UNIVERSIDAD
DE GRANADA

Sistemas Concurrentes y Distribuidos: Tema 2. Sincronización en memoria compartida.

Carlos Ureña / Jose M. Mantas / Pedro Villar

2018-19

Grado en Ingeniería Informática / Grado en Ingeniería Informática y Matemáticas.
Dpt. Lenguajes y Sistemas Informáticos
ETSI Informática y de Telecomunicación
Universidad de Granada

Tema 2. Sincronización en memoria compartida.
Índice.

Sección 1.
Introducción a la sincronización en memoria compartida..

Sincronización en memoria compartida

En esta tema estudiaremos soluciones para exclusión mutua y sincronización basadas en el uso de memoria compartida entre los procesos involucrados. Este tipo de soluciones se pueden dividir en dos categorías:

- **Soluciones de bajo nivel con espera ocupada**
  están basadas en programas que contienen explícitamente instrucciones de bajo nivel para lectura y escritura directamente a la memoria compartida, y bucles para realizar las esperas.
- **Soluciones de alto nivel**
  partiendo de las anteriores, se diseña una capa software por encima que ofrece un interfaz para las aplicaciones. La sincronización se consigue bloqueando un proceso cuando deba esperar.

Soluciones de bajo nivel con espera ocupada

Cuando un proceso debe esperar a que ocurra un evento o sea cierta determinada condición, entra en un bucle indefinido en el cual continuamente comprueba si la situación ya se da o no (a esto se le llama **espera ocupada**). Este tipo de soluciones se pueden dividir en dos categorías:

- **Soluciones software**:
  se usan operaciones estándar sencillas de lectura y escritura de datos simples (típicamente valores lógicos o enteros) en la memoria compartida
- **Soluciones hardware** (**cerrojos**):
  basadas en la existencia de instrucciones máquina específicas dentro del repertorio de instrucciones de los procesadores involucrados

Soluciones de alto nivel

Las soluciones de bajo nivel con espera ocupada se prestan a errores, producen algoritmos complicados y tienen un impacto negativo en la eficiencia de uso de la CPU (por los bucles). En las soluciones de alto nivel se ofrecen interfaces de acceso a estructuras de datos y además se usa bloqueo de procesos en lugar de espera ocupada. Veremos algunas de estas soluciones:

- **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**: son soluciones de más alto nivel que los semáforos, y que se pueden implementar sobre ellos.
- **Monitores:** son soluciones de más alto nivel que las anteriores y se pueden implementar en algunos lenguajes orientados a objetos como Java o Python.

Sección 2.
Semáforos para sincronización.

Subsección 2.1. Introducción.

Semáforos. Introducción

Los **semáforos** constituyen un mecanismo que soluciona o aminora los problemas de las soluciones de bajo nivel, y tienen un ámbito de uso más amplio:

- no se usa espera ocupada, sino bloqueo de procesos (uso mucho más eficiente de la CPU)
- resuelven fácilmente el problema de exclusión mutua con esquemas de uso sencillos
- se pueden usar para resolver problemas de sincronización (aunque en ocasiones los esquemas de uso son complejos)
- el mecanismo se implementa mediante instancias de una estructura de datos a las que se accede únicamente mediante subprogramas específicos. Esto aumenta la seguridad y simplicidad.

Bloqueo y desbloqueo de procesos

Los semáforos exigen que los procesos en espera no ocupen la CPU, esto implica que:

- un proceso en ejecución debe poder solicitar quedarse bloqueado
- un proceso bloqueado no puede ejecutar instrucciones en la CPU
- un proceso en ejecución debe poder solicitar que se desbloquee (se reanude) algún otro proceso bloqueado.
- deben poder existir simultáneamente varios conjuntos de procesos bloqueados.
- cada petición de bloqueo o desbloqueo se debe referir a alguno de estos conjuntos.

Todo esto requiere el uso de servicios externos (proporcionados por el sistema operativo o por la librería de hebras).

Subsección 2.2.
Estructura de un semáforo.

Estructura de un semáforo

Un semáforo es un instancia de una estructura de datos (un registro) que contiene los siguientes elementos:

- Un conjunto de procesos bloqueados (de estos procesos decimos que están esperando al semáforo).
- Un valor natural (un valor entero no negativo), al que llamaremos por simplicidad *valor del semáforo*

Estas estructuras de datos residen en memoria compartida. Al inicio de un programa que los usa debe poder inicializarse cada semáforo:

- el conjunto de procesos asociados estará vacío
- se deberá indicar un valor inicial del semáforo

Subsección 2.3.
Operaciones sobre los semáforos..

