COMUNICACIÓN Y SINCRONIZACIÓN ENTRE PROCESOS

Rodrigo García Carmona
Universidad San Pablo-CEU
Escuela Politécnica Superior

OBJETIVOS

- Presentar los **motivos** que justifican la **comunicación** y **sincronización** de procesos.
- Presentar dos **alternativas** básicas para comunicar procesos:
  - Memoria compartida.
  - Mensajes.
- Analizar las **condiciones de carrera** (*race conditions*) y estudiar el concepto de **serializabilidad**.
- Estudiar para la memoria compartida el problema de la **sección crítica**.
- Presentar los criterios de corrección al problema de la sección crítica y sus posibles **soluciones** de forma estructurada:
  - Soluciones hardware (cerrojos).
  - Semáforos.
  - Construcciones lingüísticas (monitores).
- Adquirir destreza en la solución de problemas de sincronización a través de **problemas clásicos**:
  - Productores-consumidores, lectores-escritores, etc.

CONTENIDO

- Introducción
- Memoria compartida
- Sección crítica
- Soporte hardware
- Semáforos
- Monitores

Bibliografía

- W. Stallings: **Sistemas Operativos**.
  - Capítulo 5.
- A.S. Tanenbaum: **Modern Operating Systems**.
  - Capítulos 2, 11 y 12.

INTRODUCCIÓN

INTRODUCCIÓN

- Los procesos (y threads) **interactúan** en un sistema multiprogramado…
  - …para compartir recursos (como por ejemplo datos compartidos).
  - …para coordinar su ejecución.
- La **ejecución entrelazada** puede dar lugar a consecuencias inesperadas.
  - Es necesario establecer un mecanismo para restringir las posibles ejecuciones entrelazadas.
  - La planificación de procesadores es invisible para las aplicaciones.
- La **sincronización** es el mecanismo que nos proporciona ese control.

PROBLEMA DE EJEMPLO (I)

- Supongamos que escribimos una serie de funciones para gestionar los ingresos y las retiradas de efectivo de cuentas bancarias como:

```
Withdraw(acct, amt) {
  balance = get_balance(acct);
  balance = balance – amt;
  put_balance(acct,balance);
  return balance;
}
```

```
Deposit(acct, amt) {
  balance = get_balance(acct);
  balance = balance + amt;
  put_balance(acct,balance);
  return balance;
}
```

- Supongamos además que compartimos nuestra cuenta con otra persona y que el saldo es de 1000 €
- Ambos titulares de la cuenta vamos a dos cajeros diferentes para retirar 100€ uno de los titulares y para ingresar 100 € el otro titular.

PROBLEMA DE EJEMPLO (II)

- Podemos representar esta situación creando threads separados para cada acción que correrán en el servidor centralizado del banco:

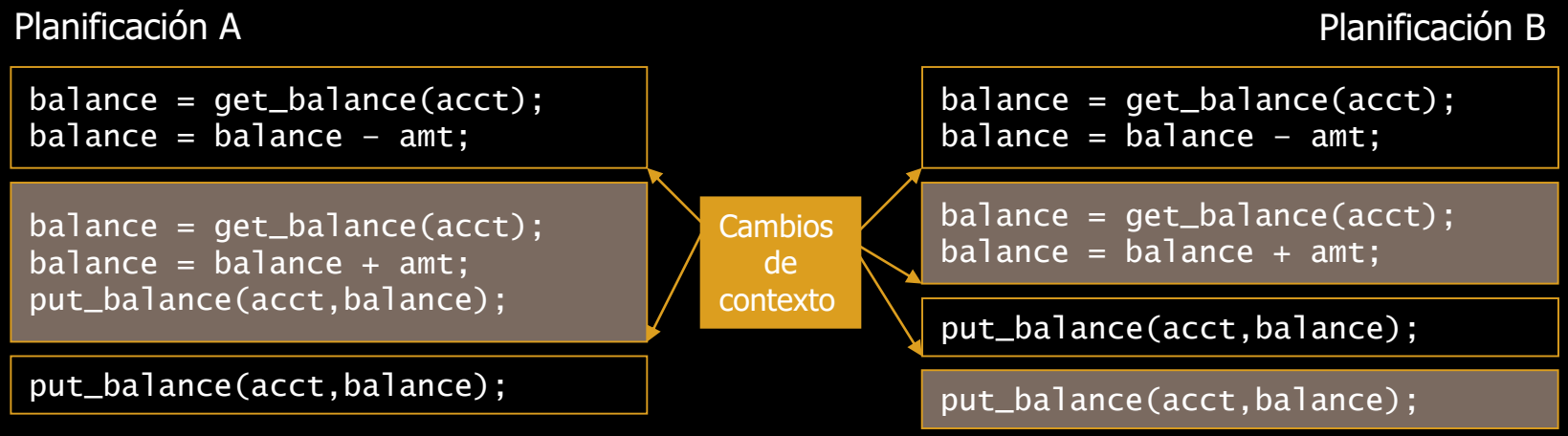
```
Withdraw(acct, amt) {
  balance = get_balance(acct);
  balance = balance – amt;
  put_balance(acct,balance);
  return balance;
}
```

```
Deposit(acct, amt) {
  balance = get_balance(acct);
  balance = balance + amt;
  put_balance(acct,balance);
  return balance;
}
```

- ¿Qué NO es correcto de esta implementación?.
  - Pensar en las posibles planificaciones que haga el sistema para estos dos threads . . .

PROBLEMA DE EJEMPLO (III)

- El problema es que la ejecución de los dos procesos puede entrelazarse:



- ¿Cuál es el saldo de la cuenta ahora?
- ¿Estará contento el banco con nuestra implementación?
- ¿Y tú?

NATURALEZA DEL PROBLEMA

- ¿Qué es lo que no funciona?:
  - Dos procesos concurrentes manipularon un recurso compartido (la cuenta bancaria) sin ningún tipo de sincronización:
    - Esto es lo que se denomina **race condition** (*condición de carrera*).
    - El resultado depende del orden en que tengan lugar los accesos.
- Debemos asegurar que **un único proceso** pueda manipular cada vez el recurso compartido.
  - De ese modo podremos justificar el comportamiento del programa.
  - Necesitamos sincronizar procesos.
- ¡Cuidado! Este problema ocurre incluso con una única variable compartida operando sobre un único procesador:
  - T1 y T2 comparten la variable X.
  - T1 incrementa X (X=X+1).
  - T2 decrementa X (X=X-1).

A NIVEL MÁQUINA SE INTERPRETA ESTO:

```
T1:   LOAD X
      INCR
      STORE X
```

```
T2:   LOAD X
      DECR
      STORE X
```

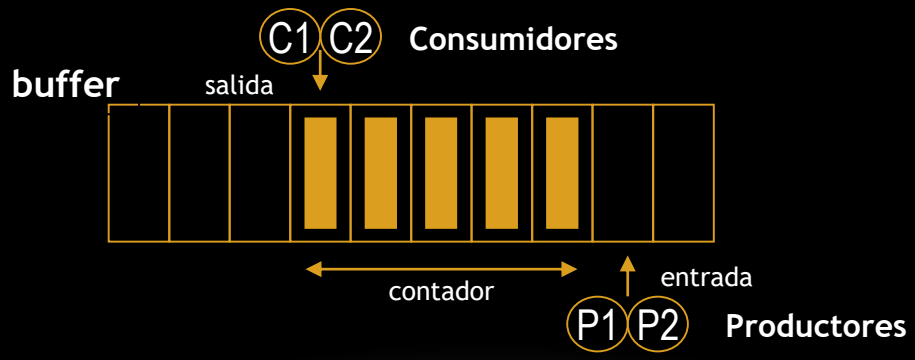
SOLUCIONES

- Existe la necesidad de comunicación entre procesos.
- La comunicación entre procesos puede seguir dos esquemas básicos:
  - **Comunicación por memoria compartida:**
    - Espacio de direcciones único.
    - El SO crea una zona de memoria accesible a un grupo de procesos.
    - **Problema de la sección crítica:** en un sistema con procesos concurrentes que se comunican compartiendo datos comunes es necesario sincronizar el acceso (lectura, escritura) a los datos compartidos para asegurar la consistencia de los mismos.
  - **Comunicación por paso de mensajes:**
    - Espacio de direcciones independientes.
    - Cuando dos procesos se comunican vía mensajes se necesitan mecanismos para que el proceso receptor espere (se suspende) a que el proceso emisor envíe el mensaje y éste esté disponible.
    - No existe el problema de la sección crítica.

MEMORIA COMPARTIDA

MEMORIA COMPARTIDA

- Es el caso del problema descrito anteriormente para el sistema de gestión de cuentas bancarias.
- Un ejemplo: productores y consumidores con *buffer* acotado:
  - **Productor:** proceso que produce elementos (a una cierta velocidad) y los deposita en un *buffer*.
  - **Consumidor:** proceso que toma elementos del *buffer* y los consume (a una velocidad probablemente distinta a la del productor).
  - **Buffer:** estructura de datos que sirve para intercambiar información entre los procesos productores y consumidores. Actúa a modo de depósito para absorber la diferencia de velocidad entre productores y consumidores.



IMPLEMENTACIÓN DEL *BUFFER*

```
Variables compartidas:
  var buffer: array[0..n-1] of elemento;
      entrada:=0, salida:=0 : 0..n-1;
      contador:=0 : 0..n;
```

```
Proceso productor:
task productor;
  var item: elemento;
  repeat
    item := producir();
    while contador=n do no-op;
    buffer[entrada] := item;
    entrada := (entrada + 1) mod n;
    contador := contador + 1;
  until false
end productor
```

```
Proceso consumidor:
task consumidor;
  var item: elemento;
  repeat
    while contador=0 do no-op;
    item := buffer[salida];
    salida := (salida + 1) mod n;
    contador := contador - 1;
  until false
end consumidor
```

CONDICIONES DE CARRERA

- **Corrección en programas secuenciales:** el programa cumple con sus especificaciones (responde a unos invariantes o reglas de corrección).
- **Corrección secuencial no implica corrección concurrente:** un programa que tiene una implementación "secuencialmente correcta" (correcta utilizando un único hilo de ejecución) puede presentar problemas cuando se intenta introducir concurrencia en forma de hilos de ejecución.
- **Condición de carrera:** la ejecución de un conjunto de operaciones concurrentes sobre una variable compartida, deja la variable en un estado inconsistente con las especificaciones de corrección. Además, el resultado de la variable depende de la velocidad relativa en que se ejecuten las operaciones (orden de ejecución).
- **Peligro potencial:** las condiciones de carrera pueden presentarse en algún escenario, pero no tienen por qué observarse en todas las posibles trazas de la ejecución del programa.

