

Tema 2: Algoritmos y mecanismos de sincronización basados en memoria compartida.

2.1. Introducción a la sincronización en memoria compartida

2.2. Semáforos para sincronización

2.3. Monitores como mecanismo de alto nivel

2.4. Soluciones software con espera ocupada para Exclusión Mutua (EM)

2.5. Soluciones hardware con espera ocupada (cerrojos) para EM



2.1. Introducción a la sincronización en memoria compartida

Estudiaremos soluciones para **exclusión mutua y sincronización** basadas en el uso de memoria compartida entre los procesos involucrados.

- **Soluciones de bajo nivel con espera ocupada:** Basadas en programas que usan instrucciones de bajo nivel para lectura/escritura directamente a memoria compartida.
 - Usan bucles para realizar las esperas (**espera ocupada**).
- **Soluciones de alto nivel:** Se diseña una capa software por encima de las soluciones anteriores que ofrece un interfaz para las aplicaciones.
 - Usan **bloqueo de proceso** para realizar las esperas requeridas por la sincronización.

2.1. Introducción a la sincronización en memoria compartida

Soluciones de bajo nivel con espera ocupada

Cuando proceso debe esperar una condición, entra en un **bucle que continuamente comprueba condición (espera ocupada)**. Dos categorías:

- **Soluciones software:** Usan operaciones estándar sencillas de lectura y escritura de datos simples (valores lógicos, enteros, ...) en memoria compartida,
- **Soluciones hardware (cerrojos):** Usan instrucciones máquina específicas del repertorio de los procesadores involucrados.

```
process P0 ;
begin
  while true do begin
    p0sc := true ;
    turno0 := false ;
    while p1sc and not turno0 do
      begin end
    { sección crítica }
    p0sc := false ;
    { resto sección }
  end
end
```

```
{ variables compartidas y valores iniciales }
var sc_ocupada : boolean := false ; { true solo si }

{ procesos }
process P[ i : 1 .. n ];
begin
  while true do begin
    while TestAndSet( sc_ocupada ) do begin end
    { seccion critica }
    sc_ocupada := false ;
    { resto seccion }
  end
end
```

2.1. Introducción a la sincronización en memoria compartida

Soluciones de alto nivel

Las **Soluciones de bajo nivel** se prestan a errores, producen algoritmos complicados y consumen más tiempo de CPU (bucles de espera ocupada).

Soluciones de alto nivel: ofrecen interfaces de acceso a estructuras de datos y usan **bloqueo de procesos** en lugar de espera ocupada. Veremos algunas:

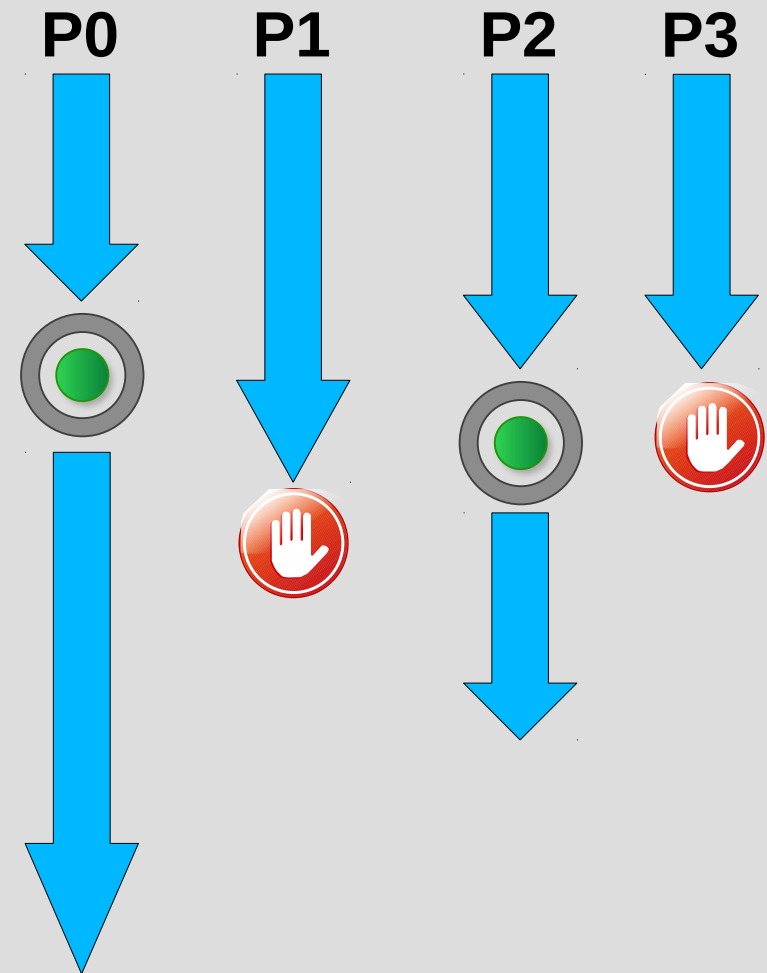
- **Semáforos**: se construyen directamente sobre las soluciones de bajo nivel, usando además servicios del SO que dan la **capacidad de bloquear y reactivar procesos**.
- **Regiones críticas condicionales**: de más alto nivel que los semáforos, y que se pueden implementar sobre ellos.
- **Monitores**: Soluciones de más alto nivel que las anteriores. Se pueden implementar en algunos **lenguajes orientados a objetos** como Java, Python o C++.

2.2. Semáforos para sincronización

Introducción

Semáforos: mecanismo que aminora problemas de soluciones de bajo nivel, y tiene un ámbito de uso más amplio.

- No se usa espera ocupada, sino **bloqueo de procesos** (Más eficiente).
- Resuelven fácilmente la **Exclusión Mutua**.
- Permiten resolver cualquier **problema de sincronización** (aunque esquemas de uso pueden ser complejos).
- Se implementa mediante instancias de una **estructura de datos** accesible únicamente mediante **subprogramas específicos** \Rightarrow aumenta la seguridad y simplicidad.



2.2. Semáforos para sincronización

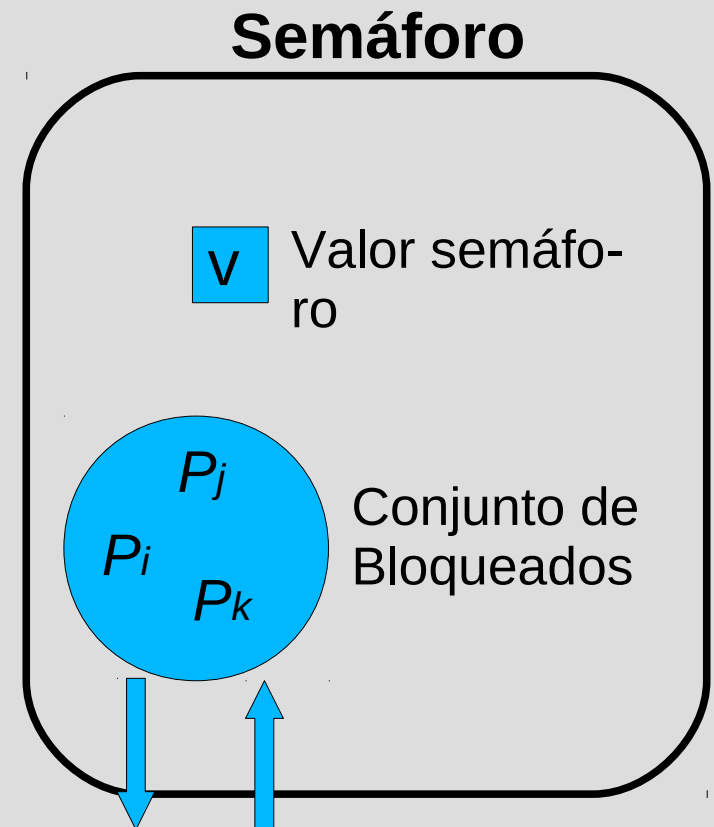
Estructura de un semáforo

Un semáforo es un **instancia de una estructura de datos** que contiene los siguientes elementos en memoria compartida:

- **Conjunto de procesos bloqueados** (“esperando al semáforo”).
- **Valor del semáforo:** valor entero no negativo.

Al **inicio de un programa** que usa un semáforo, debe poder inicializarse:

- Su **conjunto** de procesos esperando estará **vacío**
- Se debe indicar un **valor inicial** del semáforo



2.2. Semáforos para sincronización

Operaciones sobre un semáforo

Además de inicialización, solo hay dos operaciones:

sem_wait(s)

```
If (s.valor == 0)
    Bloquea proceso
s.valor = s.valor - 1;
```

sem_signal(s)

```
s.valor = s.valor + 1;
If (hay procesos esperando s)
    reanuda uno;
```

- **s.valor nunca es negativo**, ya que antes de decrementar se espera a que sea 1.
- Solo puede haber procesos esperando s cuando s.valor es 0.
- Estas operaciones se ejecutan **en exclusión mutua** sobre cada semáforo (excluyendo periodo bloqueo wait),

2.2. Semáforos para sincronización

Invariante de un semáforo

- Dado un **semáforo** s , que se **inicializó a v_0** , con valor v_t en un instante de tiempo, se verifica:

$$v_t = v_0 + n_s - n_w \geq 0$$

donde:

- n_s es el número de llamadas a `sem_signal` completadas y
 - n_w es el número de llamadas a `sem_wait` completadas.
- Los 4 valores son **enteros no negativos**.
- La **igualdad** se deriva de que `sem_signal` siempre incrementa el valor y `sem_wait` siempre lo decrementa (pero espera antes cuando es 0).
- Se mantiene cuando no se está ejecutando `sem_wait` o `sem_signal`.
- Solo **cuentan llamadas a `sem_wait` completadas totalmente** en t .

2.2. Semáforos para sincronización

Patrones de uso sencillos

- **Espera única:** Un proceso, antes de ejecutar una sentencia, debe esperar a que otro proceso complete otra sentencia (típicamente cuando un proceso debe leer una variable escrita por otro proceso).
- **Exclusión Mutua:** acceso a una sección crítica por parte de un número arbitrario de procesos.
- **Productor/Consumidor:** un proceso escribe sucesivos valores en una variable, y cada uno de ellos debe ser leído una única vez por otro proceso.

2.2. Semáforos para sincronización

Espera Única. problema

```
{ variables compartidas y valores iniciales }  
var x          : integer ;           { variable escrita por Productor }  
    puede_leer : semaphore := 0 ; { 1 si x ya escrita y aun no leída }
```

```
process Productor ; { escribe 'x' }  
    var a : integer ;  
begin  
    a := ProducirValor() ;  
    x := a ; { sentencia E }  
end
```

```
process Consumidor { lee 'x' }  
    var b : integer ;  
begin  
    b := x ; { sentencia L }  
    UsarValor(b) ;  
end
```

- Sentencias E y L son atómicas.
- Correctas interfoliaciones en las que E antes que L.

2.2. Semáforos para sincronización

Espera Única. Solución con semáforo

```
{ variables compartidas y valores iniciales }  
var x          : integer ;      { variable escrita por Productor }  
    puede_leer : semaphore := 0; { 1 si x ya escrita y aun no leída }
```

```
process Productor ; { escribe 'x' }  
    var a : integer ;  
begin  
    a := ProducirValor() ;  
    x := a ; { sentencia E }  
    sem_signal( puede_leer ) ;  
end
```

```
process Consumidor { lee 'x' }  
    var b : integer ;  
begin  
    sem_wait( puede_leer ) ;  
    b := x ; { sentencia L }  
    UsarValor(b) ;  
end
```

2.2. Semáforos para sincronización

Exclusión Mutua. Solución con semáforo

```
{ variables compartidas y valores iniciales }
var sc_libre : semaphore := 1 ; { 1 si SC está libre, 0 si SC ocupada }
                                   { (núm. de procs. que pueden entrar a SC) }

process P[ i : 0..n ];
begin
  while true do begin

    { esperar hasta que sc_libre sea 1, entonces ponerla a 0 }
    sem_wait( sc_libre ) ;

    { seccion critica: ..... }

    { poner sc_libre a 1, y desbloquear un proceso en espera si hay alguno }
    sem_signal( sc_libre ) ;

    { resto seccion: ..... }
  end
end
```

2.2. Semáforos para sincronización

Productor-Consumidor. Problema

```
{ variables compartidas y valores iniciales }  
var x : integer ; { contiene cada valor producido y pte. de leer }
```

```
process Productor ; { calcula x }  
var a : integer ;  
begin  
    while true do begin  
        a := ProducirValor() ;  
        x := a ; { sentencia E }  
    end  
end
```

```
process Consumidor { lee x }  
var b : integer ;  
begin  
    while true do begin  
        b := x ; { sentencia L }  
        UsarValor(b) ;  
    end  
end
```

- Son correctas las interfoliaciones en las que E y L se alternan, comenzando en E.

2.2. Semáforos para sincronización

Productor-Consumidor. Solución con semáforos

```
{ variables compartidas y valores iniciales }  
var x          : integer ;           { contiene cada valor producido }  
    puede_leer  : semaphore := 0 ; { 1 se puede leer x, 0 no }  
    puede_escribir : semaphore := 1 ; { 1 se puede escribir x, 0 no }
```

```
process Productor ; { escribe x }  
var a : integer ;  
begin  
    while true do begin  
        a := ProducirValor() ;  
        sem_wait( puede_escribir ) ;  
        x := a ; { sentencia E }  
        sem_signal( puede_leer ) ;  
    end  
end
```

```
process Consumidor { lee x }  
var b : integer ;  
begin  
    while true do begin  
        sem_wait( puede_leer ) ;  
        b := x ; { sentencia L }  
        sem_signal( puede_escribir ) ;  
        UsarValor(b) ;  
    end  
end
```

2.2. Semáforos para sincronización

Limitaciones de los semáforos

Semáforos resuelven fácil y eficientemente problemas de **EM y de sincronización sencilla**. Sin embargo:

- **Difícil resolver problemas complejos de sincronización.**
 - **Basados en variables globales:** impide diseño modular y reduce escalabilidad (incorporar más procesos suele requerir replantear variables globales).
 - **Uso y función de las variables no explícito:** difícil analizar corrección.
 - **Operaciones se encuentran dispersas y no protegidas:** (↑ posibilidad errores).
- **Programas erróneos o malintencionados** pueden provocar bloqueos indefinidos o estados incorrectos.

NECESIDAD: Mecanismo que permita **acceso estructurado y encapsulación**, garantizando **EM** y permitiendo implementar condiciones de **sincronización**.

2.3. Monitores. Un mecanismo de alto nivel

Introducción

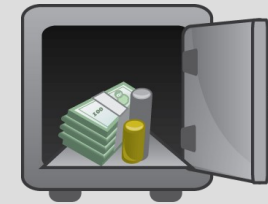
C.A.R. Hoare y Brinch Hansen, idearon el **concepto de Monitor** (1974): mecanismo de alto nivel que permite definir **objetos abstractos compartidos**, que incluyen:

- Colección de **variables encapsuladas** (datos) que representan un recurso compartido por varios procesos.
- Conjunto de **procedimientos** para manipular recurso: afectan variables encapsuladas.

