOROS

```
#define N 100                   /* número de ranuras en el búfer */
typedef int semaforo;           /* los semáforos son un tipo especial de int */
semaforo mutex = 1;             /* controla el acceso a la región crítica */
semaforo vacias = N;            /* cuenta las ranuras vacías del búfer */
semaforo llenas = 0;            /* cuenta las ranuras llenas del búfer */
void productor(void)
{
    int elemento;
    while(TRUE){                /* TRUE es la constante 1 */
        /* genera algo para colocar en el búfer */
        elemento = producir_elemento();
        wait(&vacias);          /* disminuye la cuenta de ranuras vacías */
        wait(&mutex);           /* entra a la región crítica */
        /* coloca el nuevo elemento en el búfer */
        insertar_elemento(elemento);
        signal(&mutex);         /* sale de la región crítica */
        signal(&llenas);        /* incrementa la cuenta de ranuras llenas */
    }
}
```

CONSUMIDOR CON SEMÁFOROS

```
void consumidor(void)
{
    int elemento;
    while(TRUE){          /* ciclo infinito */
        wait(&llenas);    /* disminuye la cuenta de ranuras llenas */
        wait(&mutex);     /* entra a la región crítica */
        /* saca el elemento del búfer */
        elemento = quitar_elemento();
        signal(&mutex);  /* sale de la región crítica */
        signal(&vacias);     /* incrementa la cuenta de ranuras vacías */
        /* hace algo con el elemento */
        consumir_elemento(elemento);
    }
}
```

Seguimiento paso a paso productor – consumidor con semáforos…