Es un problema tremendamente difícil de reproducir.

EJEMPLO DE CONDICIÓN DE CARRERA

- Buffer con un productor y un consumidor:

Productor:
contador := contador + 1;

```
mov reg1, contador;
inc reg1;
mov contador, reg1;
```

Consumidor:
contador := contador - 1;

```
mov reg2, contador;
dec reg2;
mov contador, reg2;
```

T	Proceso	Operación	reg1	reg2	contador
T0	productor	**mov** reg1, contador;	5	?	5
T1	productor	**inc** reg1;	6	?	5
T2	consumidor	**mov** reg2, contador;	?	5	5
T3	consumidor	**dec** reg2;	?	4	5
T4	consumidor	**mov** contador, reg2;	?	4	4
T5	productor	**mov** contador, reg1;	6	?	6

CONSISTENCIA SECUENCIAL

- El criterio de corrección/consistencia más usual para programas concurrentes es:

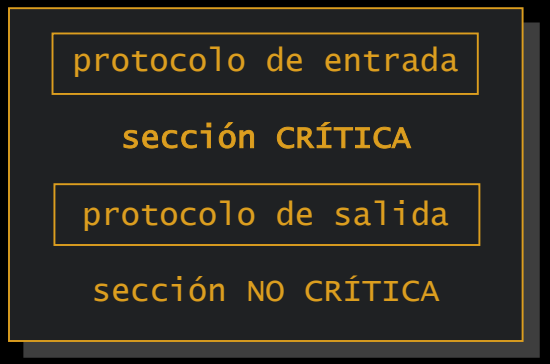
- **Serializabilidad (consistencia secuencial):** el resultado de la ejecución concurrente de un conjunto de operaciones ha de ser equivalente al resultado de ejecutar secuencialmente cada una de las operaciones, en alguno de los órdenes secuenciales posibles.

- **Condición de carrera = no serializabilidad:** No hay ninguna posible ejecución secuencial de un conjunto de operaciones que de el mismo resultado que la ejecución concurrente.

SECCIÓN CRÍTICA

EL PROBLEMA DE LA SECCIÓN CRÍTICA

- **Sección crítica:** zona de código en la que se accede a variables compartidas por varios procesos:
  - **Problemas potenciales:** puede introducir condiciones de carrera si no se adoptan las medidas adecuadas.
  - **Posible solución:** sincronizar el acceso a los datos de manera que mientras un proceso ejecuta su sección crítica ningún otro proceso ejecute la suya (***exclusión mutua***).
- Formulación del problema de la sección crítica:
  - Sean n procesos compitiendo para acceder a datos compartidos.
  - Cada proceso tiene una zona de código, denominada sección crítica, en la que accede a los datos compartidos.
  - **Problema:** encontrar un protocolo que los procesos puedan usar para cooperar



REQUISITOS PARA LA SOLUCIÓN DEL PROBLEMA DE LA SECCIÓN CRÍTICA

- **Exclusión mutua (*mutual exclusion*):** si un proceso está ejecutando su sección crítica ningún otro proceso puede estar ejecutando la suya.
- **Progreso:** si ningún proceso está ejecutando su sección crítica y hay otros que están intentando entrar en la suyas, entonces la decisión de qué proceso entrará en la sección crítica se toma en un tiempo finito y sólo puede ser seleccionado uno de los procesos que deseen entrar.
- **Espera limitada (*bounded waiting*):** después de que un proceso haya solicitado entrar en su sección crítica, existe un límite en el número de veces que se permite que otros procesos entren en sus secciones críticas (no se produce ***inanición***).
- **Rendimiento:** la sobrecarga de entrar y salir de la sección crítica no debe ser comparable respecto al trabajo que se realiza dentro de ella.

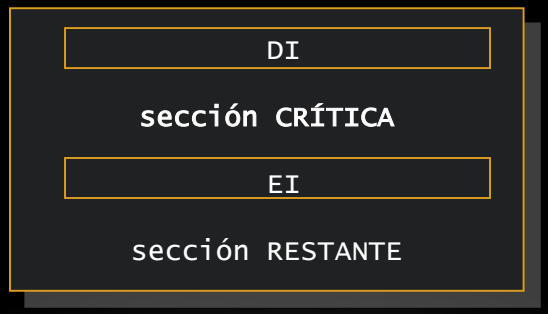
SOPORTE HARDWARE

SOPORTE HARDWARE

- Nos ayudamos de capacidades del propio *hardware* para resolver el problema de la sección crítica.
- Tenemos dos aproximaciones posibles:
  - Habilitar y deshabilitar interrupciones.
  - Recurrir a instrucciones máquina atómicas:
    - **Ejemplo:** TestAndSet (comprobar y asignar).
    - Con ellas construimos **cerrojos** (*locks*).

HABILITAR/DESHABILITAR INTERRUPCIONES

- El control de interrupciones se realiza utilizando las instrucciones:
  - **DI** (*Disable Interrupts*): deshabilitar interrupciones.
  - **EI** (*Enable Interrupts*): habilitar interrupciones.
- Se consigue la exclusión mutua inhibiendo los cambios de contexto durante la sección crítica, obligando así a que los procesos se ejecuten de manera atómica.
- Podemos deshabilitar las interrupciones antes de entrar en la sección crítica para prevenir los cambios de contexto, y volver a habilitarlas al abandonar la sección crítica.



PROPIEDADES DE HABILITAR/DESHABILITAR INTERRUPCIONES

- **Es una solución sólo disponible a nivel de núcleo del SO:** no se puede dejar el control de interrupciones en manos de procesos de usuario. Las instrucciones DI e EI sólo se pueden ejecutar en modo privilegiado.
- **Es un solución sólo disponible para máquinas monoprocesador:** Sólo se controlan las interrupciones de uno de los núcleos.
- **Es una solución con un alto coste en rendimiento:** Con las interrupciones desactivadas un procesador no puede cambiar entre procesos, viendo su rendimiento seriamente mermado.

INSTRUCCIONES ATÓMICAS: TEST AND SET (TAS)

- Lo que hace la instrucción TestAndSet es:
  - Modificar el valor de una variable.
  - Devolver el antiguo valor de dicha variable.
  - Ejecutarse de forma atómica.
- Su uso más común es como *boolean* y poniendo la variable a *true*.
- Se puede utilizar para implementar cerrojos *(locks)* simples.

```
function TAS (var objetivo: boolean): boolean;
  begin
    TAS:=objetivo;
    objetivo:=true;
  end
```

```
boolean TAS(boolean *lock) {
  boolean old = *lock;
  *lock = true;
  return old;
}
```

DEFINICIONES ALTERNATIVAS DE TAS

- Lock **siempre** es *true* al salir de un bloque TAS:
  - Tanto si ya era *true* (bloqueado) y por tanto no cambia nada, como si era *false* (disponible) y el que lo llama es el que se apropia.
- El valor devuelto **puede ser** tanto *true* cuando ya estaba bloqueado (*locked*) como *false* si estaba disponible previamente:
  - El valor devuelto podría invertirse, aunque el único efecto es invertir la condición de comprobación al usar TAS.
- Dos ejemplos de implementación:

```
boolean test_and_set(boolean *lock) {
  boolean old = *lock;
  *lock = true;
  return old;
}
```

```
boolean test_and_set(boolean *lock) {
  if(*lock == false) {
    *lock = true;
    return false;
  } else{
    return true;
  }
}
```

TAS PARA IMPLEMENTAR CERROJOS

- Existen dos operaciones básicas al operar con locks: *acquire()* y *release()*, es decir, adquirir y liberar.

```
boolean lock;

boolean acquire(boolean *lock) {
  while(test_and_set(lock));
}

void release(boolean *lock) {
  *lock = false;
}
```

La definición alternativa de TAS hace que el bucle se defina como: while (!TAS(lock))

- Se produce un ***spinlock***, una espera activa: el proceso continuamente ejecuta el bucle while en acquire(), consumiendo ciclos de CPU.
- Los cerrojos por sí mismos no evitan la inanición de procesos. Es decir, no se cumple la condición de **espera limitada**.

SOLUCIÓN CON CERROJOS AL PROBLEMA INICIAL

FUNCIONES UTILIZADAS

```
Withdraw(acct, amt) {
  acquire(lock);
  balance = get_balance(acct);
  balance = balance - amt;
  put_balance(acct,balance);
  release(lock);
  return balance;
}
```

```
Deposit(acct, amt) {
  acquire(lock);
  balance = get_balance(acct);
  balance = balance + amt;
  put_balance(acct,balance);
  release(lock);
  return balance;
}
```

POSIBLE PLANIFICACIÓN

```
acquire(lock);
balance = get_balance(acct);
balance = balance - amt;
```

```
acquire(lock);
```

```
put_balance(acct,balance);
release(lock);
```

```
balance = get_balance(acct);
balance = balance + amt;
put_balance(acct,balance);
release(lock);
```

SOLUCIÓN A LA SECCIÓN CRÍTICA USANDO TAS Y ESPERA LIMITADA (PSEUDOCÓDIGO)

- Para asegurar **espera limitada** añadimos una cola ordenada de procesos.