Permiten al programador invocar los procedimientos de forma que:

- **Acceso en EM** a las variables encapsuladas.
- Se implementa **sincronización** requerida por problema mediante **esperas bloqueadas**.

Recurso Compartido



Representación
computacional

Variables Encapsuladas

Int Saldo;
Int ...; ...

Procedimientos

Sacar(c)

Ingresar(c)

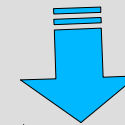
2.3. Monitores. Un mecanismo de alto nivel

Propiedades del monitor

MONITOR → **recurso compartido** usado como objeto **accedido concurrentemente**.

- **Acceso estructurado**: Usuario (proceso) solo puede acceder al recurso mediante un conjunto de operaciones.
- **Encapsulación**: Usuario ignora variables que representan al recurso e implementación procedimientos.
- **Acceso en EM a los procedimientos**
 - Garantizada por definición.
 - Implementación garantiza: **Nunca 2 procesos estarán ejecutando simultáneamente procedimientos.**

Recurso Compartido



Monitor R

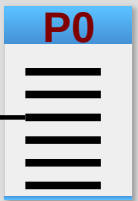
Variables Encapsuladas

Int Saldo;
Int ...; ...

Procedimientos

Sacar(c)

Ingresar(c)



Exclusión Mutua

2.3. Monitores. Un mecanismo de alto nivel

Ventajas sobre los semáforos

Frente a los semáforos, el uso de los **monitores facilitan diseño e implementación seguros**:

- **Variables protegidas**

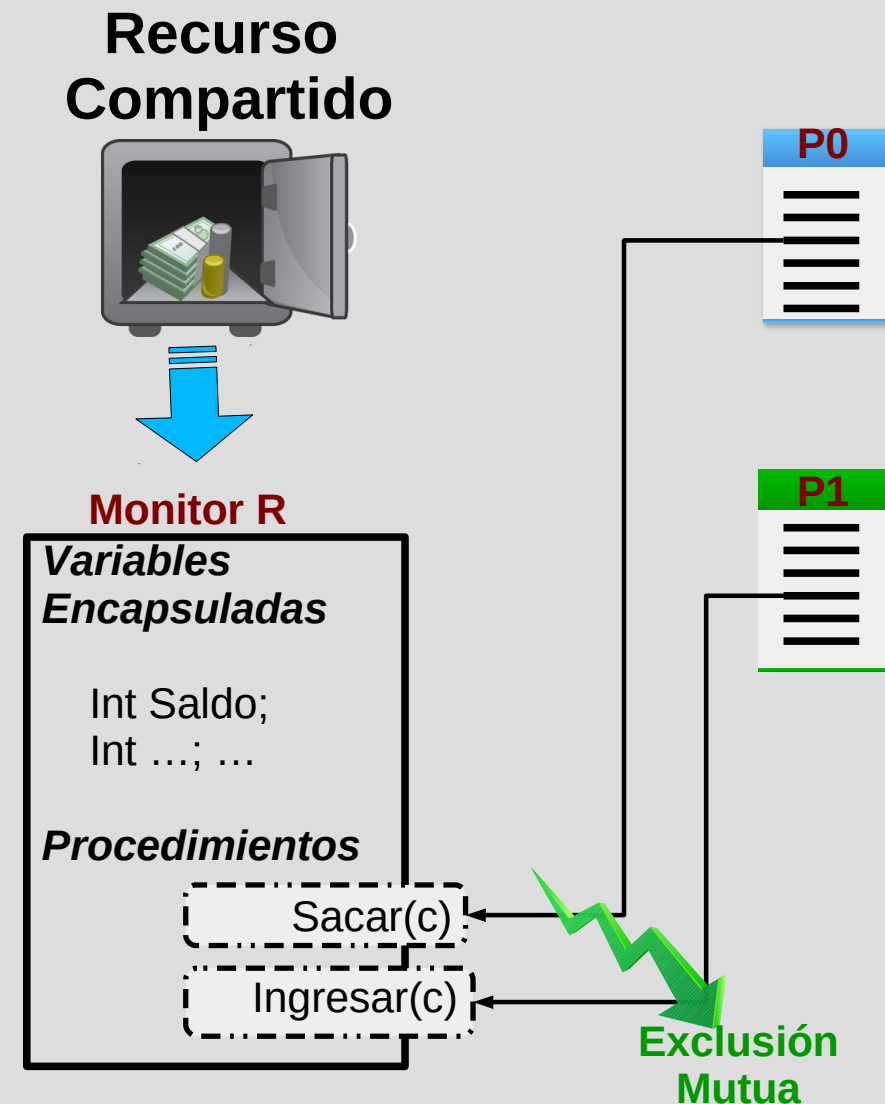
solo pueden leerse o modificarse desde el código del monitor, no desde otro punto programa.

- **EM garantizada**

Programador no tiene que usar mecanismos explícitos de EM para acceso a variables compartidas.

- **Operaciones wait-signal se usan solo dentro monitor**

Más fácil verificar corrección.



2.3. Monitores. Un mecanismo de alto nivel

Componentes de un monitor

Variables permanentes: Estado interno del recurso compartido.

- Sólo accedidas dentro del monitor (en cuerpo de procedimientos y código inicialización).
- Sin modificaciones entre dos llamadas consecutivas a procedimientos.

Procedimientos: Modifican estado (en EM).

- Pueden tener variables locales, que toman un nuevo valor en cada activación del procedimiento.
- Algunos forman interfaz externa del monitor y podrán ser llamados por procesos compartiendo recurso.

Código de inicialización: fija estado inicial.

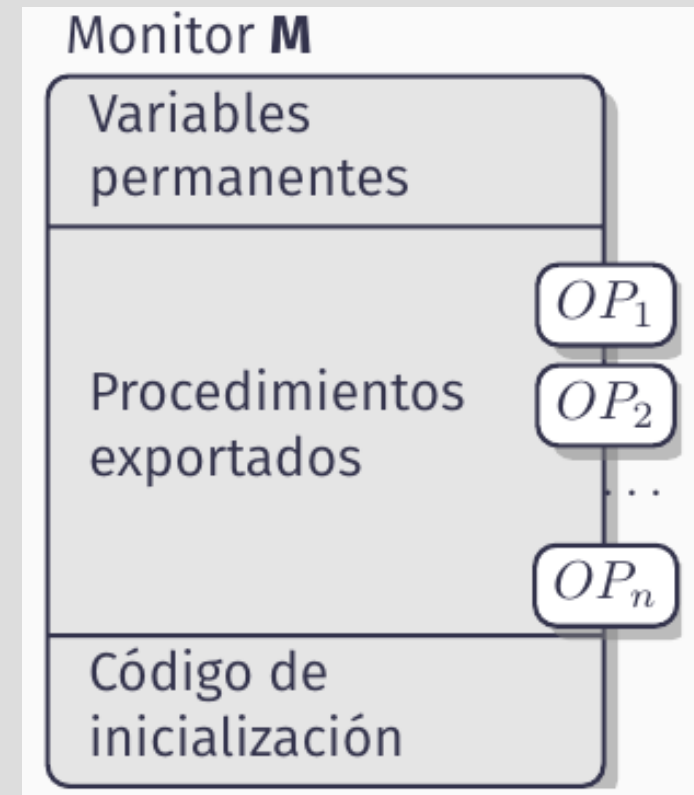
- Se ejecuta una única vez, antes de cualquier llamada a procedimiento.

```
monitor nombre_monitor ;
var
    ..... ;
export
    nom_exp_1,
    nom_exp_2, .... ;
procedure nom_exp_1( ..
    var ..... ;
begin
    ...
end
.....
begin
    ....
end
```

2.3. Monitores. Un mecanismo de alto nivel

Diagrama de componentes de un monitor

- **Interfaz con el exterior:** el uso que se hace del monitor se hace exclusivamente usando los procedimientos exportados.
- **Encapsulación:** Variables permanentes y procedimientos no exportados no son accesibles desde fuera.
- **Ventaja:** Implementación operaciones se puede cambiar sin modificar la semántica.



2.3. Monitores. Un mecanismo de alto nivel

Ejemplo de monitor

Varios procesos pueden incrementar (en una unidad) una variable compartida y examinar su valor en cualquier momento, invocando operaciones.

```
{ declaracion del monitor }
monitor VarCompartida ;

var x : integer; { permanente }
export incremento, valor;

procedure incremento( );
begin
    x := x+1 ; {incrementa valor}
end;

function valor() : integer ;
begin
    result := x; { escribe result.}
end;

begin { código de inicialización}
    x := 0 ; { inicializa valor }
end { fin del monitor}
```

Código Proceso Usuario

```
{ ejemplo de uso del monitor }
{ (debe aparecer en el ámbito }
{ de la declaración)          }

{ incrementar valor: }
VarCompartida.incremento();

{ copiar en k el valor: }
k := VarCompartida.valor() ;
```

2.3. Monitores. Un mecanismo de alto nivel

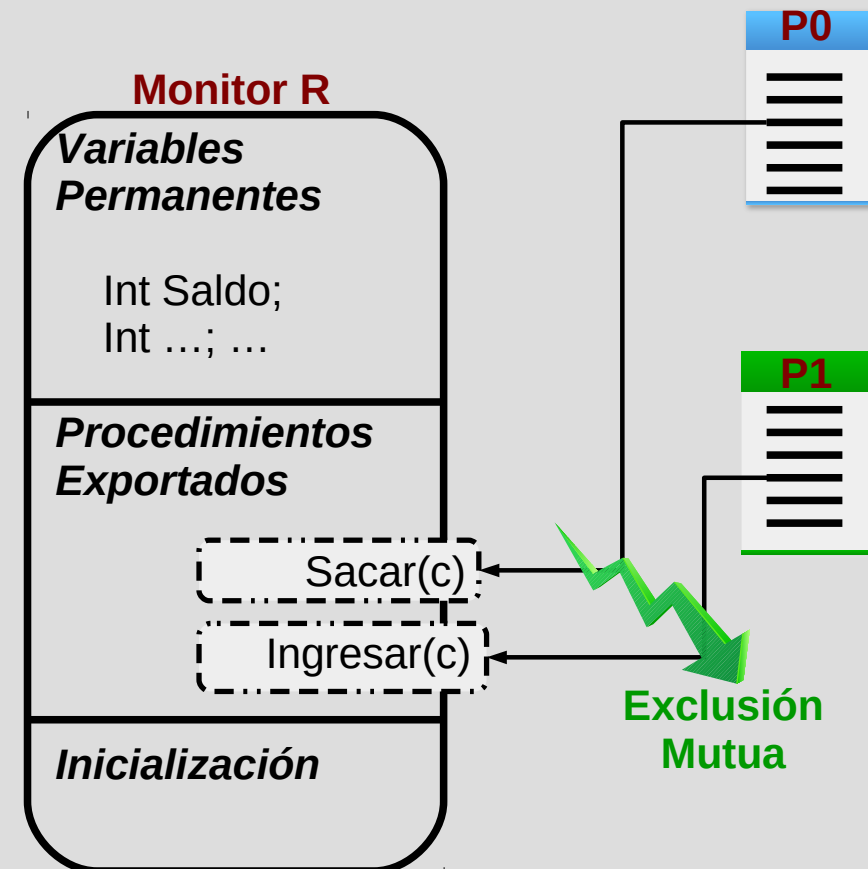
Funcionamiento de los monitores (1)

Comunicación Monitor-Mundo exterior: Cuando un proceso necesita operar sobre recurso compartido controlado por un monitor deberá invocar uno de los procedimientos exportados con los parámetros apropiados.

- Mientras proceso P está ejecutando un procedimiento del monitor decimos que **P está dentro del monitor**.

Exclusión mutua: Si proceso P está dentro monitor, cualquier otro proceso que invoque un procedimiento deberá esperar a que P salga:

- Asegura: **variables permanentes nunca son accedidas concurrentemente**.
- Debe garantizarse en la implementación del monitor.



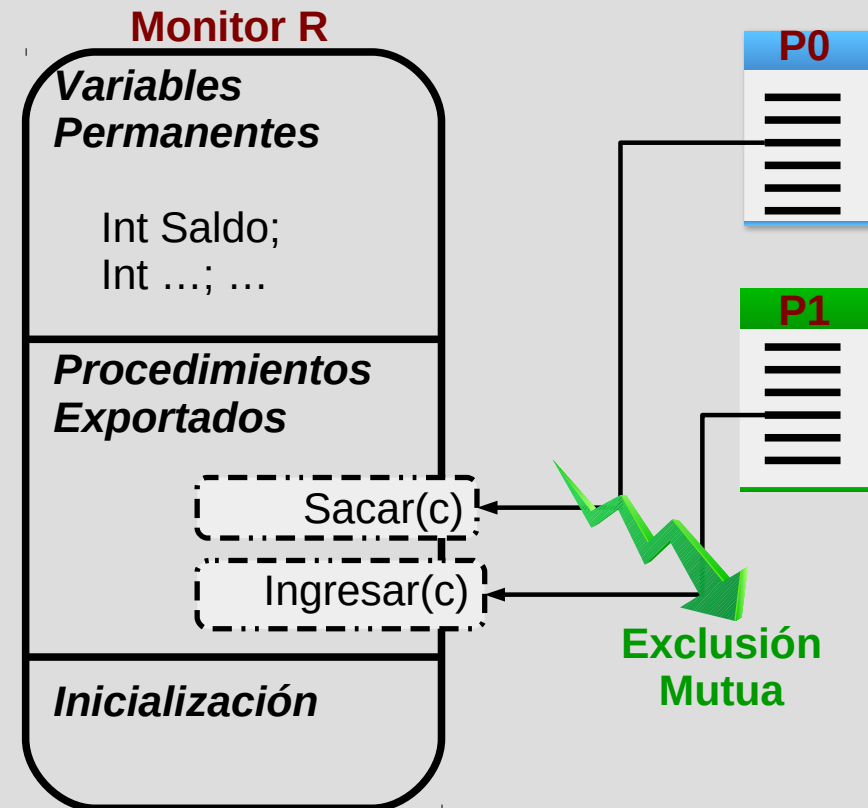
2.3. Monitores. Un mecanismo de alto nivel

Funcionamiento de los monitores (2)

Son objetos pasivos: Tras ejecutar código de inicialización, un monitor es un objeto pasivo y sus procedimientos sólo se ejecutan cuando son invocados.