Operaciones sobre los semáforos

Además de la inicialización, solo hay dos operaciones básicas que se pueden realizar sobre una variable u objeto cualquiera de tipo semáforo (que llamamos s) :

- **`sem_wait`**(s)
  1. Si el valor de s es 0, bloquear el proceso, que será reanudado después en un instante en que el valor ya es 1.
  2. Decrementar el valor del semáforo en una unidad.
- **`sem_signal`**(s)
  1. Incrementar el valor de s en una unidad.
  2. Si hay procesos esperando en s, reanudar o despertar a uno de ellos (ese proceso pone el semáforo a 0 al salir).

Este diseño implica que **el valor del semáforo nunca es negativo**, ya que antes de decrementar se espera a que sea 1. Además, **solo puede haber procesos esperando cuando el valor es** 0.

Invariante de un semáforo

Dado un semáforo s, en un instante de tiempo cualquiera t su valor (v_t) será el valor inicial (v_0) más el número de llamadas a **`sem_signal`** completadas (n_s), menos el número de llamadas a **`sem_wait`** completadas (n_w). Ese valor nunca es negativo. Es decir:

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

- Los cuatro valores son enteros no negativos.
- La igualdad se demuestra viendo que **`sem_signal`** siempre incrementa el valor y **`sem_wait`** siempre lo decrementa (pero espera antes cuando es 0).
- La igualdad se cumple cuando no se está ejecutando **`sem_wait`** o **`sem_signal`** (cuando no se está actualizando el valor)
- No cuentan las llamadas a **`sem_wait`** no completadas en el instante t por haber dejado el proceso en espera.

Implementación de sem_wait y sem_signal.

Se pueden implementar con este esquema:

```
procedure sem_wait( s :semaphore);
begin
   if s.valor == 0 then
      bloquearme( s.procesos ) ;
   s.valor := s.valor - 1 ;
end
```

```
procedure sem_signal( s :semaphore);
begin
   s.valor := s.valor + 1 ;
   if not vacio( s.procesos ) then
      desbloquear_uno( s.procesos ) ;
end
```

En un semáforo cualquiera, estas operaciones se ejecutan en exclusión mutua sobre cada semáforo, es decir, no puede haber dos procesos distintos ejecutando estas operaciones a la vez sobre un mismo semáforo (excluyendo el período de bloqueo que potencialmente conlleva la llamada a **sem_wait**).

Subsección 2.4.
Uso de semáforos: patrones sencillos.

Patrones de uso sencillos

En esta sección, veremos varios patrones sencillos de uso de semáforos. Cada patrón es un esquema que permite solucionar, usando semáforos, la sincronización de un problema típico sencillo. En concreto, veremos estos tres problemas:

- **Espera única**: un proceso, antes de ejecutar una sentencia, debe esperar a que otro proceso complete otra sentencia (ocurre típicamente cuando un proceso debe leer una variable escrita por otro proceso).
- **Exclusión mutua**: acceso en exclusión mutua a una sección crítica por parte de un número arbitrario de procesos.
- **Problema del Productor/Consumidor**: similar a la espera única, pero de forma repetida en un bucle (un proceso escribe sucesivos valores en una variable, y cada uno de ellos debe ser leído una única vez por otro proceso).

Espera única: problema

El caso más sencillo de sincronización ocurre cuando un proceso (**Consumidor**) lee una variable compartida escrita por otro proceso (**Productor**):

```
{ 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
```

- Las sentencias E y L son atómicas.
- Únicamente son correctas las interfolicaciones en las que E ocurre antes que L.

Espera única: solución con un semáforo

Para introducir una espera previa a L, usamos un semáforo (**puede_leer**), cuyo valor es 1 solo cuando **x** tiene un valor ya escrito pero aun está pendiente de leer:

```
{ 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
```

Espera única: verificación

En cualquier instante durante la ejecución (en el que no se está actualizando el valor del semáforo)

- El número de veces que se ha ejecutado E es $\#E$
- El número de veces que se ha ejecutado L es $\#L$
- Solo son correctas interfolicaciones donde $\#L \leq \#E$
- Puesto que **`sem_signal`** va después de E, se cumple $n_s \leq \#E$
- Puesto que **`sem_wait`** va antes de L, se cumple $\#L \leq n_w$.
- El valor de **`puede_leer`** es $v_t = n_s - n_w$, como no puede ser negativo nunca se deduce $n_w \leq n_s$.