Variables compartidas

```
var esperando:=FALSE; array[0..n-1] of boolean; cerrojo:= FALSE : boolean;
```

Algoritmo del Proceso i

```
task Pi;
  var j: 0..n-1;
      llave: boolean;

  esperando[i]:=TRUE;
  llave:=TRUE;
  while esperando[i] and llave do llave:=test_and_set(cerrojo);
  esperando[i]:=FALSE;

  seccion critica

  j:=i+1 mod n;
  while (j<>i) and (not esperando[j]) do j:=j+1 mod n;
  if j=i then cerrojo:=FALSE else esperando[j]:=FALSE;

  seccion restante
end Pi;
```

ESPERA ACTIVA

SOLUCIÓN A LA SECCIÓN CRÍTICA USANDO TAS Y ESPERA LIMITADA (LENGUAJE C)

- Para asegurar **espera limitada** añadimos una cola ordenada de procesos.

Variables compartidas

```
boolean esperando[n]; boolean cerrojo = false;
```

Algoritmo del Proceso i

```
int j = 0; /* j = 0...n-1 */
  boolean llave;

  esperando[i] = true;
  llave = true;
  while (esperando[i] && llave)
     llave = test_and_set(cerrojo);
  esperando[i] = false;

  seccion critica

  j:=(i+1)%n;
  while (j!=i && (esperando[j]==false))
     j=(j+1)%n;
  if (j==i) cerrojo = false; else esperando[j] = false;

  seccion restante
```

ESPERA ACTIVA

SOLUCIÓN A LA SECCIÓN CRÍTICA USANDO TAS Y ESPERA LIMITADA

Proceso 0 (i=0)

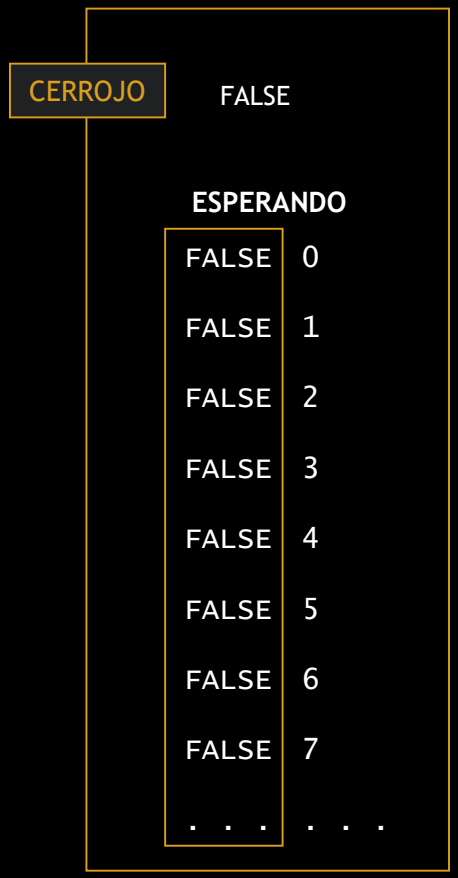
```
esperando[i]:=TRUE;
llave:=TRUE;
while esperando[i] and llave do llave:=test_and_set(cerrojo);
esperando[i]:=FALSE;
```

```
j:=i+1 mod n;
while (j<>i) and (not esperando[j]) do j:=j+1 mod n;
if j=i then cerrojo:=FALSE else esperando[j]:=FALSE;
```

Proceso 4 (i=4)

```
esperando[i]:=TRUE;
llave:=TRUE;
while esperando[i] and llave do llave:=test_and_set(cerrojo);
esperando[i]:=FALSE;
```

```
j:=i+1 mod n;
while (j<>i) and (not esperando[j]) do j:=j+1 mod n;
if j=i then cerrojo:=FALSE else esperando[j]:=FALSE;
```



SOLUCIÓN A LA SECCIÓN CRÍTICA USANDO TAS Y ESPERA LIMITADA

Proceso 0 (i=0)

```
esperando[i]:=TRUE;
llave:=TRUE;
while esperando[i] and llave do llave:=test_and_set(cerrojo);
esperando[i]:=FALSE;
```

```
j:=i+1 mod n;
while (j<>i) and (not esperando[j]) do j:=j+1 mod n;
if j=i then cerrojo:=FALSE else esperando[j]:=FALSE;
```

Proceso 4 (i=4)

```
esperando[i]:=TRUE;
llave:=TRUE;
while esperando[i] and llave do llave:=test_and_set(cerrojo);
esperando[i]:=FALSE;
```

```
j:=i+1 mod n;
while (j<>i) and (not esperando[j]) do j:=j+1 mod n;
if j=i then cerrojo:=FALSE else esperando[j]:=FALSE;
```

SOLUCIÓN A LA SECCIÓN CRÍTICA USANDO TAS Y ESPERA LIMITADA

Proceso 0 (i=0)

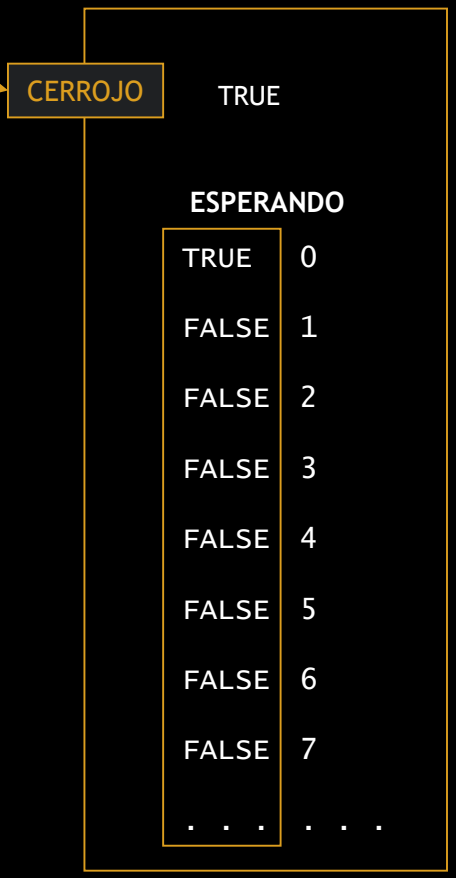
```
esperando[i]:=TRUE;
llave:=TRUE;
while esperando[i] and llave do llave:=test_and_set(cerrojo);
esperando[i]:=FALSE;
```

```
j:=i+1 mod n;
while (j<>i) and (not esperando[j]) do j:=j+1 mod n;
if j=i then cerrojo:=FALSE else esperando[j]:=FALSE;
```

Proceso 4 (i=4)

```
esperando[i]:=TRUE;
llave:=TRUE;
while esperando[i] and llave do llave:=test_and_set(cerrojo);
esperando[i]:=FALSE;
```

```
j:=i+1 mod n;
while (j<>i) and (not esperando[j]) do j:=j+1 mod n;
if j=i then cerrojo:=FALSE else esperando[j]:=FALSE;
```



SOLUCIÓN A LA SECCIÓN CRÍTICA USANDO TAS Y ESPERA LIMITADA

Proceso 0 (i=0)

```
esperando[i]:=TRUE;
llave:=TRUE;
while esperando[i] and llave do llave:=test_and_set(cerrojo);
esperando[i]:=FALSE;
```

```
j:=i+1 mod n;
while (j<>i) and (not esperando[j]) do j:=j+1 mod n;
if j=i then cerrojo:=FALSE else esperando[j]:=FALSE;
```

Proceso 4 (i=4)

```
esperando[i]:=TRUE;
llave:=TRUE;
while esperando[i] and llave do llave:=test_and_set(cerrojo);
esperando[i]:=FALSE;
```

```
j:=i+1 mod n;
while (j<>i) and (not esperando[j]) do j:=j+1 mod n;
if j=i then cerrojo:=FALSE else esperando[j]:=FALSE;
```

SOLUCIÓN A LA SECCIÓN CRÍTICA USANDO TAS Y ESPERA LIMITADA

Proceso 0 (i=0)

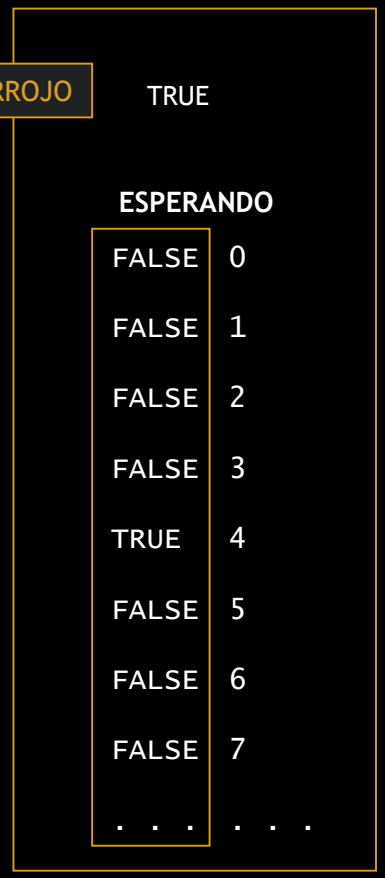
```
esperando[i]:=TRUE;
llave:=TRUE;
while esperando[i] and llave do llave:=test_and_set(cerrojo);
esperando[i]:=FALSE;
```

```
j:=i+1 mod n;
while (j<>i) and (not esperando[j]) do j:=j+1 mod n;
if j=i then cerrojo:=FALSE else esperando[j]:=FALSE;
```

Proceso 4 (i=4)

```
esperando[i]:=TRUE;
llave:=TRUE;
while esperando[i] and llave do llave:=test_and_set(cerrojo);
esperando[i]:=FALSE;
```

```
j:=i+1 mod n;
while (j<>i) and (not esperando[j]) do j:=j+1 mod n;
if j=i then cerrojo:=FALSE else esperando[j]:=FALSE;
```



SOLUCIÓN A LA SECCIÓN CRÍTICA USANDO TAS Y ESPERA LIMITADA

Proceso 0 (i=0)