Instanciación de clases de monitores: En algunos casos es conveniente crear múltiples instancias independientes de un tipo de monitor:

- Cada instancia tiene sus variables permanentes propias.
- La E.M. ocurre en cada instancia por separado.



```
class monitor nombre_clase_monitor( parametros_formales ) ;  
    .... { cuerpo del monitor semejante a los anteriores }  
end  
var nombre_instancia_1 : nombre_clase_monitor( parametros_actuales_1 ) ,  
    nombre_instancia_2 : nombre_clase_monitor( parametros_actuales_2 ) ;
```


2.3. Monitores. Un mecanismo de alto nivel

Instanciación de una clase de monitor. Ejemplo

{ declaracion de la clase monitor }

```
class monitor VarComp(pini,pinc : integer)
```

```
    var x, inc : integer ;  
    export incremento, valor;
```

```
    procedure incremento( );  
    begin
```

```
        x := x+inc ;
```

```
    end;
```

```
    function valor(): integer ;  
    begin
```

```
        result := x ;
```

```
    end;
```

```
begin
```

```
    x:= pini ; inc := pinc ;
```

```
end
```

{ ejemplo de uso }

```
var mv1    : VarComp(0,1);
```

```
    mv2    : VarComp(10,4);
```

```
    i1,i2 : integer ;
```

```
begin
```

```
    mv1.incremento();
```

```
    i1:= mv1.valor();{ i1 == 1 }
```

```
    mv2.incremento();
```

```
    i2:= mv2.valor();{ i2 == 14 }
```

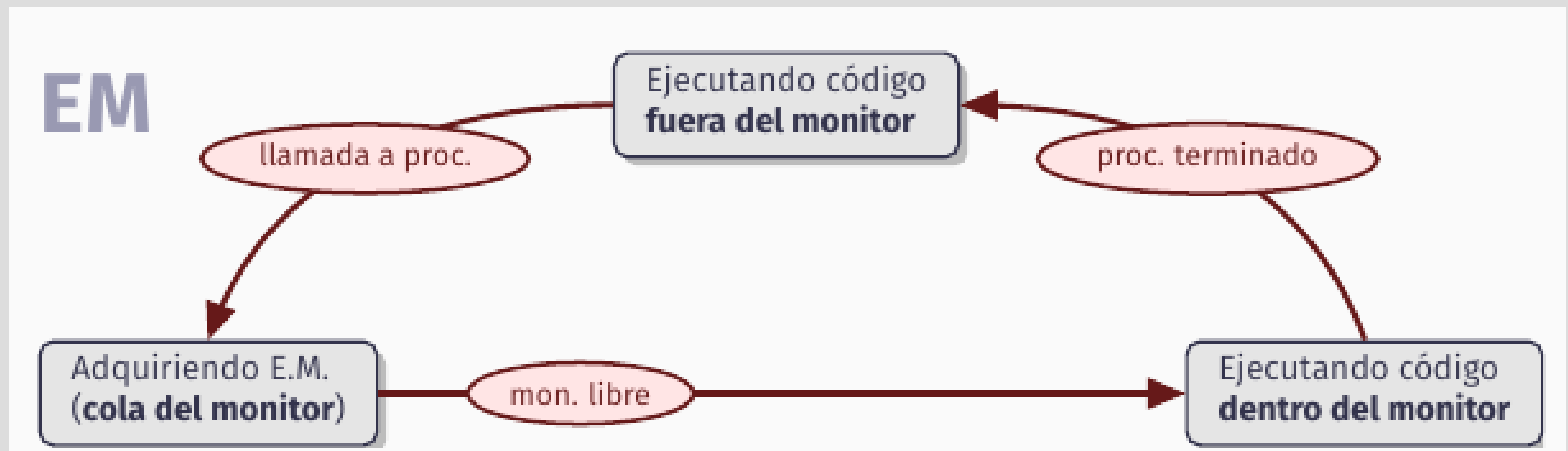
```
end
```

2.3. Monitores. Un mecanismo de alto nivel

Cola del monitor para EM

Control de la EM se basa en existencia de una **Cola del monitor**:

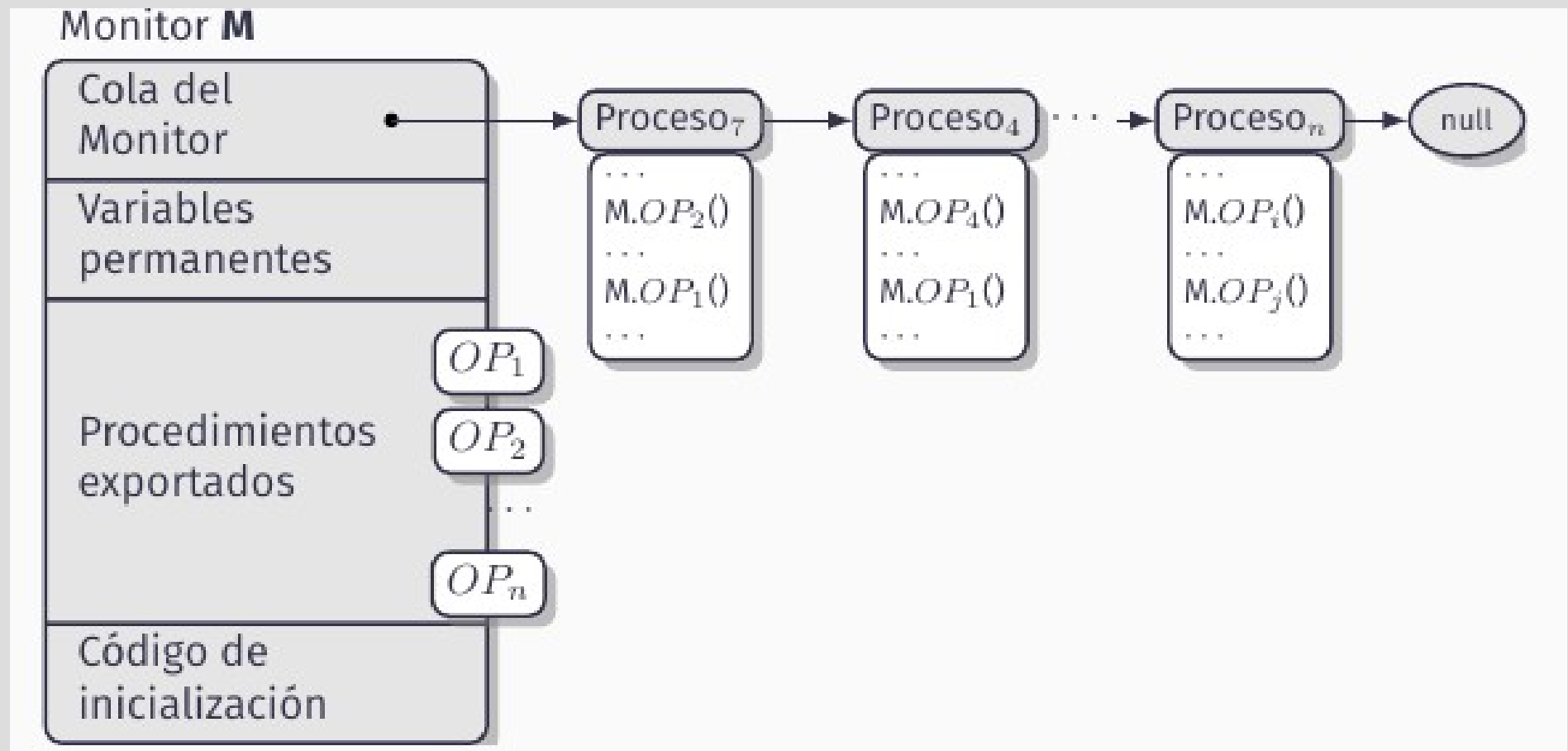
- Si **proceso P dentro monitor** y otro proc. Q intenta ejecutar procedimiento, Q queda bloqueado, insertándose en la cola del monitor
- **Proceso abandona el monitor** (finaliza ejec. procedimiento) \Rightarrow Se desbloquea un proceso de la cola, y puede entrar.
- **Cola vacía** \Rightarrow Monitor libre. Primer proceso que invoque procedimiento, entrará.
- Planificación de la cola debe seguir una **política FIFO** \Rightarrow Garantiza **vivacidad**.



2.3. Monitores. Un mecanismo de alto nivel

Estado del Monitor

Estado del monitor: incluye la cola de procesos esperando ejecución procedim.



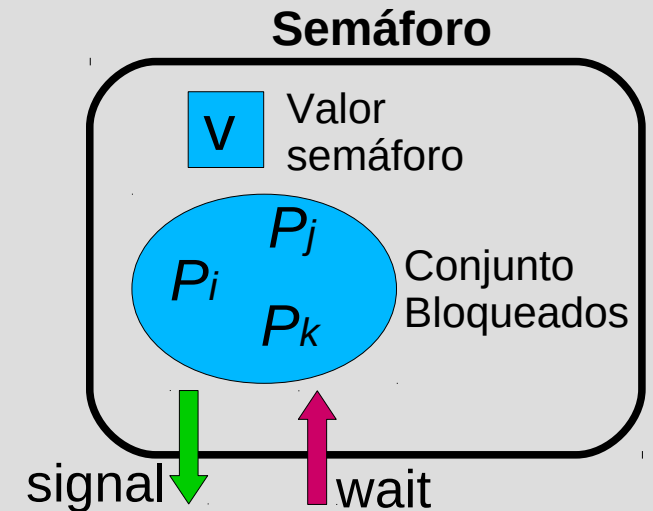
2.3. Monitores. Un mecanismo de alto nivel

Sincronización con monitores

Implementar sincronización requiere permitir **esperas bloqueadas** en los procesos, hasta que una condición sea cierta:

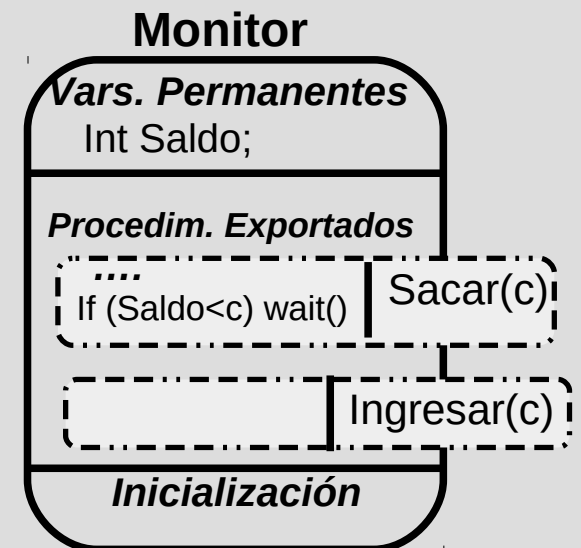
Semáforos

- **Bloqueo (sem_wait) y activación (sem_signal) + Valor del semáforo** (indica si condición se cumple o no).



Monitores

- Sentencias de **bloqueo y activación** sin valor.
- **Los valores de las variables permanentes** determinan si la condición se cumple.

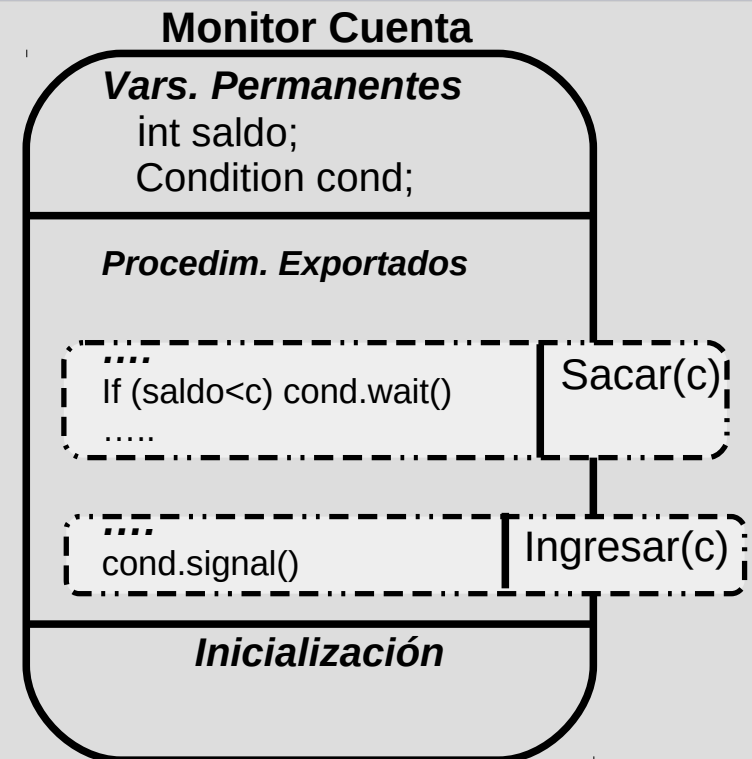


2.3. Monitores. Un mecanismo de alto nivel

Bloqueo y activación con variables condición

Para cada condición de espera distinta, se debe de declarar una variable permanente de tipo **condition**. A esas variables las llamamos **señales o variables condición**:

- Cada **variable condition** tiene asociada una **cola de procesos esperando condición** cierta.
- Sobre una var. condition **cond**, un proceso puede invocar **dos operaciones**:
 - **cond.wait()**: Espero a que alguna condición ocurra.
 - **cond.signal()**: Señalo que condición ocurre.

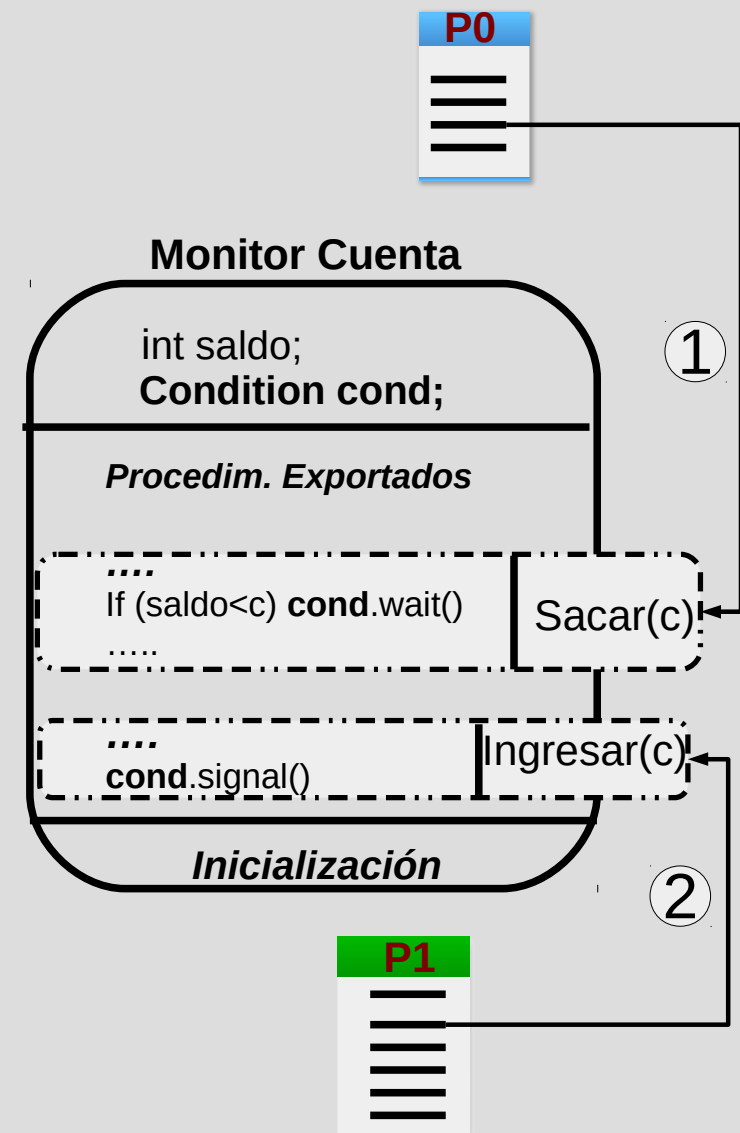


2.3. Monitores. Un mecanismo de alto nivel

Bloqueo y activación con variables condición

Dada una **variable condición cond**, se definen:

- **cond.wait()**: Bloquea al proceso invocador y lo introduce en la cola de la variable cond.
- **cond.signal()**: si hay procesos bloqueados en cond, libera uno de ellos.
 - **Política FIFO**: Reactivará al que lleve más tiempo esperando.
Evita inanición: cada proceso en cola obtendrá eventualmente turno.
- **cond.queue()**: **true** si cola de condición no vacía, y **false** en caso contrario.