Encadenando las tres últimas desigualdades, obtenemos

$$\#L \leq \#E$$

como queríamos demostrar.

Uso de semáforos para exclusión mutua

Los semáforos se pueden usar para EM usando un semáforo inicializado a 1, y haciendo **wait** antes de la sección crítica y **signal** después de la sección crítica:

```
{ 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
```

Verificación de la exclusión mutua

En el programa anterior, se cumplen estas propiedades:

- Por la estructura del programa, el número de procesos ejecutando la sección crítica (n_{sc}) en un momento dado es:

$$0 \quad \leq \quad n_{sc} \quad = \quad n_w - n_s$$

- Puesto que el valor inicial v_0 es 1, por el invariante y las propiedades de los semáforos, el valor v_t de `sc_libre` es un entero no negativo, que cumple:

$$0 \quad \leq \quad v_t \quad = \quad 1 + n_s - n_w \quad = \quad 1 - n_{sc}$$

- Puesto que $0 \leq 1 - n_{sc}$, sabemos que n_{sc} cumple:

$$0 \quad \leq \quad n_{sc} \quad \leq \quad 1$$

  lo que **demuestra que se cumple la exclusión mutua.**, ya que solo puede haber 0 o 1 procesos en S.C.

Sincronización tipo Productor/Consumidor

El problema del Productor-Consumidor (que vimos) es igual al problema simple anterior, pero con las lecturas y escrituras (E y L) repetidas en un bucle infinito:

```
{ 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
```

Ahora, **únicamente son correctas las interfolicaciones en las que E y L se alternan, comenzando en E** (es decir, interfoliaciones de la forma $E, L, E, L, \ldots$).

Solución del Productor/Consumidor con semáforos

La solución parecida a la simple, pero con un nuevo semáforo (**puede_escribir**), inicializado a 1:

```
{ 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
```

Verificación de la solución

En este caso, solo son correctas las interfoliaciones que hacen siempre ciertas estas dos desigualdades:

$$0 \quad \leq \quad \#E - \#L \quad \leq \quad 1$$

ya que el número de valores escritos en **x** y pendientes de leer, es $\#E - \#L$, y ese valor solamente puede ser 0 o 1. En la solución:

- El semáforo **`puede_escribir`** asegura la condición

$$\#E - \#L \quad \leq \quad 1$$

- El semáforo **`puede_leer`** asegura la condición:

$$0 \quad \leq \quad \#E - \#L$$

Veremos como se pueden demostrar estas dos desigualdades a partir de la estructura del programa y del invariante de cada semáforo.

Verificación (semáforo puede_leer)

Respecto del semáforo **puede_leer**, el análisis es similar al que ya hemos hecho para la versión sin bucles. Se cumplen estas propiedades:

- **sem_signal** se ejecuta una vez después de $\#E$, luego $n_s \leq \#E$
- L se ejecuta una vez después de **sem_wait**, luego $\#L \leq n_w$
- El valor del semáforo es $n_s - n_w$, siempre positivo, luego $n_w \leq n_s$

Al igual que antes, encadenando las tres desigualdades, obtenemos:

$$\#L \quad \leq \quad n_w \quad \leq \quad n_s \quad \leq \quad \#E$$

lo cual demuestra, como queríamos, que:

$$0 \quad \leq \quad \#E - \#L$$

Verificación (semáforo puede_escribir)

Respecto del semáforo **puede_escribir**, se cumplen estas propiedades:

- **sem_signal** se ejecuta una vez después de $\#L$, luego $n_s \leq \#L$
- E se ejecuta una vez después de **sem_wait**, luego $\#E \leq n_w$
- El valor del semáforo es $1 + n_s - n_w$, siempre positivo, luego $n_w \leq 1 + n_s$

De nuevo encadenamos las tres desigualdades, obtenemos:

$$\#E \quad \leq \quad n_w \quad \leq \quad 1 + n_s \quad \leq \quad 1 + \#L$$

lo cual demuestra, como queríamos, que:

$$\#E - \#L \quad \leq \quad 1$$

Limitaciones de los semáforos

Los semáforos resuelven de una forma eficiente y sencilla el problema de la exclusión mutua y problemas sencillos de sincronización, sin embargo:

- los problemas más complejos de sincronización se resuelven de forma algo más compleja (es difícil verificar su validez, y es fácil que sean incorrectos).
- al igual que los cerrojos, programas erróneos o malintencionados pueden provocar que haya procesos bloqueados indefinidamente o en estados incorrectos.