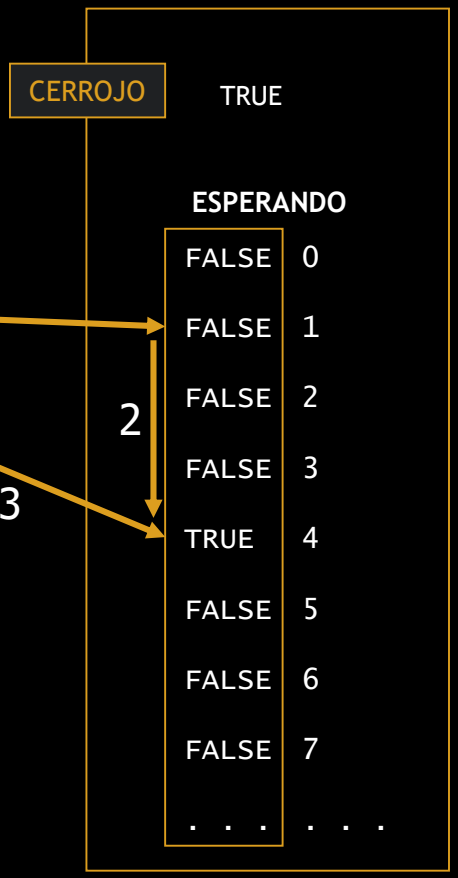
```
esperando[i]:=TRUE;
llave:=TRUE;
while esperando[i] and llave do llave:=test_and_set(cerrojo);
esperando[i]:=FALSE;
```

```
j:=i+1 mod n;
while (j<>i) and (not esperando[j]) do j:=j+1 mod n;
if j=i then cerrojo:=FALSE else esperando[j]:=FALSE;
```

Proceso 4 (i=4)

```
esperando[i]:=TRUE;
llave:=TRUE;
while esperando[i] and llave do llave:=test_and_set(cerrojo);
esperando[i]:=FALSE;
```

```
j:=i+1 mod n;
while (j<>i) and (not esperando[j]) do j:=j+1 mod n;
if j=i then cerrojo:=FALSE else esperando[j]:=FALSE;
```



SOLUCIÓN A LA SECCIÓN CRÍTICA USANDO TAS Y ESPERA LIMITADA

Proceso 0 (i=0)

```
esperando[i]:=TRUE;
llave:=TRUE;
while esperando[i] and llave do llave:=test_and_set(cerrojo);
esperando[i]:=FALSE;
```

```
j:=i+1 mod n;
while (j<>i) and (not esperando[j]) do j:=j+1 mod n;
if j=i then cerrojo:=FALSE else esperando[j]:=FALSE;
```

Proceso 4 (i=4)

```
esperando[i]:=TRUE;
llave:=TRUE;
while esperando[i] and llave do llave:=test_and_set(cerrojo);
esperando[i]:=FALSE;
```

```
j:=i+1 mod n;
while (j<>i) and (not esperando[j]) do j:=j+1 mod n;
if j=i then cerrojo:=FALSE else esperando[j]:=FALSE;
```

SOLUCIÓN A LA SECCIÓN CRÍTICA USANDO TAS Y ESPERA LIMITADA

Proceso 0 (i=0)

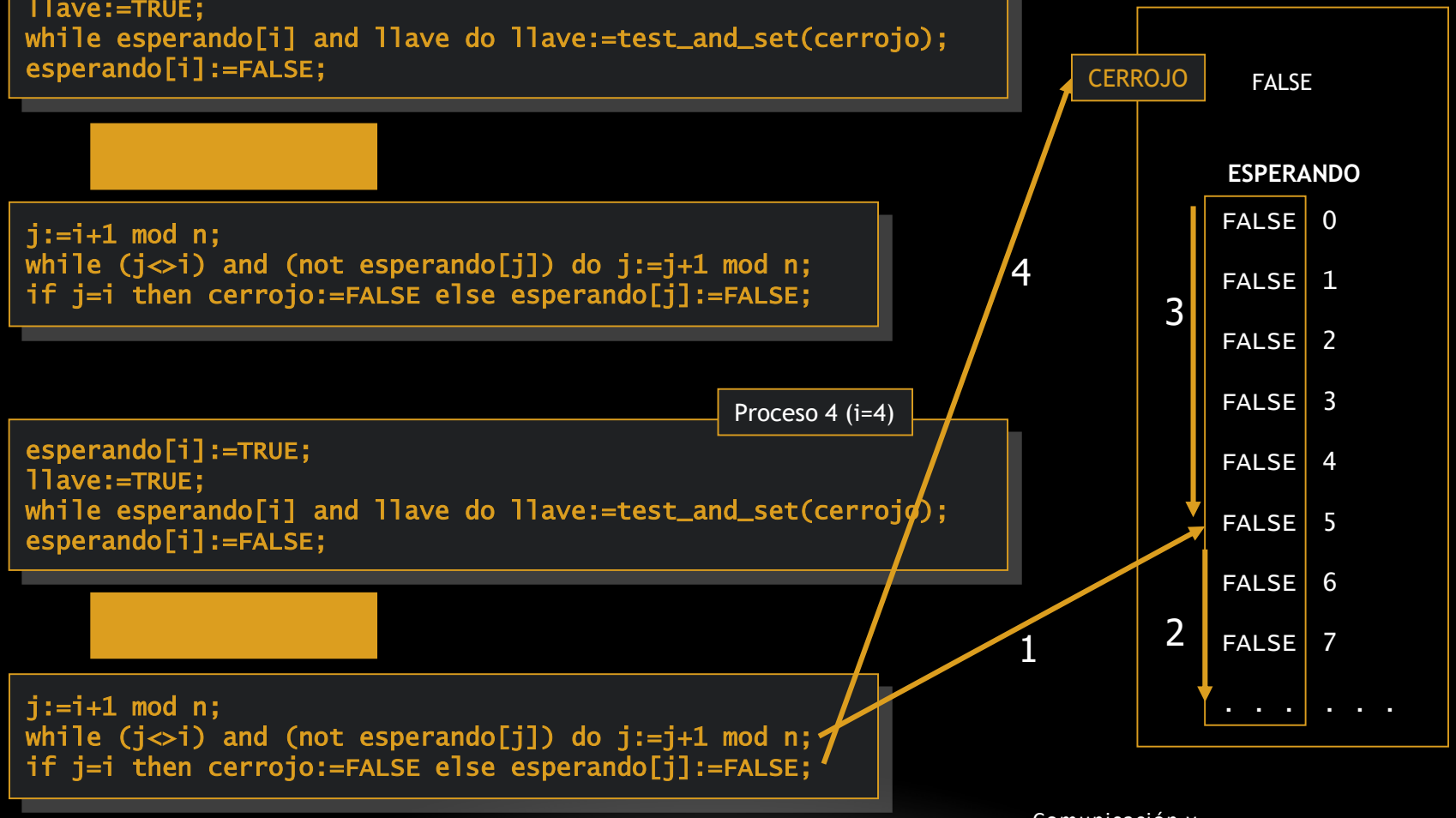
```
esperando[i]:=TRUE;
llave:=TRUE;
while esperando[i] and llave do llave:=test_and_set(cerrojo);
esperando[i]:=FALSE;
```

```
j:=i+1 mod n;
while (j<>i) and (not esperando[j]) do j:=j+1 mod n;
if j=i then cerrojo:=FALSE else esperando[j]:=FALSE;
```

Proceso 4 (i=4)

```
esperando[i]:=TRUE;
llave:=TRUE;
while esperando[i] and llave do llave:=test_and_set(cerrojo);
esperando[i]:=FALSE;
```

```
j:=i+1 mod n;
while (j<>i) and (not esperando[j]) do j:=j+1 mod n;
if j=i then cerrojo:=FALSE else esperando[j]:=FALSE;
```



PROPIEDADES DE LAS INSTRUCCIONES MÁQUINA ATÓMICAS

- Tienen ciertas ventajas:
  - Sirven para sistemas multiprocesador.
  - Son simples, y por tanto, fáciles de comprobar.
  - Pueden utilizarse para varias secciones críticas.
    - Cada sección crítica define su variable.
- Pero también presentan varios problemas:
  - Se emplea **espera activa**: alto consumo de CPU.
  - Es posible la **inanición** (*starvation*)*:* entrar en la sección crítica depende del planificador.
  - Es posible el **interbloqueo** (*deadlock*): un proceso en su sección crítica es interrumpido (mediante interrupciones) por otro que quiere acceder a un recurso reservado.

ABSTRACCIONES DE ALTO NIVEL

- **Cerrojos** (*locks*):
  - *Ventajas:* mínima semántica.
  - *Desventajas:* muy primitivos, espera activa, inanición.
- **Semáforos** (*semaphores*):
  - *Ventajas:* básicos, fáciles de entender.
  - *Desventajas:* la programación es complicada.
- **Monitores** (*monitors*):
  - *Ventajas:* de alto nivel.
  - *Desventajas:* se necesita soporte del lenguaje de programación.
- **Mensajes** (*messages*):
  - *Ventajas:* modelo de comunicación y sincronización simple, aplicación directa a sistemas distribuidos.
  - *Desventajas:* muy alto coste en recursos.

SEMÁFOROS

SEMÁFOROS

- Inventados por Dijkstra en 1965.
- Los semáforos son estructuras de datos proporcionadas por el sistema operativo que permiten implementar sincronización.
- Estas estructuras incluyen:
  - **Una variable entera** a la que se accede exclusivamente mediante dos operaciones atómicas.
  - **La operación atómica *wait*** (también llamada P o decremento): decrementa la variable. Si la variable pasa a ser negativa bloquea el proceso que ha invocado la operación.
  - **La operación atómica *signal*** (también llamada V o incremento): incrementa la variable y desbloquea un proceso que haya invocado anteriormente *wait*, si existe.
  - Una **cola de procesos** en espera.

TIPOS DE SEMÁFOROS

- **Mutex o binarios:**
  - Acceso individual a un recurso.
  - Sólo un proceso puede acceder a la vez al recurso asociado.
  - Garantizan la exclusión mutua a una sección crítica.
- **De conteo:**
  - Representan un recurso con varias unidades disponibles o un recurso que permita ciertos tipos de accesos concurrentes sincronizados (como puede ser la lectura).
  - Múltiplos procesos puede acceder al recurso asociado.
  - El número máximo de procesos se determina mediante el valor inicial del semáforo (*count*).
- Los mutex pueden verse como una particularización de los de conteo dónde *count = 1*.

MUTEX

- Similares a los cerrojos, pero con semántica diferente.