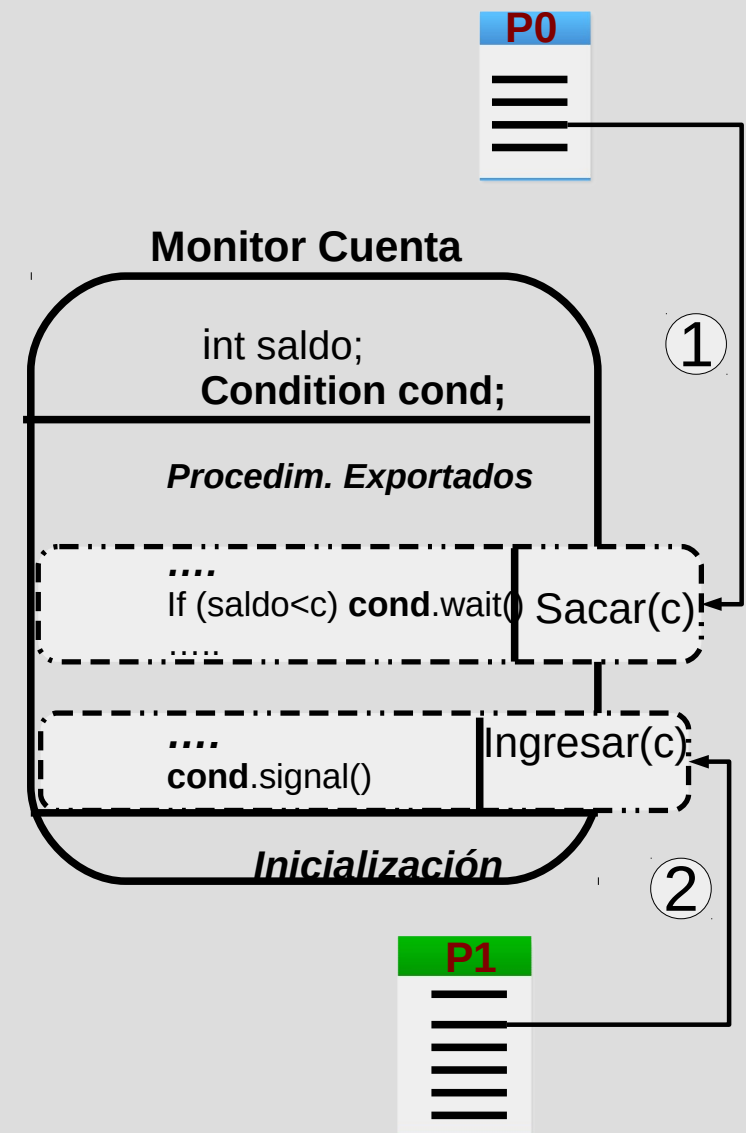


2.3. Monitores. Un mecanismo de alto nivel

Esperas bloqueadas y E.M. en el monitor

Los procesos pueden estar dentro del monitor, pero bloqueados:

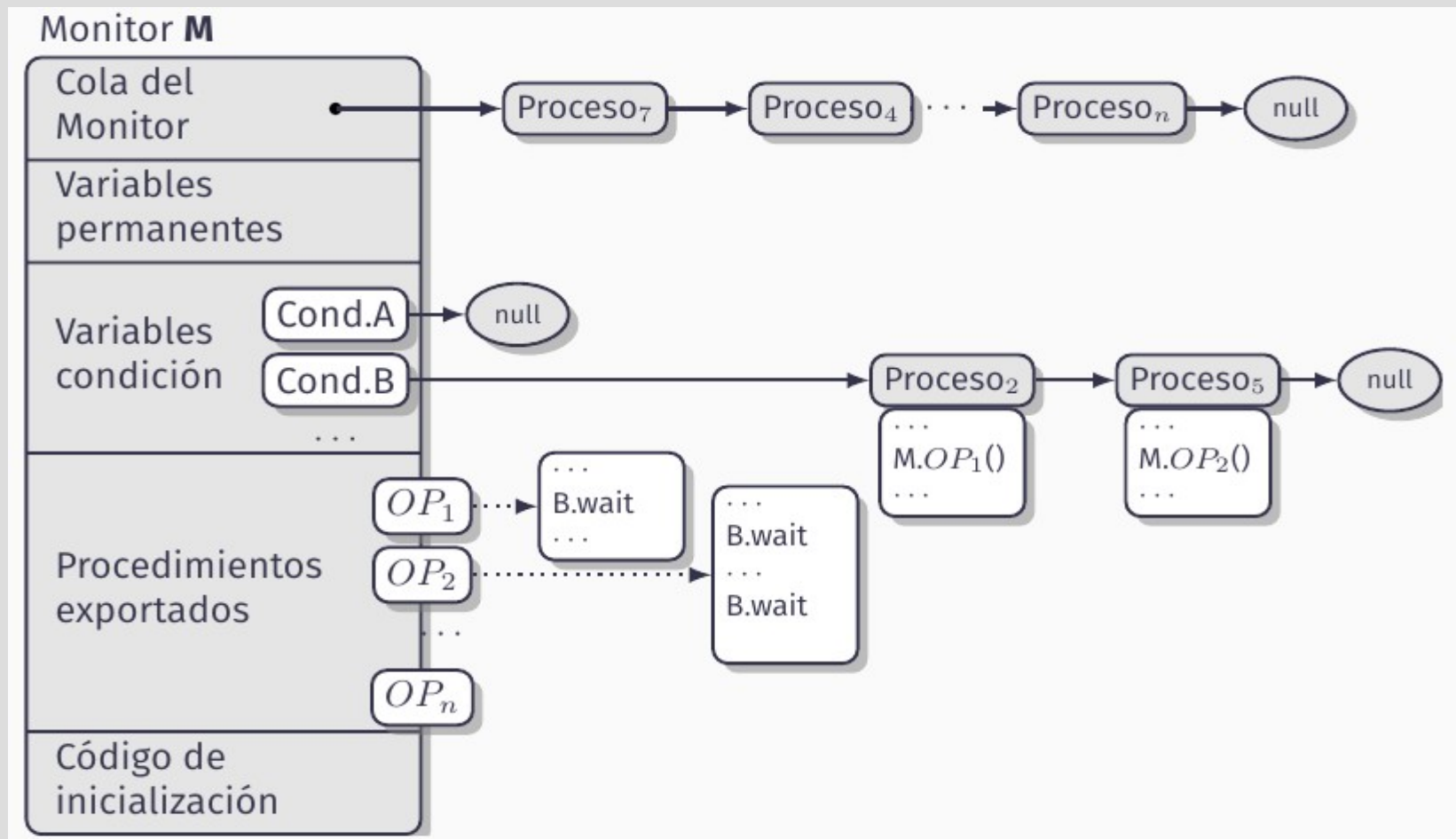
- Cuando **proceso llama a wait** y queda bloqueado, **se libera EM del monitor**.
 - En caso contrario \Rightarrow **interbloqueo** (proceso quedaría bloqueado y el resto al intentar entrar).
- Cuando **proceso es reactivado**, adquiere EM antes de ejecutar la sentencia siguiente a wait.
- **Más de un proceso podrá estar dentro del monitor**, aunque **solo uno de ellos estará ejecutándose**, el resto estarán bloqueados en colas de condición.



2.3. Monitores. Un mecanismo de alto nivel

Estado de un monitor con varias colas

Supongamos: los procesos 2 y 5 ejecutan las operaciones OP1 y OP2, ambas producen esperas de la condición B.



2.3. Monitores. Un mecanismo de alto nivel

Espera única. Interacción entre los procesos

Monitor EU se puede usar para sincronizar la lectura y escritura de una variable compartida, de esta forma:

```
{ variables compartidas }  
var x : integer ; { contiene cada valor producido }
```

```
process Productor ; { escribe x }  
var a : integer ;  
begin  
    a := ProducirValor() ;  
    x := a ;                { sentencia E }  
    EU.notificar() ; { sentencia N }  
end
```

```
process Consumidor { lee x }  
var b : integer ;  
begin  
    EU.esperar() ; { sentencia W }  
    b := x ;        { sentencia L }  
    UsarValor(b) ;  
end
```

2.3. Monitores. Un mecanismo de alto nivel

Espera única. Monitor EU

```
monitor EU ;                { Monitor de Espera Única (EU) }

var terminado : boolean;    { true si se ha terminado E, false sino }
    cola      : condition;  { cola consumidor esperando terminado==true }
export esperar, notificar; { nombra procedimientos públicos }

procedure esperar();        { para llamar antes de L }
begin
    if not terminado then { si no se ha terminado E }
        cola.wait();      { esperar hasta que termine }
    end
procedure notificar();      { para llamar después de E }
begin
    terminado := true;      { registrar que ha terminado E }
    cola.signal();          { reactivar el otro proceso, si espera }
end
begin
    { inicializacion: }
    terminado := false;    { al inicio no ha terminado E }
end
```

2.3. Monitores. Un mecanismo de alto nivel

Exclusión Mutua. Interacción entre los procesos

```
process Usuario[ i : 0..n ]  
begin  
  while true do begin  
    EM.entrar(); { esperar SC libre, registrar SC ocupada }  
    .....      { sección crítica }  
    EM.salir();  { registrar SC libre, señalar }  
    .....      { otras actividades (RS) }  
  end  
end
```

2.3. Monitores. Un mecanismo de alto nivel

Exclusión Mutua. Monitor EM

```
monitor EM ;

var ocupada : boolean ; { true hay un proceso en SC, false sino }
    cola      : condition; { cola de procesos esperando ocupada==false}
export entrar, salir ;   { nombra procedimientos públicos }

procedure entrar();      { protocolo de entrada (sentencia E)}
begin
    if ocupada then      { si hay un proceso en la SC }
        cola.wait();     { esperar hasta que termine }
        ocupada := true;  { indicar que la SC está ocupada }
    end
procedure salir();      { protocolo de salida (sentencia S)}
begin
    ocupada := false;    { marcar la SC como libre }
    cola.signal();       { si al menos un proceso espera, reactivar uno }
end

begin                  { inicializacion: }
    ocupada := false;   { al inicio no hay procesos en SC }
end
```

2.3. Monitores. Un mecanismo de alto nivel

Productor-Consumidor. Sincronización con monitor PC

```
process Productor ; { calcula x }  
  var a : integer ;  
begin  
  while true do begin  
    a := ProducirValor() ;  
    PC.escribir(a);{ copia a en valor}  
  end  
end
```

```
process Consumidor { lee x }  
  var b : integer ;  
begin  
  while true do begin  
    PC.leer(b); { copia valor en b }  
    UsarValor(b) ;  
  end  
end
```

- El procedimiento **escribir** escribe valor en variable compartida.
- El procedimiento **leer** lee el valor que hay en la variable.

2.3. Monitores. Un mecanismo de alto nivel

Productor-Consumidor. Monitor PC

```
Monitor PC ;  
var valor_com : integer ;    { valor compartido }  
    pendiente : boolean ;    { true solo si hay valor escrito y no leído }  
    cola_prod : condition ;  { espera productor hasta que pendiente == false }  
    cola_cons : condition ;  { espera consumidor hasta que pendiente == true }
```

```
procedure escribir( v : integer );  
begin  
    if pendiente then  
        cola_prod.wait();  
    valor_com := v ;  
    pendiente := true ;  
    cola_cons.signal();  
end
```

```
function leer() : integer ;  
begin  
    if not pendiente then  
        cola_cons.wait();  
    result := valor_com ;  
    pendiente := false ;  
    cola_prod.signal();  
end
```

```
begin { inicialización }  
    pendiente := false ;  
end
```

2.3. Monitores. Un mecanismo de alto nivel

Colas de Condición con prioridad

Por defecto → colas de espera FIFO.

A veces resulta útil **dar prioridad a unos procesos sobre otros**, aportando un parámetro entero al invocar wait. **Sintaxis:**

cond.wait(p)

- **p**: entero no negativo que refleja la prioridad (cuanto menor mayor prioridad).
- **cond.signal()**: reanudará un proceso que especificó el valor mínimo de p de entre los que esperan (si hay más de uno, se usa política FIFO).
- Se deben **evitar riesgos** como la inanición.
- **No tiene efecto sobre la corrección** del programa: el funcionamiento es similar con/sin prioridad.
- **Sólo mejoran las características dependientes del tiempo.**

2.3. Monitores. Un mecanismo de alto nivel

Ejemplo Colas con prioridad. Asignador recurso

Los procesos, cuando adquieren un recurso, especifican un valor entero de duración de uso del recurso. Se **prioriza a los que requieran menos tiempo**.

```
monitor RecursoPrio;  
  var  ocupado : boolean ;  
       cola    : condition ;  
  export adquirir, liberar ;
```

```
procedure adquirir(tiempo: integer);  
begin  
  if ocupado then  
    cola.wait( tiempo );  
    ocupado := true ;  
  end
```

```
procedure liberar() ;  
begin  
  ocupado := false ;  
  cola.signal();  
end
```

```
{ inicialización }  
begin  
  ocupado := false ;  
end
```


2.3. Monitores. Un mecanismo de alto nivel

Ejemplo Colas con prioridad. Reloj con Alarma

```
Monitor Despertador;  
  var   ahora      : integer ;   { instante actual }  
        despertar  : condition ; { procesos esperando a su hora }  
  export despertame, tick ;
```

```
procedure despertame( n: integer );  
  var alarma : integer;  
begin  
  alarma := ahora + n;  
  while ahora < alarma do  
    despertar.wait( alarma );  
    despertar.signal();  
    { por si otro proceso  
      coincide en la alarma }  
end
```

```
{ un proceso ejecuta esto }  
{ regularmente, tras cada }  
{ unidad de tiempo }
```

```
procedure tick();  
begin  
  ahora := ahora+1 ;  
  despertar.signal();  
end
```

```
{ Inicialización }  
begin  
  ahora := 0 ;  
end
```

2.3. Monitores. Un mecanismo de alto nivel

El Problema de los Lectores/Escritores (LE)

Dos tipos de procesos acceden concurrentemente a datos compartidos:

- **Escritores:** Escriben en la Estructura de Datos (ED). El código de escritura no puede ejecutarse concurrentemente con otra escritura ni lectura, ya que ponen la ED en un estado no usable por otros procesos.
- **Lectores:** Leen la ED, pero no modifican su estado. La lectura puede ejecutarse concurrentemente por varios lectores de forma arbitraria, pero no con la escritura.