Existe un semáforo S, asociado a la cuenta

```
typedef struct account {
  double balance;
  semaphore S;
} account_t;

Withdraw(account_t *acct, amt) {
  double bal;
  wait(lacct->S);
  bal = acct->balance;
  bal = bal - amt;
  acct->balance = bal;
  signal(acct->S);
  return bal;
}
```

Tres threads ejecutan Withdraw()

```
wait(acct->S);
bal = acct->balance;
bal = bal - amt;
```

```
wait(acct->S);
```

```
wait(acct->S);
```

```
acct->balance= bal;
signal(acct->S);
```

```
. . .
signal(acct->S);
```

```
. . .
signal(acct->S);
```

No está definido qué thread se ejecuta después de un signal

PROPIEDADES DE LOS SEMÁFOROS

- Existen dos tipos de semáforos: **robustos** y **débiles** (*strong and weak*):
  - El uso correcto de los semáforos no depende del orden en el que los procesos son desbloqueados.
  - Si se especifica un orden en la definición del semáforo (cola FIFO, que **asegura la espera limitada**) tenemos un semáforo robusto.
  - Si el orden se deja libre, sin especificación, se trata de un semáforo débil.
- Atomicidad de *wait()* y *signal()*:
  - Debemos asegurarnos de que dos procesos no pueden ejecutar *wait* y *signal* simultáneamente
  - Esto provoca otro problema de sección crítica. Soluciones:
    - Usar primitivas de bajo nivel.
    - **Monoprocesadores:** deshabilitar interrupciones.
    - **Multiprocesadores:** usar instrucciones máquina o soluciones software como el algoritmo de Peterson (visto anteriormente).

DEFINICIÓN FORMAL DE SEMÁFORO

- Tipo de datos que toma valores enteros y sobre el que se definen las siguientes operaciones atómicas:
  - S: semaforo(N) (semáforo S con valor inicial N >=0)
  - P(S): wait(S)

```
S=S-1;
if (S<0) block(S);
```

ATÓMICAMENTE

  - V(S): signal(S)

```
S=S+1;
if (S<=0) ready(S);
```

ATÓMICAMENTE

  - La operación *block(S)* suspende al proceso que ha invocado al semáforo y lo introduce en una cola de espera asociada a S.
  - La operación *ready(S)* extrae un proceso de la cola de espera asociada a S y lo activa.

Semáforo

SOLUCIÓN A LA SECCIÓN CRÍTICA USANDO SEMÁFOROS

- Variables compartidas por todos los procesos:

```
semaphore mutex = 1;
```

Algoritmo del Proceso i

```
...

  wait(mutex)

  seccion critica

  signal(mutex)

  seccion restante

...
```

- Los semáforos **no requieren de espera activa**.

SEMÁFOROS COMO MECANISMO DE SINCRONIZACIÓN DE USO GENERAL

- Para conseguir exclusión mutua.
- Para forzar relaciones de precedencia, como por ejemplo:
  - El proceso P_j debe ejecutar B después de que el proceso P_i haya ejecutado A.

```
semaphore sinc = 0;
```

```
/* Pi */
  . . .
  A;
  V(sinc);
  . . .
```

```
/* Pj */
  . . .
  P(sinc);
  B;
  . . .
```

```
/* Pi */
  . . .
  A;
  signal(sinc);
  . . .
```

```
/* Pj */
  . . .
  wait(sinc);
  B;
  . . .
```

USO INCORRECTO DE SEMÁFOROS

- **Interbloqueos:** si se da un conjunto de procesos en el que todos esperan indefinidamente a un evento que sólo otro proceso del conjunto puede producir.

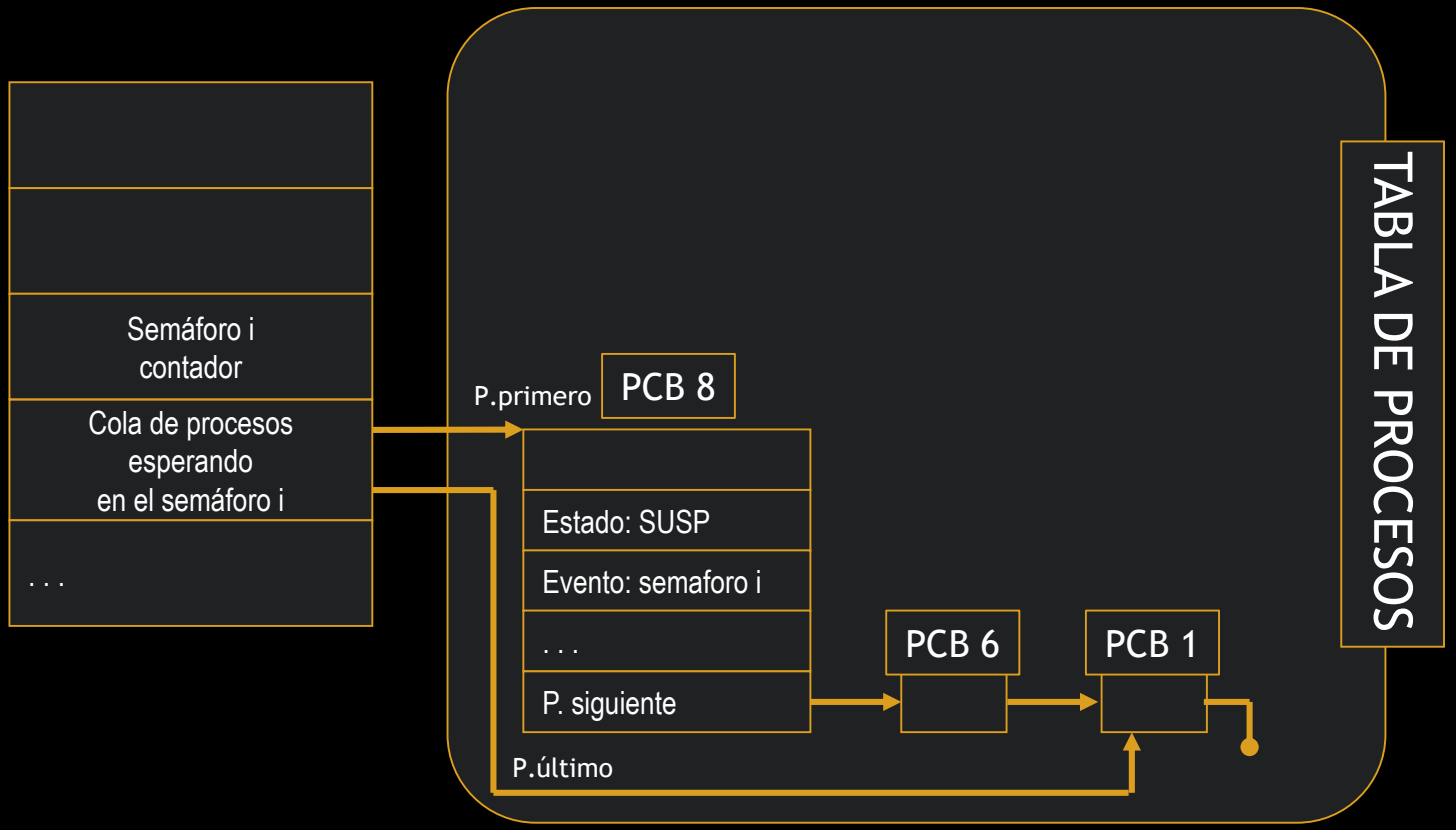
```
semaphore s1 = 1;
semaphore s2 = 1;
```

```
/* Pi */
  . . .
  P(s1);
  P(s2);
  . . .
  V(s1);
  V(s2);
  . . .
```

```
/* Pj */
  . . .
  P(s2);
  P(s1);
  . . .
  V(s1);
  V(s2);
  . . .
```

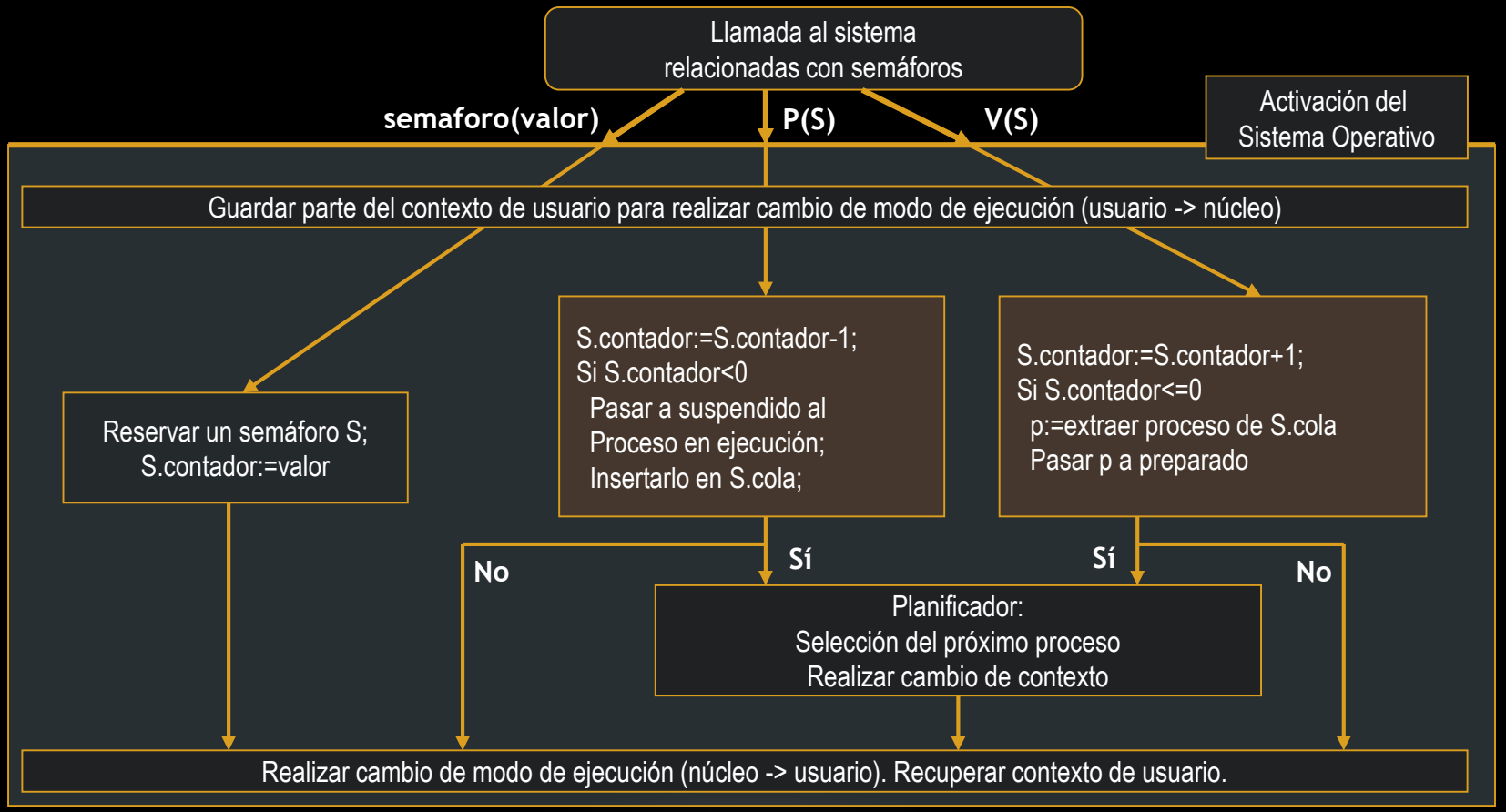
IMPLEMENTACIÓN DE SEMÁFOROS (I)

- Estructuras de control del sistema:



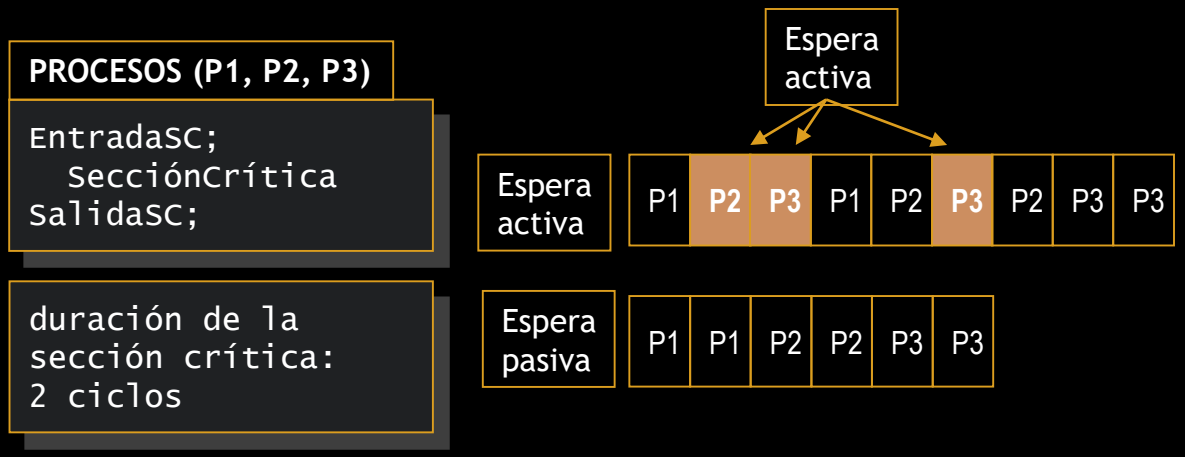
IMPLEMENTACIÓN DE SEMÁFOROS (II)

- Flujo de control del sistema operativo:

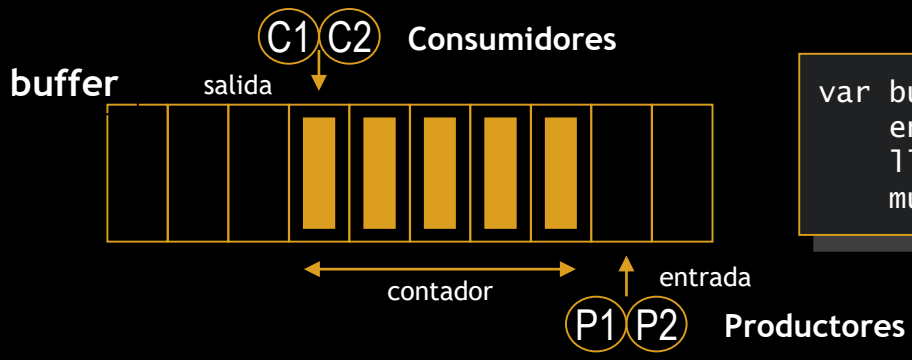


SEMÁFOROS VS. ESPERA ACTIVA

- Espera activa:
  - El proceso que espera entrar en la sección crítica desaprovecha el tiempo de CPU comprobando cuándo puede entrar (ejecutando las instrucciones del protocolo de ingreso).
- Semáforos:
  - La espera se realiza en la cola de semáforos, que es una cola de procesos suspendidos.
  - El proceso que espera entrar en la sección crítica no utiliza CPU; está suspendido. Será avisado cuando sea su turno.



PROBLEMA DE LOS PRODUCTORES Y CONSUMIDORES CON *BUFFER* ACOTADO USANDO SEMÁFOROS



Variables compartidas:

```
var buffer: array[0..n-1] of elemento;
    entrada:=0, salida:=0, contador:=0: 0..n;
    lleno: semaforo(0), vacio: semaforo(n);
    mutex: semaforo(1);
```

```
task productor;
  var item: elemento;
  repeat
    item:=producir();
    P(vacio);
    P(mutex);
      buffer[entrada]:=item;
      entrada:=(entrada+1) mod n;
      contador:=contador+1;
    V(mutex);
    V(lleno);
  until false
end productor;
```

```
task consumidor;
  var item: elemento;
  repeat
    P(lleno);
    P(mutex);
      item:=buffer[salida];
      salida:=(salida+1) mod n;
      contador:=contador-1;
    V(mutex);
    V(vacio);
    consumir(item);
  until false
end consumidor;
```

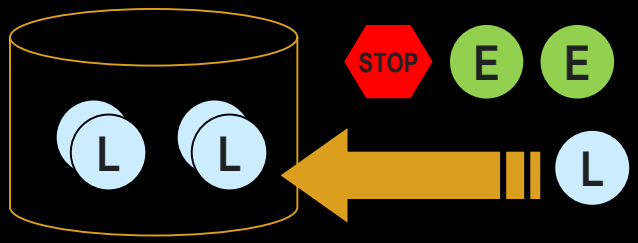
PROBLEMA DE LOS LECTORES Y ESCRITORES

- El problema de los lectores y escritores:
  - Hay un conjunto de datos comunes (un fichero, una base de datos, etc.).
  - Los procesos lectores, acceden a los datos en modo de sólo lectura.
  - Los procesos escritores, acceden a los datos en modo lectura-escritura.
- Especificación del protocolo: reglas de corrección
  - Diversos lectores pueden leer concurrentemente.
  - Los escritores se excluyen mutuamente entre ellos.
  - Los escritores se excluyen mutuamente con los lectores.

PROBLEMA DE LOS LECTORES Y ESCRITORES: POSIBLES VARIANTES

- Especificación del protocolo: reglas de prioridad
  - **Primera variante:** prioridad a los lectores:
    - Si hay lectores leyendo y…
    - …escritores esperando, entonces…
    - …un nuevo lector tiene preferencias sobre el escritor que espera.
    - **Hay inanición para los escritores** si no paran de llegar lectores.
  - **Segunda variante:** prioridad a los escritores:
    - Si hay escritores esperando, entonces…
    - …éstos tienen preferencia sobre los nuevos lectores que lleguen.
    - **Hay inanición para los lectores** si no paran de llegar escritores.
  - **Tercera variante**, sin inanición para ningún tipo de proceso:
    - Sólo se puede implementar con semáforos robustos.
    - Es compleja y subóptima.

PROBLEMA DE LOS LECTORES Y ESCRITORES: PRIMERA VARIANTE



Variables compartidas:

```
semaphore mutex = 1;
semaphore writer = 1;
int readers =0;
```

```
/* escritor */
  ...
  wait(writer);
    escribir();
  signal(writer);
  ...
```

PRIMERA VARIANTE: PRIORIDAD A LOS LECTORES

```
/* lector */
  ...
  wait(mutex);
    readers = readers+1;
    if (readers==1) wait(writer);
  signal(mutex);
    leer();
  wait(mutex);
    readers = readers-1;
    if (readers==0) signal(writer);
  signal(mutex);
  ...
```

PROBLEMA DE LOS LECTORES Y ESCRITORES: SEGUNDA VARIANTE

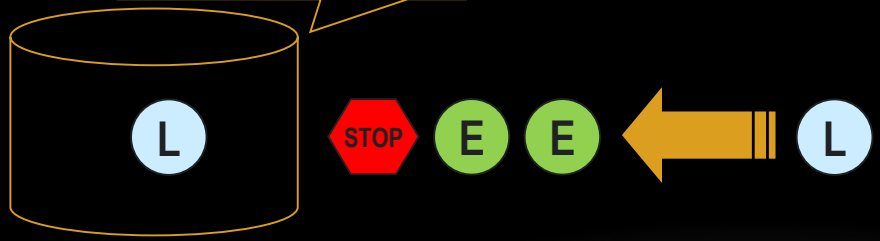
Variables compartidas:

```
semaphore mutexWri = 1;
semaphore mutexRead = 1;
semaphore writer = 1;
semaphore reader = 1;
semaphore preReader = 1;
int writers = 0;
int readers =0;
```

```
/* escritor */
...
  wait(mutexWri);
    writers = writers+1;
    if (escritores==1) wait(reader);
  signal(mutexWri);
  wait(writer);
    escribir();
  signal(writer);
  wait(mutexWri);
    writers = writers-1;
    if (writers==0) signal(reader);
  signal(mutexWri);
...
```

```
/* lector */
...
wait(preReader);
  wait(reader);
    wait(mutexRead);
      readers = readers+1;
      if (readers==1) wait(writer);
    signal(mutexRead);
  signal(reader);
signal(preReader);
leer();
wait(mutexRead);
  readers = readers-1;
  if (readers==0) signal(writer);
signal(mutexRead);
...
```

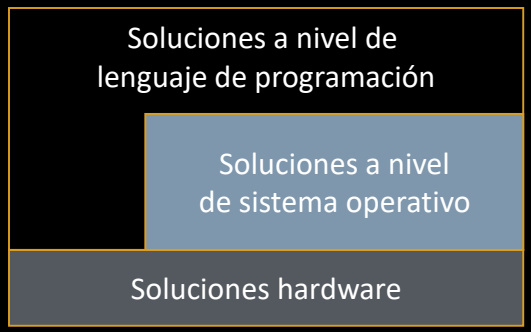
SEGUNDA VARIANTE: PRIORIDAD A LOS ESCRITORES



MONITORES

CONSTRUCCIONES LINGÜÍSTICAS

- Con el fin de simplificar la programación de aplicaciones concurrentes, algunos lenguajes de programación proporcionan herramientas (construcciones lingüísticas) que facilitan la sincronización entre las tareas y proporcionan un modelo de programación uniforme:
  - Monitores.
  - Tipos protegidos en ADA.
  - Métodos *synchronized* en Java.