Solución: compleja con semáforos, pero sencilla con monitores.



2.3. Monitores. Un mecanismo de alto nivel

El Problema LE. solución con monitor

```
process Lector[ i:1..n ] ;  
begin  
  while true do begin  
    .....  
    Lec_Esc.ini_lectura() ;  
    { código de lectura: .... }  
    Lec_Esc.fin_lectura() ;  
    .....  
  end  
end
```

```
process Escritor[ i:1..m ] ;  
begin  
  while true do begin  
    .....  
    Lec_Esc.ini_escritura() ;  
    { código de escritura: .... }  
    Lec_Esc.fin_escritura() ;  
    .....  
  end  
end
```

- Esta implementación da **prioridad a los lectores** (en el momento que un escritor termina, si hay escritores y lectores esperando, pasan los lectores).
- Hay otras opciones: prioridad a escritores, prioridad al que más tiempo lleva esperando.

2.3. Monitores. Un mecanismo de alto nivel

El Problema LE. Descripción Monitor Lec_Esc (1)

```
monitor Lec_Esc ;

var n_lec      : integer;  { numero de lectores leyendo }
    escrib     : boolean;  { true si hay algun escritor escribiendo }
    lectura    : condition; { no hay escrit. escribiendo, lectura posible }
    escritura  : condition; { no hay lect. ni escrit., escritura posible }

export ini_lectura, fin_lectura,      { invocados por lectores }
       ini_escritura, fin_escritura ; { invocados por escritores }
```

```
procedure ini_lectura()
begin
    if escrib then { si hay escritor: }
        lectura.wait(); { esperar }
    { registrar un lector más }
    n_lec := n_lec + 1 ;
    { desbloqueo en cadena de }
    { posibles lectores bloqueados }
    lectura.signal()
end
```

```
procedure fin_lectura()
begin
    { registrar un lector menos }
    n_lec := n_lec - 1 ;
    { si es el ultimo lector: }
    { desbloquear un escritor }
    if n_lec == 0 then
        escritura.signal()
    end
```

2.3. Monitores. Un mecanismo de alto nivel

El Problema LE. Descripción Monitor Lec_Esc (2)

```
procedure ini_escritura()  
begin  
  { si hay otros, esperar }  
  if n_lec > 0 or escribiendo then  
    escritura.wait()  
  { registrar que hay un escritor }  
  escribiendo := true;  
end;
```

```
procedure fin_escritura()  
begin  
  { registrar que ya no hay escritor }  
  escribiendo := false;  
  { si hay lectores, despertar uno }  
  { si no hay, despertar un escritor }  
  if lectura.queue() then  
    lectura.signal();  
  else  
    escritura.signal();  
end;
```

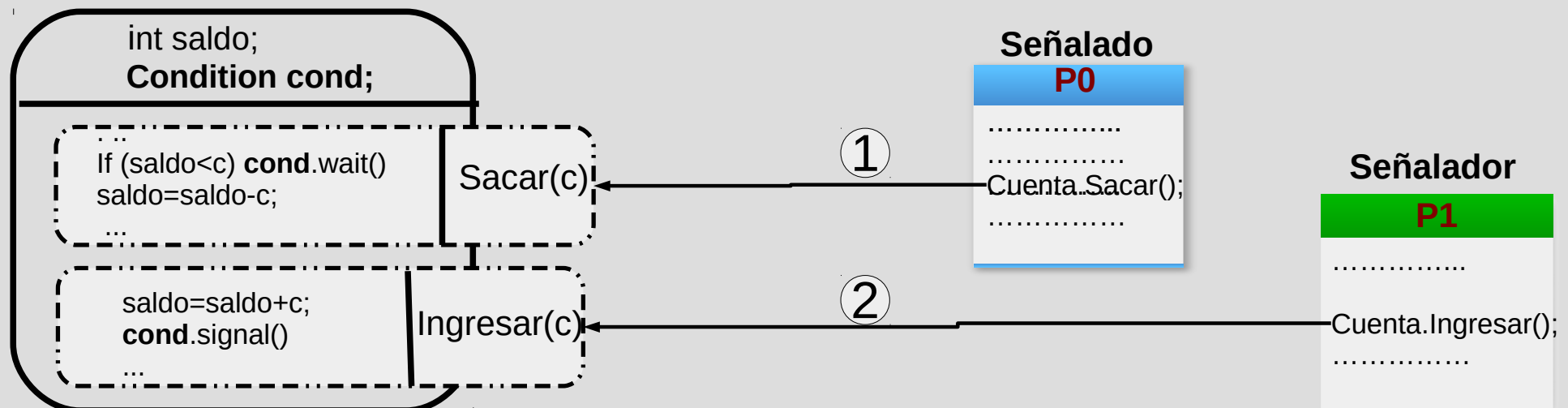
```
begin { inicializacion }  
  n_lec      := 0 ;  
  escribiendo := false ;  
end
```

2.3. Monitores. Un mecanismo de alto nivel

Semántica de las señales de los monitores

Supongamos que proceso invoca **signal sobre cola no vacía** (proc. **señalador**) y reactiva a un proceso que esperaba tras un wait (proc. **señalado**):

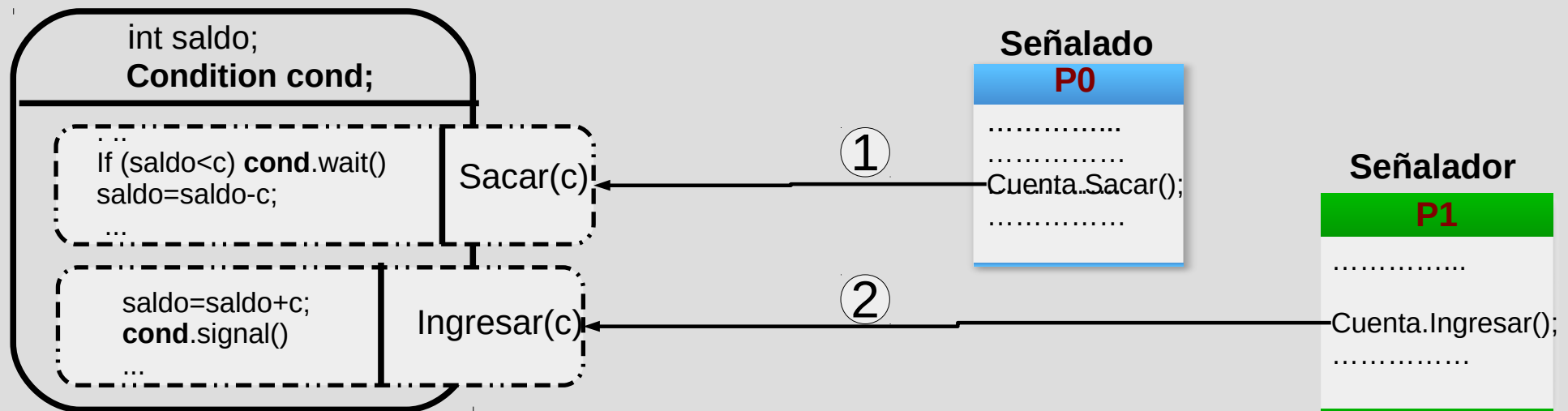
- Suponemos que **hay código restante** tras ese wait y ese signal.
- **Al finalizar el signal, solo uno de los procs. puede continuar** ejecutando su código restante. En otro caso, se violaría la **EM del monitor**.
- **Semántica de señales**: política concreta para resolver conflicto tras signal.



2.3. Monitores. Un mecanismo de alto nivel

Posibles semánticas de las señales. Esquema general

- **Proceso señalador continúa la ejecución tras signal.** Señalado se bloquea hasta adquirir E.M. de nuevo (**SC: Señalar y Continuar**).
- **Proceso señalado se reactiva inmediatamente.**
 - Señalador abandona monitor tras signal (**SS: Señalar y Salir**).
 - Señalador queda bloqueado en:
 - Cola del monitor (EM) (**SE: Señalar y Esperar**).
 - Cola con máxima prioridad (**SU: Señalar y espera Urgente**).



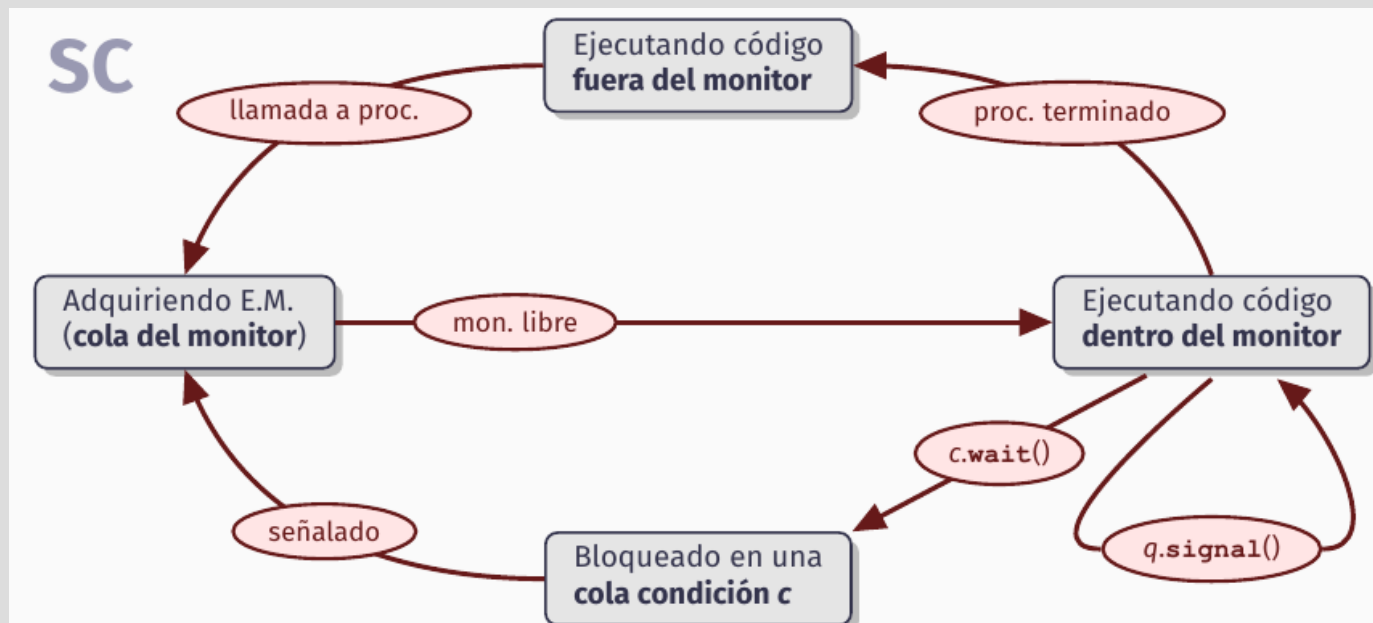
2.3. Monitores. Un mecanismo de alto nivel

Semántica de las señales. Señalar y Continuar (SC)

Señalador continúa ejecución dentro del monitor después signal.

- Señalado abandona cola condición y espera en cola monitor a readquirir E.M.
- Tanto señalador como otros procesos pueden hacer falsa condición de reactivación.
- En proceso señalado no se puede garantizar que la condición asociada a **cond** sea cierta al terminar `cond.wait()`. Esta semántica obliga a realizar wait en un bucle:

```
while not condicion_lógica_desbloqueo do  
    cond.wait() ;
```

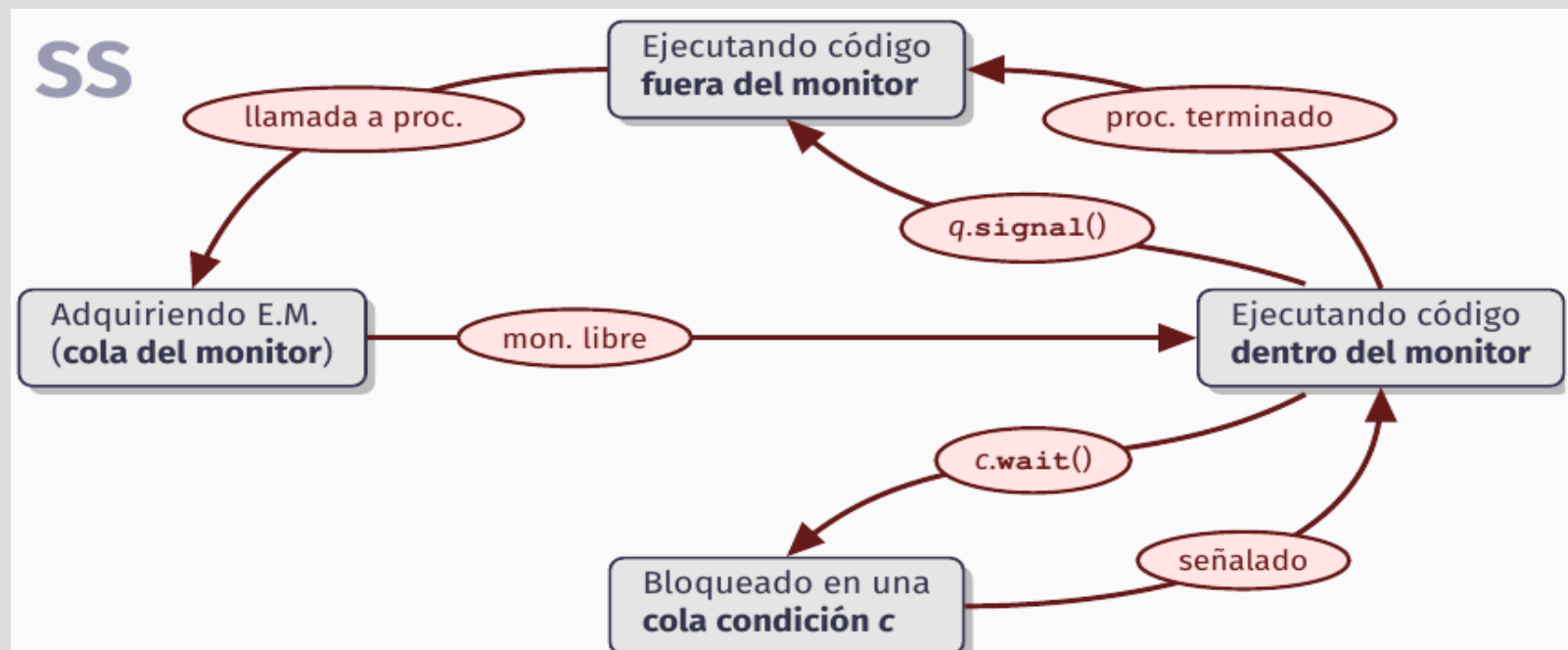


2.3. Monitores. Un mecanismo de alto nivel

Semántica de las señales. Señalar y Salir (SS)

Señalador sale del monitor después cond.signal(). Si hay código tras signal, no se ejecuta. El proceso **señalado reanuda inmediatamente ejecución.**

- Operación **signal** conlleva: **a)** Liberación Señalado y **b)** Terminación procedimiento que ejecutaba Señalador.
- **Se cumple condición de activación señalador:** Se asegura el estado que permite al Señalado continuar ejecución del procedimiento donde se bloqueó.
- Obliga a **colocar signal como última instrucción** procedimientos monitor.