MONITORES

- Un monitor es un tipo de datos que encapsula **datos** y **operaciones**.
- Proporciona: **exclusión mutua** y **sincronización**.
- Sintaxis:

```
type nombre_monitor = monitor
   ...declaración de variables

procedure entry P1(...);
   begin ... end;

function entry F2(...);
   begin ... end;
...
begin
  ...Código de inicialización
end;
```

Variables internas al monitor:

- Variables compartidas
- Variables condición

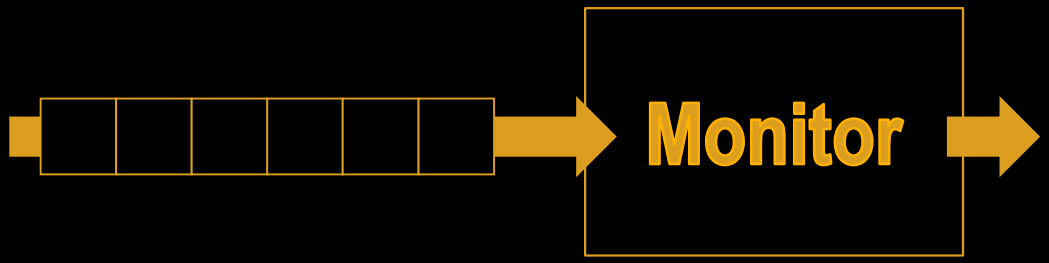
Métodos de acceso:

- Única vía de acceso a las variables compartidas
- Exclusión mutua

- Se invoca automáticamente al instanciar el monitor y antes de ser accedido por ningún proceso.
- Se utiliza para la inicialización de las variables del monitor.

EXCLUSIÓN MUTUA EN MONITORES

- La propia estructura del monitor garantiza la exclusión mutua.
- El monitor tiene una cola de entrada.
- **Sólo un proceso** puede encontrarse a la vez dentro de un monitor.
  - Sea cual sea el método de acceso que se esté ejecutando.



SINCRONIZACIÓN EN MONITORES

- En monitores, para definir esquemas complejos de sincronización, se pueden definir las variables del tipo *condition* (condición).
  - **var x: condition**
- Sobre las variables *condition* se pueden realizar las siguientes operaciones:
  - **x.wait**: suspende (en una cola asociada a x) el proceso que invocó la operación.
  - **x.signal**: reanuda (si existe) un proceso suspendido en la cola asociada a x.
  - **x.awaited**: indica el número de procesos suspendidos en la cola asociada a x.

FUNCIONAMIENTO DE LOS MONITORES

- En los monitores *wait* y *signal* **no tienen el mismo significado** que en los semáforos.
- Si no hay procesos suspendidos, una operación *signal* sobre la variable de condición no tiene ningún efecto sobre el monitor, en contraste con la operación *signal* realizada sobre un semáforo, que siempre modifica el estado del semáforo.
- Supongamos que el proceso P ejecuta una operación *signal* sobre x y que existe además un proceso Q suspendido asociado a la variable de condición x; los dos procesos pueden conceptualmente continuar su ejecución, aunque sin embargo no pueden estar activos simultáneamente en el monitor.

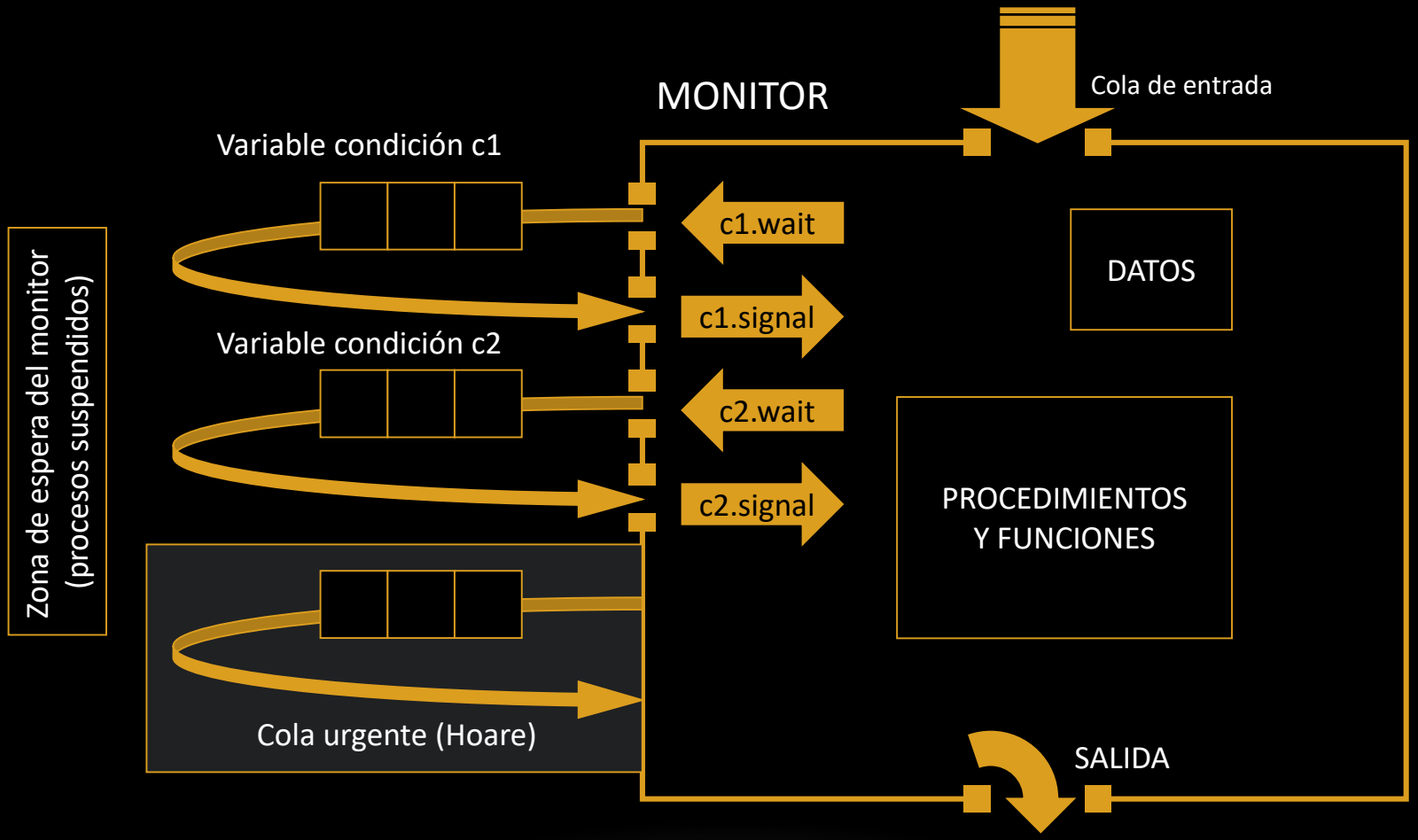
VARIANTES DEL MONITOR

- Existen diferentes variantes en la definición de un monitor según resuelvan el siguiente problema:
- Un proceso P ejecuta *x.signal* y activa a otro proceso Q. Potencialmente P y Q pueden continuar su ejecución dentro del monitor. ¿Cuál de ellos se ejecuta en realidad?
  - **Modelo de Hoare:** El proceso que invoca la operación *x.signal* (P) se suspende (en una cola de urgencia) de forma que el proceso reanudado por la operación *x.signal* (Q) pasa a ejecutarse dentro del monitor.
  - **Modelo de Brinch-Hansen:** La operación *x.signal* ha de ser la última operación que un proceso invoca antes de salir del monitor.
  - **Modelo de Lampson y Redell:** x.signal no existe, en su lugar se utiliza *x.notify*. El proceso que invoca la operación *x.notify* (P) continúa ejecutándose y Q se ejecutará cuando el monitor quede libre.

SIGNAL VS. NOTIFY

- *Signal:*
  - Si *signal* no es la última operación del proceso se necesitará un cambio de contexto adicional.
  - El planificador no puede permitir que un proceso "se cuele" entre el proceso que invoca *signal* y el que debe ser despertado.
  - **Más restrictivo.**
- *Notify:*
  - No se asegura que no se ejecuten otros procesos en el que invoca *notify* y el despertado. Tampoco que la condición de la que depende el despertado no haya cambiado.
  - El proceso despertado debe volver a comprobar la condición.
    - Usando *while()* en lugar de *if()*.
  - **Solución menos propensa a error.**
- Mejoras a notify:
  - **Broadcast:** Despertar a varios.
  - **Timer:** Tiempo máximo de espera hasta que despierte, incluso sin ser notificado.

MODELO DE COLAS EN MONITORES



IMPLEMENTACIÓN DE MONITORES USANDO SEMÁFOROS (I)

- **Monitor de Hoare**
- Variables globales, por cada monitor se definen:

```
semaphore mutex = 1;
semaphore urgente = 0; /* cola de urgencia*/
int cont_urg = 0;
```

- Por cada procedimiento o función F, se genera:

```
P(mutex);
...cuerpo de F...
if (cont_urg > 0)
  V(urgente);
else
  V(mutex);
```

IMPLEMENTACIÓN DE MONITORES USANDO SEMÁFOROS (II)

- Para cada variable condición x se define:

```
semaphore x_sem = 0;
int x_cont = 0;
```

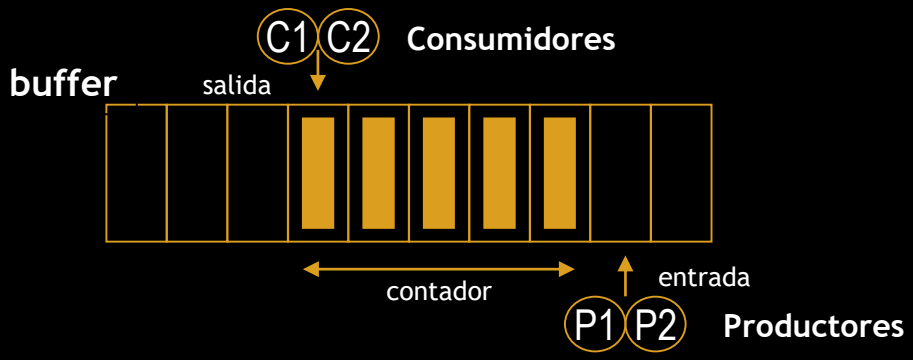
Operación X.wait

```
x_cont = x_cont+1;
if (cont_urg > 0)
  V(urgente);
else
  V(mutex);
P(x_sem);
x_cont = x_cont-1;
```

Operación X.signal

```
if (x_cont > 0) {
  cont_urg = cont_urg+1;
  V(x_sem);
  P(urgente);
  cont_urg = cont_urg-1;
}
```

PROBLEMA DE LOS PRODUCTORES Y CONSUMIDORES CON *BUFFER* ACOTADO USANDO MONITORES (I)



Monitor

```
monitor b; /* buffer limitado */
void productor();
  char item;
  while(true) {
    ...producir un elemento...
    b.insertar(item);
  }
}
```

```
monitor b; /* buffer limitado */
void consumidor();
  char item;
  while(true) {
    b.extraer();
    ...consumir un elemento...
  }
}
```

PROBLEMA DE LOS PRODUCTORES Y CONSUMIDORES CON *BUFFER* ACOTADO USANDO MONITORES (II)

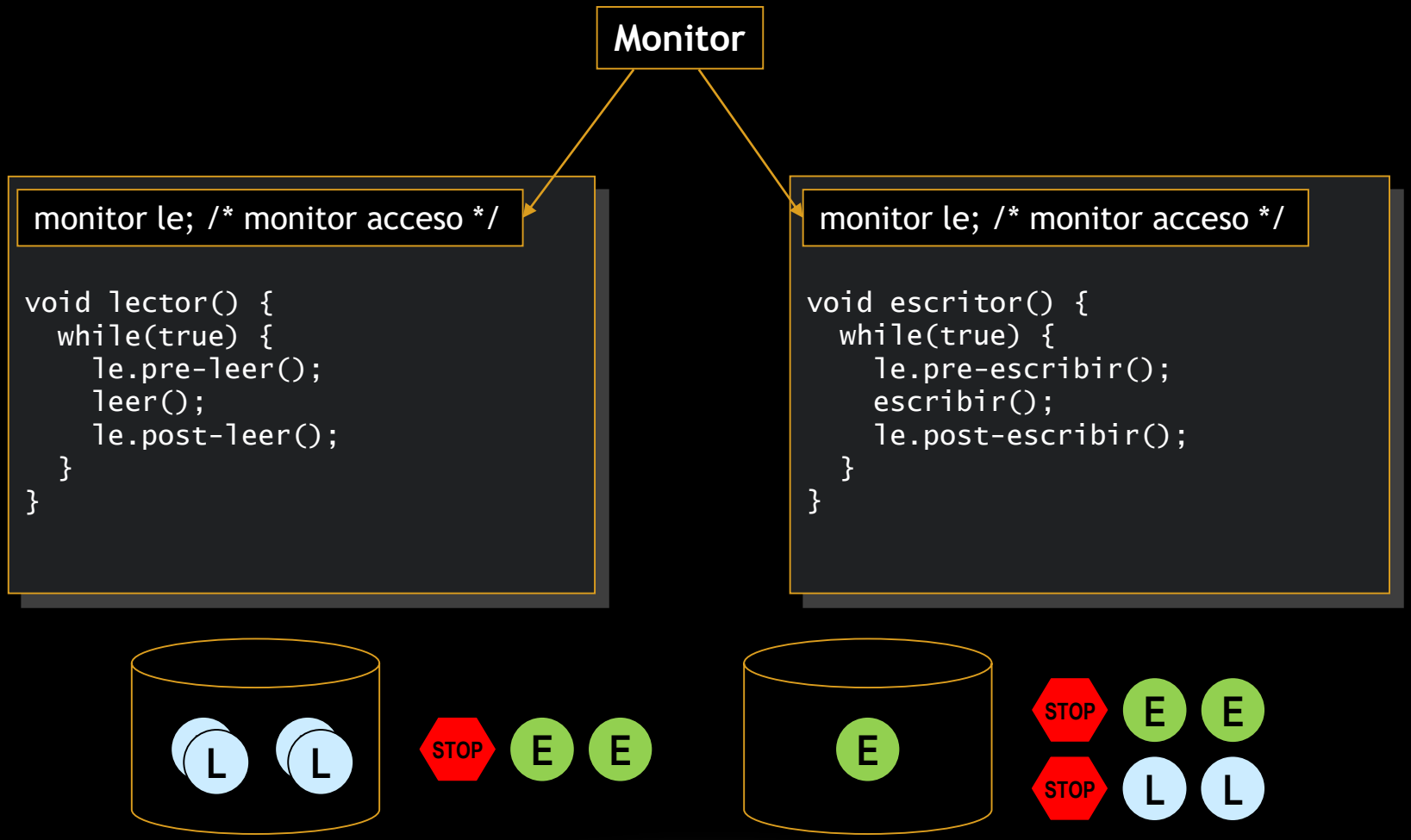
Monitor

```
monitor b {
  char buffer[N];
  int entrada, salida;
  int cont;
  cond lleno, vacio;

  void insertar(char elemento);
     ...

  void extraer(char elemento);
     ...

{
  entrada = 0;
  salida = 0;
  cont = 0;
}
```

```
void insertar(char elemento) {
  if (cont==n)
     lleno.wait;
  cont = cont+1;
  buffer[entrada] = elemento;
  entrada = (entrada+1)%n;
  vacio.signal;
}
```

```
void extraer(char elemento) {
  if (cont==0)
     vacio.wait;
  cont = cont-1;
  elemento = buffer[salida];
  salida = (salida+1)%n;
  lleno.signal;
}
```

PROBLEMA DE LOS LECTORES Y ESCRITORES USANDO MONITORES



PROBLEMA DE LOS LECTORES Y ESCRITORES USANDO MONITORES: PRIMERA VARIANTE

Inanición para escritores

Monitor

```
monitor le {
  int lectores, escritores;
  cond leer, escribir;

 void pre-leer() {
   if (escritores > 0) leer.wait;
   lectores = lectores+1;
   leer.signal; /*despertar al siguiente*/
 }

 void post-leer() {
   lectores = lectores-1;
   if (lectores == 0) escribir.signal;
 }
```

```
 void pre-escribir() {
   if ((escritores>0) || (lectores > 0))
      escribir.wait;
   escritores = escritores+1;
 }

 void post-escribir();
  escritores = escritores-1;
  if (leer.awaited > 0)
    leer.signal;
  else escribir.signal;
 }

{
  lectores = 0;
  escritores = 0;
}
}
```

PROBLEMA DE LOS LECTORES Y ESCRITORES USANDO MONITORES: SEGUNDA VARIANTE

Inanición para lectores

Monitor

```
monitor le {
  int lectores, escritores;
  cond leer, escribir;

 void pre-leer() {
   if (escritores>0 || escribir.awaited>0)
      leer.wait;
   lectores = lectores+1;
   leer.signal; /*despertar al siguiente*/
 }

 void post-leer() {
   lectores = lectores-1;
   if (lectores == 0) escribir.signal;
 }
```

```
 void pre-escribir() {
   if ((escritores>0) || (lectores > 0))
      escribir.wait;
   escritores = escritores+1;
 }

 void post-escribir();
  escritores = escritores-1;
  if (escribir.awaited > 0)
    escribir.signal;
  else leer.signal;
 }

{
  lectores = 0;
  escritores = 0;
}
}
```

PROBLEMA DE LOS LECTORES Y ESCRITORES USANDO MONITORES: TERCERA VARIANTE

Sin inanición

Monitor

```
monitor le {
  int lectores, escritores;
  cond leer, escribir;

 void pre-leer() {
   if (escritores>0 || escribir.awaited>0)
      leer.wait;
   lectores = lectores+1;
   leer.signal; /*despertar al siguiente*/
 }

 void post-leer() {
   lectores = lectores-1;
   if (lectores == 0) escribir.signal;
 }
```

```
 void pre-escribir() {
   if ((escritores>0) || (lectores > 0))
      escribir.wait;
   escritores = escritores+1;
 }

 void post-escribir();
  escritores = escritores-1;
  if (leer.awaited > 0)
    leer.signal;
  else escribir.signal;
 }

{
  lectores = 0;
  escritores = 0;
}
```

CARACTERÍSTICAS DE LOS MONITORES

- **Exclusión mutua:** Garantizada por la estructura del propio monitor.
- **Sincronización:** Concentrada en un único punto del código, en vez de estar repartida entre varios programas o funciones.
- Si el monitor para un recurso es correcto todos los accesos a dicho recurso son correctos.
- **Son sencillos de programar…**
- **…pero requieren soporte del lenguaje de programación:**
  - Ada, c#, Java, Python, Ruby…
  - Lenguajes orientados a objetos.
  - Si no, hay librerías que los proporcionan.