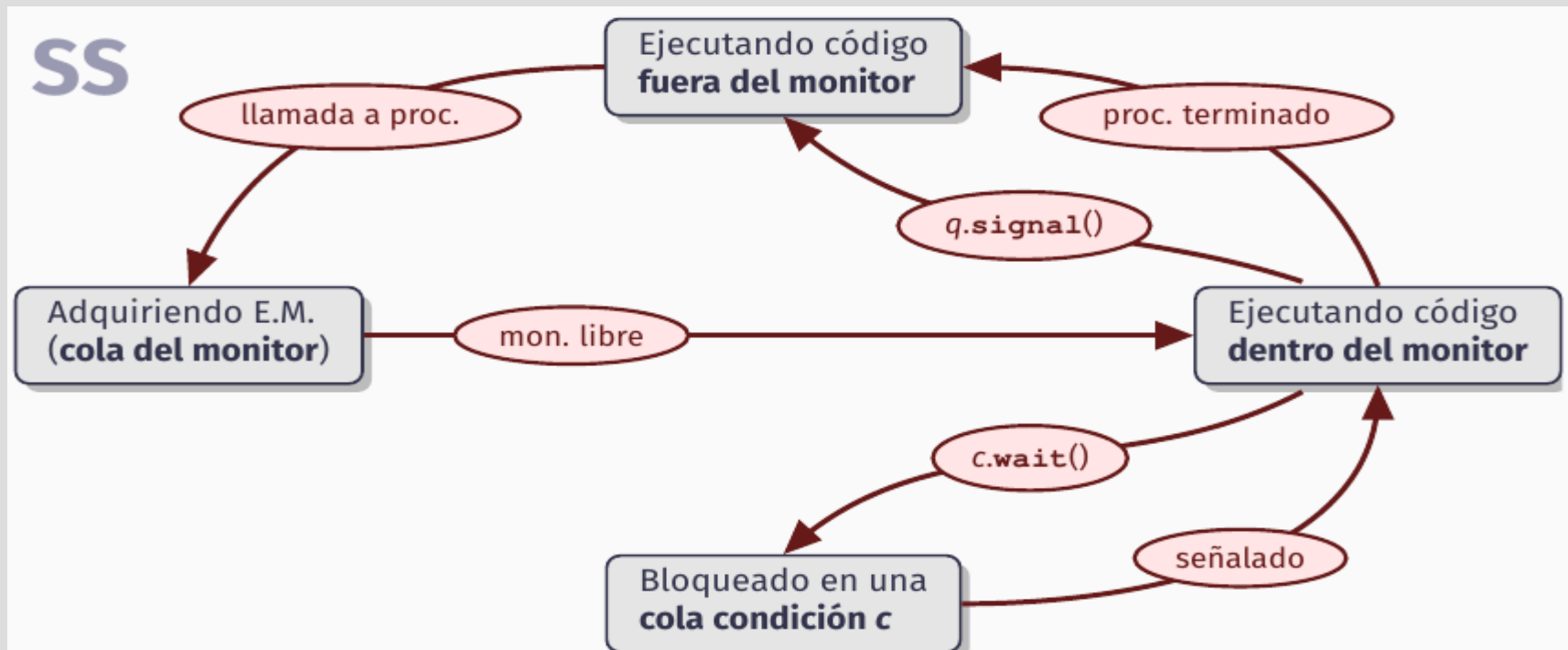


2.3. Monitores. Un mecanismo de alto nivel

Semántica de las señales. Señalar y Esperar (SE)

Señalador se bloquea en la cola del monitor justo después **signal** y **Señalado reanuda** inmediatamente.

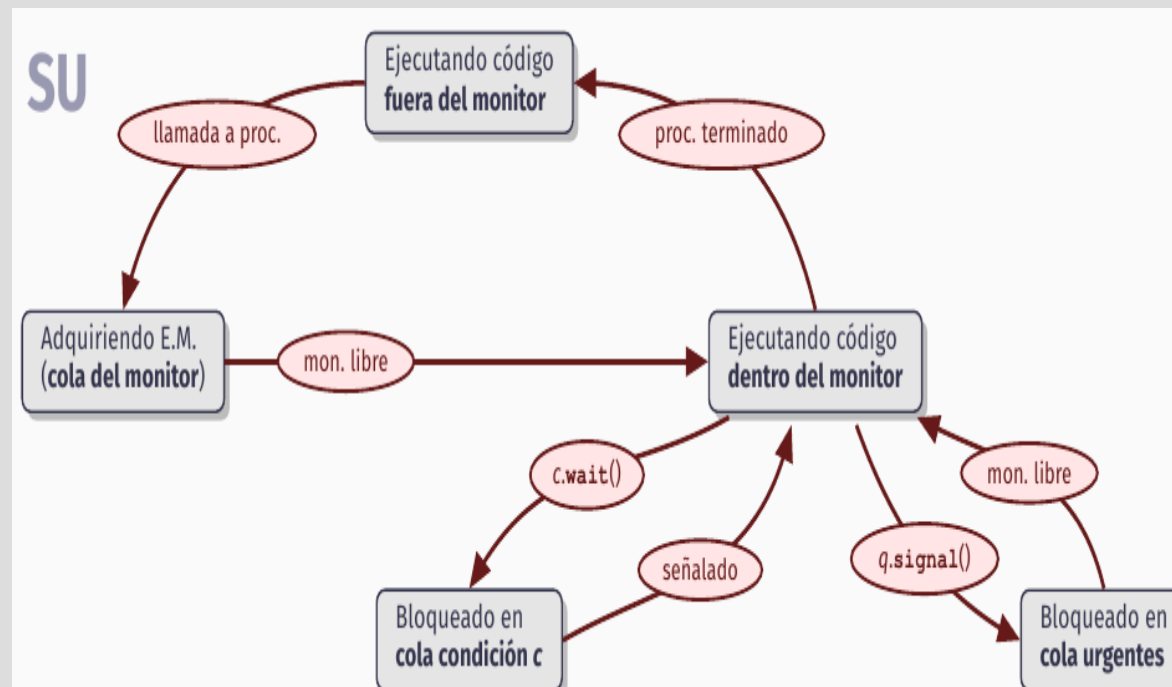
- Se garantiza **condición de activación** para el señalado.
- **Señalador debe competir** por EM con resto procesos.
 - Semántica **injusta respecto al proceso señalador** (ya había obtenido el acceso).



2.3. Monitores. Un mecanismo de alto nivel

Semántica de las señales. Señalar y Espera Urgente (SU)

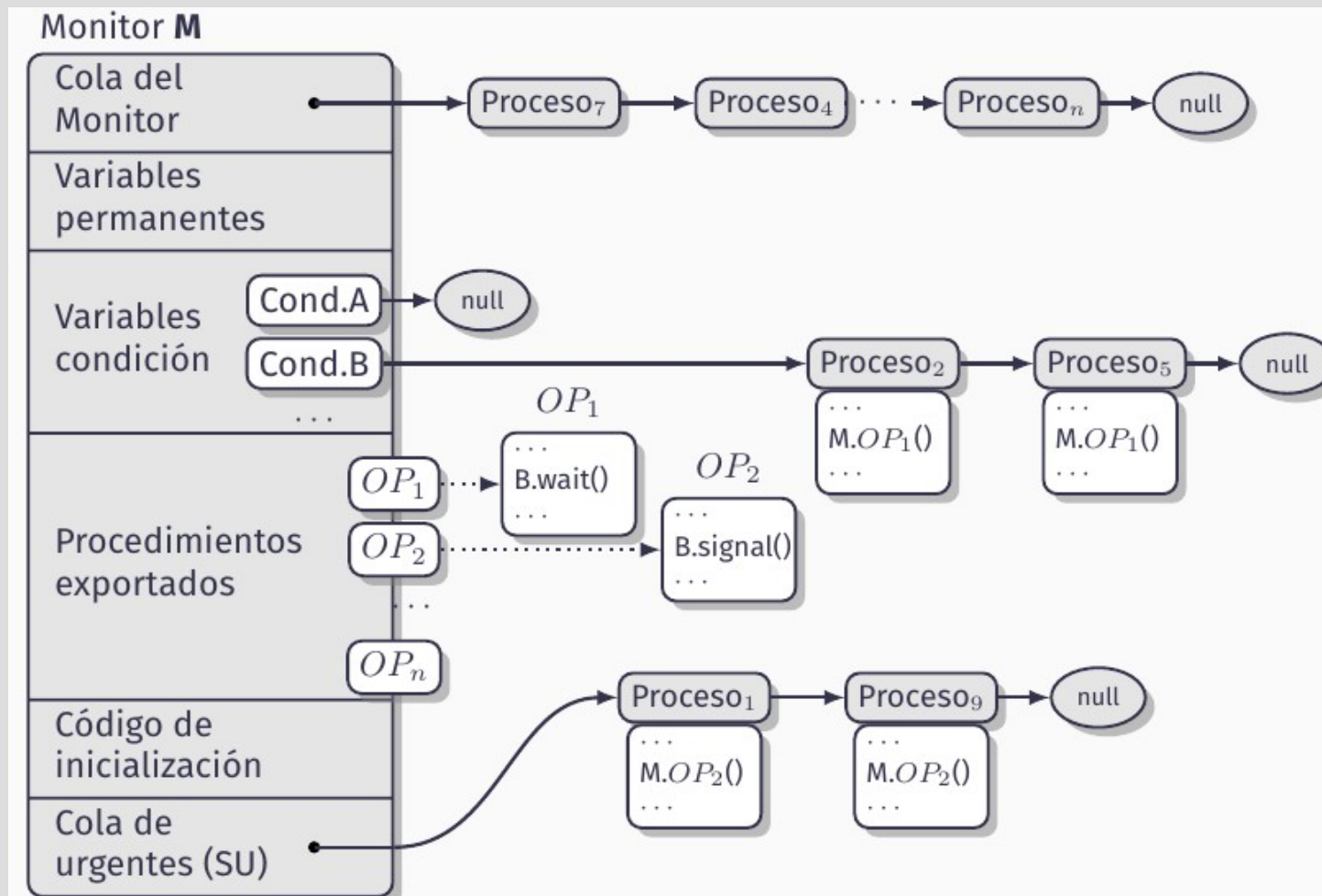
- Similar SE, pero se intenta corregir el problema de falta de equidad:
- Proceso **señalador se bloquea tras signal** y señalado reanuda inmediatamente, garantizándose condición de reactivación.
- **Señalador entra en una nueva cola** de procesos que esperan para acceder al monitor, llamada **cola de procesos urgentes**.
 - Procesos en esta tienen **preferencia frente a procesos en cola monitor**.



2.3. Monitores. Un mecanismo de alto nivel

Señalar y Espera Urgente (SU). Cola de Urgentes

- Procesos **1** y el **9** han ejecutado la **OP2**, que hace signal de cond.B.



2.3. Monitores. Un mecanismo de alto nivel

Señalar y Espera Urgente (SU). Cola de Urgentes

Potencia expresiva

- **Equivalentes:** Todas las semánticas son capaces de resolver los mismos problemas.

Facilidad de uso

- **Semántica SS condiciona el estilo de programación** y puede llevar a aumentar de forma artificial el número de procedimientos.

Eficiencia

- **SE y SU podrían resultar ineficientes cuando no hay código tras signal**, ya que señalador debe bloquearse y reactivarse, para justo después abandonar el monitor.
- **SC también es algo ineficiente** al obligar a usar un bucle para cada instrucción signal.

2.3. Monitores. Un mecanismo de alto nivel

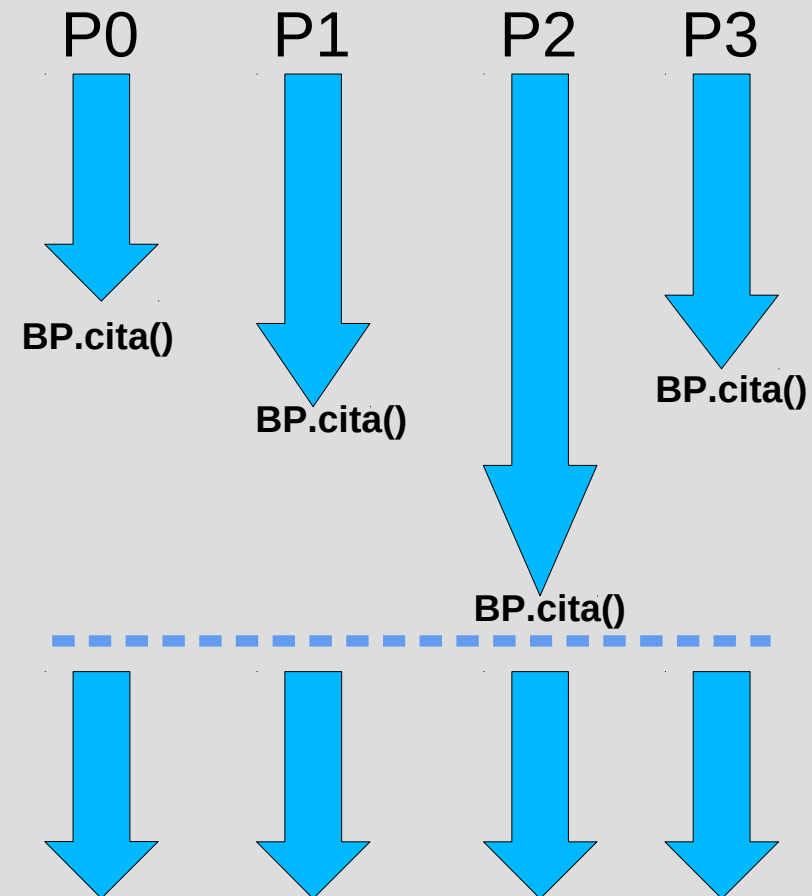
Ejemplo Comparativo semánticas. Barrera parcial

Ejemplo para mostrar diferencias entre semánticas: Monitor **barrera parcial (BP)**.

- Monitor con un **único procedimiento** público: ***cita***.
- Hay **p procesos** ejecutando bucle infinito, en cada iteración realizan una actividad de duración arbitraria y después **invocan *cita***.

Requisitos de Sincronización

- 1) Ningún proceso termina ***cita*** antes de que al menos n ($1 < n < p$) la hayan iniciado. Tras la cita cada proceso imprime un mensaje.
- 2) Un grupo de n procesos llegan a la cita e imprime su mensaje antes de que lo haga cualquier proceso del siguiente grupo.



2.3. Monitores. Un mecanismo de alto nivel

Implementación del Monitor Barrera parcial

Cada proceso debe esperar a los otros $n-1$ procesos.

- Se usa una variable condición (**cola**) para la espera y un contador entero (**contador**) para registrar el número de los que han llegado ya. **Condición de reactivación:** (contador == n)

```
Monitor BP                                { monitor Barrera Parcial }
var cola      : condition ; { procesos esperando contador==n }
    contador  : integer ;   { número de procesos ejecutando cita }

procedure cita() ;
begin
    contador := contador+1 ; { registrar un proceso más ejecutando cita }
    if contador < n then    { si todavía no hay n procesos: }
        cola.wait();       { esperar a que los haya }
    else begin              { si ya hay n procesos ejecutando la cita }
        for i := 1 to n-1 do { para cada uno de los que esperan }
            cola.signal();   { despertalo }
            contador := 0 ;   { volver a poner el contador a 0 }
        end
        print("salgo de cita"); { mensaje de salida }
    end
begin                        { inicialización: }
    contador := 0 ;          { inicialmente, no hay procesos en cita }
end
```


2.3. Monitores. Un mecanismo de alto nivel

Barrera parcial. Implementación alternativa

Los procesos **señalados** se despiertan en cadena.
contador= número de procesos en la cola.

- **No funciona con SC:** Tras wait los señalados vuelven a cola del monitor, donde ya podría haber esperando procesos del siguiente grupo que entrarían a la cita antes de tiempo.
- **Funciona con el resto (SE, SS, SU):** Procesos señalados completan la ejecución de cita inmediatamente después de salir del wait y los del siguiente grupo no tienen opción de colarse.

Seamos **cuidadosos con la semántica en uso**, especialmente si el monitor tiene código tras signal. Generalmente, **semántica SC puede complicar** mucho el diseño.

```
Monitor BP
var cola      : condition ;
    contador  : integer ;

procedure cita() ;
begin
    contador := contador+1 ;
    if contador < n then
        cola.wait();
    contador := contador-1;
    print("salgo de la cita")
    if contador > 0 then
        cola.signal();
    end
begin
    contador := 0 ;
end
```


2.3. Monitores. Un mecanismo de alto nivel

Implementación de monitores con semáforos. Enfoque

Es posible implementar cualquier monitor usando semáforos para modelar las colas del monitor.

- **Cola del monitor:** Se implementa con un **semáforo binario mutex** (0 si algún proceso está ejecutando código y 1 en otro caso).
- **Colas de condición:** Para cada variable condición será necesario definir un semáforo (siempre a 0) y una variable entera que registra número procesos esperando.
- **Cola de procesos urgentes** (en semántica SU): debe haber un contador entero (num. procesos esperando) y un semáforo (siempre a 0).

Limitaciones

- No permite llamadas recursivas a los procedimientos
- No asegura orden FIFO en colas.

“Los semáforos y monitores son equivalentes en potencia expresiva pero los monitores facilitan el desarrollo”.

2.3. Monitores. Un mecanismo de alto nivel

Implementación con semáforos. Exclusión mutua

Implementación de la EM en el acceso a los procs

```
procedure P1(...) { impl. de un proc. }  
begin                { del monitor }  
    sem_wait(mutex);  
    { cuerpo del procedimiento }  
    sem_signal(mutex);  
end
```

```
{ inicialización }  
mutex := 1 ;
```

Con semántica SU: necesitamos un semáforo **urgentes** y un contador entero **n_urgentes** (núm. bloqueados en urgentes):

```
procedure P1(...) { impl. de un proc. }  
begin                { del monitor }  
    sem_wait(mutex);  
    { cuerpo del procedimiento }  
    if n_urgentes > 0 then sem_signal(urgentes);  
                           else sem_signal(mutex);  
end
```

```
{ inicialización }  
urgentes := 0 ;  
n_urgentes := 0 ;
```

2.3. Monitores. Un mecanismo de alto nivel

Implementación con semáforos. Variables condición

Para cada variable condición **cond** definimos un semáforo asociado **cond_sem**, (siempre a 0) y un contador entero **n_cond** (inicializado a 0) para contabilizar procesos bloqueados en dicho semáforo:

Cond.wait()

```
n_cond := n_cond + 1 ;  
if n_urgentes != 0 then  
    sem_signal(urgentes) ;  
else  
    sem_signal(mutex);  
    sem_wait(cond_sem);  
n_cond := n_cond - 1 ;
```

Cond.signal()

```
if n_cond != 0 then begin  
    n_urgentes := n_urgentes + 1 ;  
    sem_signal(cond_sem);  
    sem_wait(urgentes);  
    n_urgentes := n_urgentes - 1 ;  
end
```

2.4. Soluciones software con espera ocupada

Introducción

Veremos diversas **soluciones algorítmicas para lograr EM** en una sección crítica usando:

- Variables compartidas entre los procesos involucrados.
- Espera ocupada cuando sea necesario en el protocolo de entrada.

Veremos dos algoritmos para 2 procesos: **Algoritmos de Dekker y de Peterson.**

Estructura de los procesos en estos algoritmos:

- **Protocolo de entrada (PE):** Instrucciones que incluyen **espera** cuando no se pueda acceder a sección crítica.
- **Sección crítica (SC):** Instrucciones que solo pueden ser ejecutadas por un proceso como mucho.
- **Protocolo de salida (PS):** permiten que otros procesos sepan que este proceso ha terminado SC.
- **Resto de Sentencias (RS):** Sentencias que no forman parte de las etapas anteriores.

```
PE
p0sc := true ;
turno0 := false ;
while p1sc and not turno0 do
begin end
{ sección crítica }
p0sc := false ; PS
{ resto sección }
```

2.4. Soluciones software con espera ocupada

Condiciones comportamiento procesos (1)

Para **simplificar análisis**, se hacen estas suposiciones:

- Cada proceso tiene una **única SC**, formada por un **único bloque contiguo** de instrucciones.
- Proceso es un **bucle infinito** que ejecuta dos pasos repetidamente:
 - **Sección crítica** (con PE antes y el PS después)
 - **Resto de sentencias**: se emplea un tiempo arbitrario no acotado, e incluso el proc. podría finalizar en esta sección.
- **No suposición sobre cuántas veces** un proceso intenta **acceder SC**.

```
process P0 ;  
begin  
  while true do begin                                PE  
    p0sc := true ;  
    turno0 := false ;  
    while p1sc and not turno0  
      begin end  
    { sección crítica }  
    (p0sc := false ;) PS  
    { resto sección }  
  end  
end
```

2.4. Soluciones software con espera ocupada

Condiciones comportamiento procesos (2)

Para implementar soluciones correctas al problema de EM, es necesario suponer que:

Los procesos siempre terminan una SC y la ejecutan en un intervalo de tiempo finito.

Durante ejecución de la SC, el proceso:

- NO finaliza/aborta o es finalizado/abortado externamente.
- NO entra en bucle infinito.
- NO es bloqueado o suspendido indefinidamente de forma externa.

En general, es **deseable** que el tiempo empleado en **SC** sea el **menor posible**.

```
process P0 ;  
begin  
  while true do begin PE  
    p0sc := true ;  
    turno0 := false ;  
    while p1sc and not turno0  
      begin end  
    { sección crítica }  
    p0sc := false ; PS  
    { resto sección }  
  end  
end
```

2.4. Soluciones software con espera ocupada

Propiedades requeridas algoritmo EM

Para que un **algoritmo para EM** sea correcto, se deben cumplir estas **tres propiedades mínimas**:

- 1) Exclusión mutua
- 2) Progreso
- 3) Espera limitada

Además, hay **propiedades deseables** adicionales que también deben cumplirse:

- 4) Eficiencia
- 5) Equidad

Si bien consideramos correcto un algoritmo que no sea muy eficiente o para el que no pueda demostrarse claramente la equidad.

```
process P0 ;  
begin  
  while true do begin PE  
    p0sc := true ;  
    turno0 := false ;  
    while p1sc and not turno0  
      begin end  
    { sección crítica }  
    p0sc := false ; PS  
    { resto sección }  
  end  
end
```

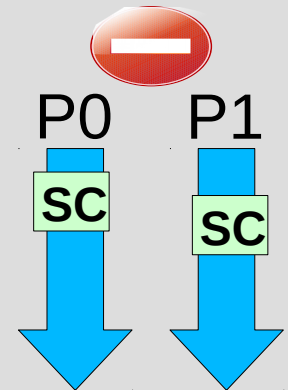
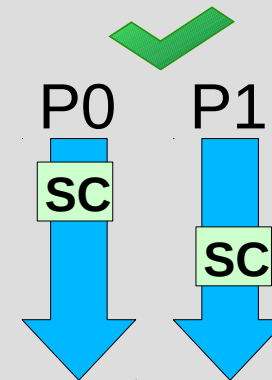
2.4. Soluciones software con espera ocupada

Propiedades mínimas requeridas.

Exclusión Mutua

En cada instante de tiempo, y para cada SC existente, habrá como mucho un proceso ejecutando alguna sentencia de dicha SC.

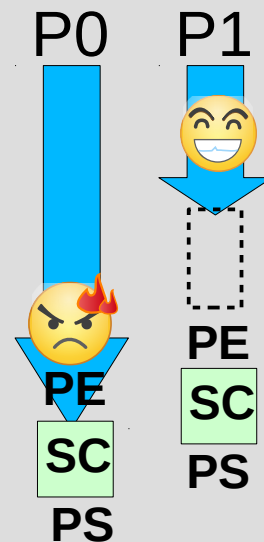
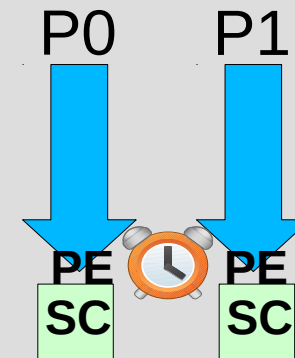
- Propiedad fundamental pero no suficiente.



Propiedad de Progreso

- 1) *Tras intervalo de tiempo finito desde que ingresó el 1er proceso al PE, uno de los procesos en el PE podrá acceder a SC.*

- Si se incumple \Rightarrow posibilidad **interbloqueo**.



- 2) *La elección del proceso que accede es completamente independiente del comportamiento de procesos que durante ese intervalo ni han estado en SC ni han intentado acceder.*

2.4. Soluciones software con espera ocupada

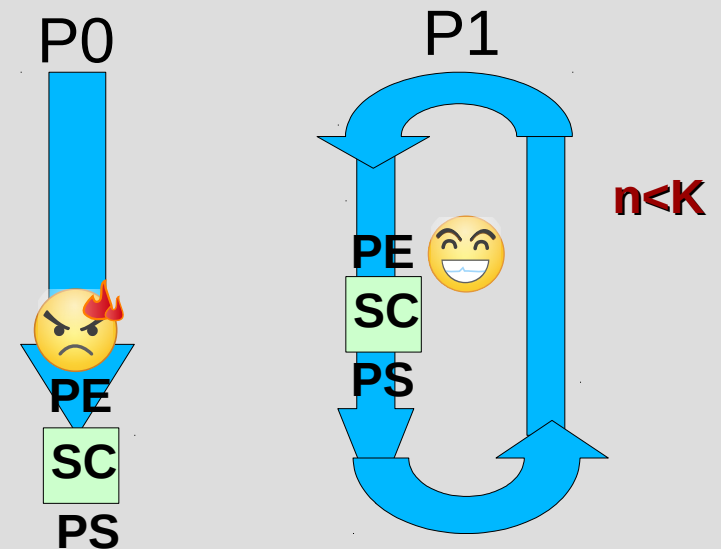
Propiedades mínimas. Espera limitada

Espera Limitada

Supongamos que un **proceso** emplea un intervalo de tiempo **en el PE intentando acceder** a SC. Durante ese intervalo, cualquier **otro proceso activo puede entrar n veces** a ese PE y acceder a SC (incluyendo $n = 0$). La propiedad establece:

Un algoritmo de EM debe diseñarse de forma que n nunca superará un valor determinado.

- Esperas en PE siempre serán finitas (si procs. emplean tiempo finito en SC).



2.4. Soluciones software con espera ocupada

Propiedades deseables. Eficiencia y equidad

Eficiencia:

Los protocolos de entrada y salida deben emplear **poco tiempo de procesamiento** (excluyendo esperas ocupadas en PE), y las variables compartidas deben usar **poca cantidad de memoria**.

Equidad:

Cuando haya varios procesos compitiendo por acceder a SC (de forma repetida en el tiempo), **no debería existir posibilidad de que sistemáticamente se perjudique a algunos y se beneficie a otros**.

2.4. Soluciones software con espera ocupada

Refinamiento sucesivo de Dijkstra. Introducción

El Refinamiento sucesivo de Dijkstra hace referencia a una **serie de algoritmos que intentan resolver el problema de EM.**

- **Comienza con una versión muy simple**, incorrecta (no cumple alguna propiedad), y se hacen sucesivas mejoras para intentar cumplir las 3 propiedades mínimas.
- **Muestra patologías comunes** de estos algoritmos e ilustra muy bien la importancia de las propiedades.
- La **versión final** correcta se denomina **Algoritmo de Dekker**.
- Por simplicidad, únicamente **algoritmos para 2 procesos** .
- Suponemos **2 procesos, P0 y P1**, cada uno ejecuta bucle infinito conteniendo: PE, SC, PS y RS.

2.4. Soluciones software con espera ocupada

Refinamiento sucesivo de Dijkstra. Versión 1

Usa variable lógica compartida **p01sc** que valdrá **true** solo si algún proceso está en SC.

```
{ variables compartidas y valores iniciales }  
var p01sc : boolean := false ; { indica si la SC esta ocupada }
```

```
process P0 ;  
begin  
  while true do begin  
    while p01sc do begin end  
    p01sc := true ;  
    { sección crítica }  
    p01sc := false ;  
    { resto sección }  
  end  
end
```

```
process P1 ;  
begin  
  while true do begin  
    while p01sc do begin end  
    p01sc := true ;  
    { sección crítica }  
    p01sc := false ;  
    { resto sección }  
  end  
end
```

2.4. Soluciones software con espera ocupada

Refinamiento sucesivo de Dijkstra. Versión 2

Se usará una **única variable lógica (turno0)**, cuyo valor indicará cuál debe entrar a SC. Valdrá true si es P0, y false si es P1.

```
{ variables compartidas y valores iniciales }  
var turno0 : boolean := true ; { podría ser también false }
```

```
process P0 ;  
begin  
  while true do begin  
    while not turno0 do begin end  
    { sección crítica }  
    turno0 := false ;  
    { resto sección }  
  end  
end
```

```
process P1 ;  
begin  
  while true do begin  
    while turno0 do begin end  
    { sección crítica }  
    turno0 := true ;  
    { resto sección }  
  end  
end
```

2.4. Soluciones software con espera ocupada

Refinamiento sucesivo de Dijkstra. Versión 3

Para impedir la alternancia, se usan dos variables lógicas (**p0sc**, **p1sc**) en lugar de solo una. Cada variable vale true si el correspondiente proceso está en SC.

```
{ variables compartidas y valores iniciales }  
var p0sc   : boolean := false ; { verdadero solo si proc. 0 en SC }  
    p1sc   : boolean := false ; { verdadero solo si proc. 1 en SC }
```

```
process P0 ;  
begin  
  while true do begin  
    while p1sc do begin end  
    p0sc := true ;  
    { sección crítica }  
    p0sc := false ;  
    { resto sección }  
  end  
end
```

```
process P1 ;  
begin  
  while true do begin  
    while p0sc do begin end  
    p1sc := true ;  
    { sección crítica }  
    p1sc := false ;  
    { resto sección }  
  end  
end
```

2.4. Soluciones software con espera ocupada

Refinamiento sucesivo de Dijkstra. Versión 4

Para solucionar el problema anterior se puede **cambiar el orden de las dos sentencias del PE**. Variables lógicas p0sc y p1sc están a true cuando el correspondiente proceso está en SC o intenta entrar a SC desde PE.

```
{ variables compartidas y valores iniciales }  
var p0sc : boolean := falso ; { verdadero solo si proc. 0 en PE o SC }  
    p1sc : boolean := falso ; { verdadero solo si proc. 1 en PE o SC }
```

```
process P0 ;  
begin  
    while true do begin  
        p0sc := true ;  
        while p1sc do begin end  
        { sección crítica }  
        p0sc := false ;  
        { resto sección }  
    end  
end
```

```
process P1 ;  
begin  
    while true do begin  
        p1sc := true ;  
        while p0sc do begin end  
        { sección crítica }  
        p1sc := false ;  
        { resto sección }  
    end  
end
```

2.4. Soluciones software con espera ocupada

Refinamiento sucesivo de Dijkstra. Versión 5

Para solucionarlo, si un proceso ve que el otro quiere entrar, el primero pone su variable temporalmente a false.

```
var p0sc : boolean := false ; { true solo si proc. 0 en PE o SC }  
    p1sc : boolean := false ; { true solo si proc. 1 en PE o SC }
```

```
1 process P0 ;  
2 begin  
3   while true do begin  
4     p0sc := true ;  
5     while p1sc do begin  
6       p0sc := false ;  
7       { espera durante un tiempo }  
8       p0sc := true ;  
9     end  
10    { sección crítica }  
11    p0sc := false ;  
12    { resto sección }  
13  end  
14 end
```

```
1 process P1 ;  
2 begin  
3   while true do begin  
4     p1sc := true ;  
5     while p0sc do begin  
6       p1sc := false ;  
7       { espera durante un tiempo }  
8       p1sc := true ;  
9     end  
10    { sección crítica }  
11    p1sc := false ;  
12    { resto sección }  
13  end  
14 end
```


2.4. Soluciones software con espera ocupada

Algoritmo de Dekker. Introducción

El algoritmo de Dekker, es un **algoritmo correcto** (cumple propiedades mínimas).

- Se puede considerar el **resultado final del refinamiento** de Dijkstra:
- **Incorpora espera de cortesía** en cada proceso (como versión 5), durante la cual cede al otro la posibilidad de acceder SC, cuando ambos coinciden en PE.
- **Evita interbloqueos** mediante **variable turno**: la espera de cortesía solo la realiza uno de los dos procesos de forma alterna.
 - Variable **turno** permite también **detectar final de espera de cortesía**, implementada mediante espera ocupada.

```
{ variables compartidas y valores iniciales }  
var p0sc      : boolean := falso ; { true solo si proc.0 en PE o SC }  
    p1sc      : boolean := falso ; { true solo si proc.1 en PE o SC }  
    turno0    : boolean := true  ; { true ==> pr.0 no hace espera de cortesía }
```

2.4. Soluciones software con espera ocupada

Algoritmo de Dekker. Pseudocódigo

```
{ variables compartidas y valores iniciales }  
var p0sc      : boolean := falso ; { true solo si proc.0 en PE o SC }  
    p1sc      : boolean := falso ; { true solo si proc.1 en PE o SC }  
    turno0    : boolean := true  ; { true ==> pr.0 no hace espera de cortesía
```

```
1 process P0 ;  
2 begin  
3     while true do begin  
4         p0sc := true ;  
5         while p1sc do begin  
6             if not turno0 then begin  
7                 p0sc := false ;  
8                 while not turno0 do  
9                     begin end  
10                p0sc := true ;  
11            end  
12        end  
13        { sección crítica }  
14        turno0 := false ;  
15        p0sc := false ;  
16        { resto sección }  
17    end  
18 end
```

```
1 process P1 ;  
2 begin  
3     while true do begin  
4         p1sc := true ;  
5         while p0sc do begin  
6             if turno0 then begin  
7                 p1sc := false ;  
8                 while turno0 do  
9                     begin end  
10                p1sc := true ;  
11            end  
12        end  
13        { sección crítica }  
14        turno0 := true ;  
15        p1sc := false ;  
16        { resto sección }  
17    end  
18 end
```

2.4. Soluciones software con espera ocupada

Algoritmo de Peterson. Introducción

El algoritmo de Peterson, es otro **algoritmo correcto pero más simple y más eficiente** que el algoritmo de Dekker.

- También usa **dos variables lógicas** que expresan la presencia de cada proceso en PE o SC, **más** una variable de **turno** que permite romper el interbloqueo en caso de acceso simultáneo al PE.
- **Asignación de turno se hace al inicio del PE** en lugar de en PS, con lo cual, en caso de acceso simultáneo al PE, el último proceso en ejecutar la asignación (atómica) al turno da preferencia al anterior.
- A diferencia de Dekker, el PE **no usa dos bucles anidados**, sino que unifica ambos en uno solo.

```
{ variables compartidas y valores iniciales }  
var p0sc      : boolean := falso ; { true solo si proc.0 en PE o SC }  
    p1sc      : boolean := falso ; { true solo si proc.1 en PE o SC }  
    turno0    : boolean := true  ; { true ==> pr.0 no hace espera de cortesía }
```

2.4. Soluciones software con espera ocupada

Algoritmo de Peterson. Introducción

```
{ variables compartidas y valores iniciales }  
var p0sc      : boolean := falso ; { true solo si proc.0 en PE o SC }  
    p1sc      : boolean := falso ; { true solo si proc.1 en PE o SC }  
    turno0    : boolean := true  ; { true ==> pr.0 no hace espera de cortesía }
```

```
1 process P0 ;  
2 begin  
3     while true do begin  
4         p0sc := true ;  
5         turno0 := false ;  
6         while p1sc and not turno0 do  
7             begin end  
8         { sección crítica }  
9         p0sc := false ;  
10        { resto sección }  
11    end  
12 end
```

```
1 process P1 ;  
2 begin  
3     while true do begin  
4         p1sc := true ;  
5         turno0 := true ;  
6         while p0sc and turno0 do  
7             begin end  
8         { sección crítica }  
9         p1sc := false ;  
10        { resto sección }  
11    end  
12 end
```

2.4. Soluciones software con espera ocupada

Algoritmo de Peterson. Demostración EM

Hipótesis de partida: Existe instante t en que ambos procesos están en SC.

- Supongamos última asignación a `turno0` se hizo en un instante anterior $s < t$.
- En intervalo $(s, t]$, ni `p0sc` ($= \text{true}$), ni `p1sc` ($= \text{true}$), ni `turno0` cambian su valor.
- Si cualquiera (**P0 ó P1**) fue el que asignó `turno0` en instante $s \Rightarrow$ No entraría a SC durante $(s, t]$ ya que condición del while sería true durante $(s, t]$.
- **Contradicción hipótesis** \Rightarrow Nunca ambos estarán ejecutando su SC

```
process P0 ;
begin
  while true do begin
    p0sc := true ;
    turno0 := false ;
    while p1sc and not turno0 do
      begin end
    { sección crítica }
    p0sc := false ;
    { resto sección }
  end
end
```

```
process P1 ;
begin
  while true do begin
    p1sc := true ;
    turno0 := true ;
    while p0sc and turno0 do
      begin end
    { sección crítica }
    p1sc := false ;
    { resto sección }
  end
end
```

2.4. Soluciones software con espera ocupada

Algoritmo de Peterson. Espera Limitada

Supongamos que hay un proceso (p.ej. P0) en espera ocupada en el PE, en un instante t , y veamos **cuántas veces m puede entrar P1 a SC antes de que lo logre P0**.

- P0 puede pasar a la SC antes que P1, en ese caso $m = 0$.
- P1 puede pasar a la SC antes que P0 (que continúa en bucle). En ese caso $m = 1$.

En ambos casos, **P1 no puede volver a SC mientras P0 continúa en el bucle**, ya que pasaría por asignación $\text{turno0} = \text{true}$ provocando que, tras un tiempo finito, P0 entrara a SC mientras P1 continúa en bucle. La cota que requiere la propiedad es $n = 1$.

```
process P0 ;
begin
  while true do begin
    p0sc := true ;
    turno0 := false ;
    while p1sc and not turno0 do
      begin end
    { sección crítica }
    p0sc := false ;
    { resto sección }
  end
end
```

```
process P1 ;
begin
  while true do begin
    p1sc := true ;
    turno0 := true ;
    while p0sc and turno0 do
      begin end
    { sección crítica }
    p1sc := false ;
    { resto sección }
  end
end
```


2.4. Soluciones software con espera ocupada

Algoritmo de Peterson. Progreso en la ejecución

Debemos demostrar **dos propiedades**:

- **Ausencia de interbloqueos en PE**: Si suponemos que hay interbloqueo de ambos, eso significa que son indefinida y simultáneamente verdaderas ambas condiciones de sus bucles $\Rightarrow \text{turno0} = \text{true} \text{ and } (\text{not } \text{turno0}) = \text{true} \Rightarrow \text{ABSURDO}$.
- **Independencia de procesos en RS**: Si un proceso (p.ej. P0) está en PE y el otro (P1) está en RS, entonces $\text{p1sc} = \text{false}$ y P0 puede progresar a la SC independientemente del comportamiento de P1 (que podría terminar o bloquearse estando en RS, sin impedir por ello el progreso de P0). El mismo razonamiento puede hacerse al revés.

```
process P0 ;
begin
  while true do begin
    p0sc := true ;
    turno0 := false ;
    while p1sc and not turno0 do
      begin end
    { sección crítica }
    p0sc := false ;
    { resto sección }
  end
end
```

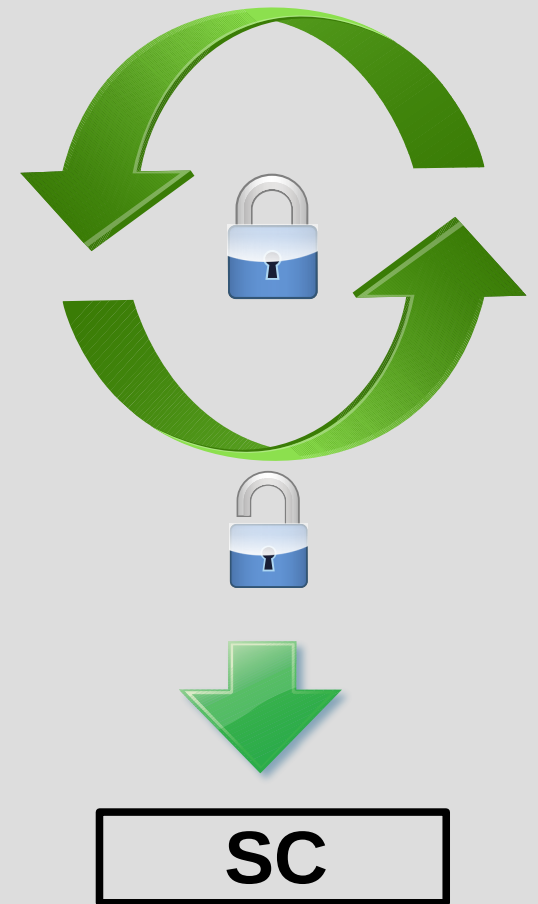
```
process P1 ;
begin
  while true do begin
    p1sc := true ;
    turno0 := true ;
    while p0sc and turno0 do
      begin end
    { sección crítica }
    p1sc := false ;
    { resto sección }
  end
end
```


2.5. Soluciones hardware para EM

Introducción a los cerrojos

- **Cerrojo**: Solución hardware **basada en espera ocupada** que puede usarse en procesos concurrentes con mem. compartida para solucionar problema EM.
 - **Espera ocupada**: Bucle que se ejecuta hasta que ningún otro proceso esté dentro de su SC.
- **Valor lógico en posición de memoria compartida (cerrojo)** indica si algún proceso está en SC.
- **En PS** se actualiza cerrojo para reflejar **SC libre**.

Veremos solución elemental (incorrecta) que ilustra necesidad de instrucciones hardware específicas (o soluciones más elaboradas).



2.5. Soluciones hardware para EM

Posible solución elemental

Incorrecta. **No garantiza EM**. Existen secuencias de interfoliación que permiten a varios procesos acceder SC a la vez:

Situación (ya vista por nosotros)

- N procesos acceden a PE y todos leen el valor de `sc_ocupada` a false (ninguno lo escribe antes de que otro lo lea).
- Todos registran que SC está libre, y todos acceden.

Solución: Usar instrucciones máquina atómicas para acceso a la zona de memoria donde se aloja el cerrojo.

- Veremos una de ellas: **TestAndSet**.

```
var sc_ocupada : boolean := false ;

{ procesos }
process P[ i : 1 .. n ];
begin
    while true do begin
        while sc_ocupada do begin end
        sc_ocupada := true ;
        { seccion critica }
        sc_ocupada := false ;
        { resto seccion }
    end
end
```

2.5. Soluciones hardware para EM

Instrucción TestAndSet y Solución

TestAndSet: Instrucción máquina disponible en el repertorio de algunos procesadores.

- **Argumento: Dirección** de memoria de la variable lógica que actúa como **cerrojo**.
- Se puede invocar como una función desde LLPP de alto nivel.

Ejecuta atómicamente (no afectada por el efecto de instrucciones de otros procs.):

- 1) Lee valor anterior del cerrojo.
- 2) Pone cerrojo a true.
- 3) Devuelve valor anterior del cerrojo.

Solución basada en TestAndSet

```
var sc_ocupada : boolean := false ; { true solo si
                                     SC esta ocupada }

{ procesos }
process P[ i : 1 .. n ];
begin
    while true do begin
        while TestAndSet( sc_ocupada ) do begin end
        { seccion critica }
        sc_ocupada := false ;
        { resto seccion }
    end
end
```

2.5. Soluciones hardware para EM

Desventajas y uso de los cerrojos

Cerrojos constituyen una **solucion válida para EM**: consume poca memoria, es eficiente en tiempo (excluyendo esperas ocupadas) y es válida para cualquier num. de procesos.

No obstante presenta **Desventajas**:

- **Esperas ocupadas consumen tiempo de CPU** que podría dedicarse a otros procesos para hacer trabajo útil
- **Se puede acceder directamente a los cerrojos** y por tanto un programa erróneo o escrito malintencionadamente puede poner un cerrojo en un estado incorrecto, pudiendo dejar a otros procesos indefinidamente en espera ocupada.
- **No se cumplen ciertas condiciones de equidad.**

Desventajas hacen que su **uso sea restringido**:

- **Por seguridad**, solo se usan desde componentes software que forman parte del sistema operativo, librerías de hebras, de tiempo real o similares (suelen estar bien comprobados y libres de errores o código malicioso).
- **Por eficiencia** debido a la espera ocupada: solo en casos en los que la ejecución de la **SC conlleva un intervalo de tiempo muy corto.**