En la siguiente sección se verá una solución de más alto nivel sin estas limitaciones (monitores).

Sección 3.
Monitores como mecanismo de alto nivel.

Subsección 3.1.
Introducción. Definición de monitor.

Limitaciones de los semáforos

Como se ha descrito, hay algunos **inconvenientes de usar mecanismos como los semáforos**:

- Basados en variables globales: esto impide un diseño modular y reduce la escalabilidad (incorporar más procesos al programa suele requerir la revisión del uso de las variables globales).
- El uso y función de las variables no se hace explícito en el programa, lo cual dificulta razonar sobre la corrección de los programas.
- Las operaciones se encuentran dispersas y no protegidas (posibilidad de errores).

Por tanto, es necesario un mecanismo que permita el acceso estructurado y la encapsulación, y que además proporcione herramientas para garantizar la exclusión mutua e implementar condiciones de sincronización.

Estructura y funcionalidad de un monitor

C.A.R. Hoare, en 1974, idea el concepto de **Monitor**: es un mecanismo de **alto nivel** que permite definir **objetos abstractos compartidos**, que incluyen:

- Una **colección de variables encapsuladas** (datos) que representan un recurso compartido por varios procesos.
- Un **conjunto de procedimientos** para manipular el recurso: afectan a las variables encapsuladas.

Ambos conjuntos de elementos permiten al programador invocar a los procedimientos de forma que en ellos

- Se garantiza el **acceso en exclusión mutua** a las variables encapsuladas.
- Se implementan la **sincronización requerida** por el problema mediante esperas bloqueadas.

Propiedades del monitor

Un monitor es un **recurso compartido** que se usa como un objeto al que se accede concurrentemente.

Acceso estructurado y encapsulación:

- El usuario (proceso) solo puede acceder al recurso mediante un conjunto de operaciones.
- El usuario ignora la/s variable/s que representan al recurso y la implementación de las operaciones asociadas.

Exclusión mutua en el acceso a los procedimientos

- La exclusión mutua en el acceso a los procedimientos del monitor está garantizada por definición.
- La implementación del monitor garantiza que nunca dos procesos estarán ejecutando simultáneamente algún procedimiento del monitor (el mismo o distintos).

Ventajas sobre los semáforos

La propiedades descritas de los monitores los hacen preferibles respecto de los semáforos, dado que el uso de los monitores **facilita el diseño e implementación de programas libres de errores**

- **Las variables están protegidas**: solo pueden leerse o modificarse desde el código del monitor, no desde cualquier punto de un programa que puede tener miles de líneas de código.
- **La exclusión mutua está garantizada**: el programador no tiene que usar mecanismos explícitos de exclusión mutua en el acceso a las variables compartidas.
- **Las operaciones de esperas bloqueadas y de señalización se programan exclusivamente dentro del monitor**: es más fácil verificar que el diseño es correcto.

Sintaxis de un monitor en pseudo-código

Una instancia del monitor se declara especificando las **variables permanentes**, los **procedimientos del monitor** y opcionalmente el **código de inicialización**.

```
monitor nombre_monitor ; { identificador con el que se referencia }
  var                    { decl. variables permanentes (privadas) }
   ...... ;              { (puede incluir valores iniciales)      }
  export                 { nombres de procedimientos públicos     }
    nom_exp_1,           { (si no aparece, todos son públicos)    }
    nom_exp_2, .... ;
  { declaraciones e implementación de procedimientos  }
  procedure nom_exp_1( ........ ); { nombre y parámetros formales }
    var .....   ;        { variables locales del procedimiento    }
  begin
      ...                { código que implementa el procedimiento }
  end
  .......                { resto de procedimientos del monitor    }
begin                    { código inicialización de vars. perm.   }
  ....                   { (opcional)                             }
end                      { fin de la declaración del monitor      }
```

Componentes de un monitor

Variables permanentes: son el estado interno del monitor.

- Sólo pueden ser accedidas dentro del monitor (en el cuerpo de los procedimientos y código de inicialización).
- Permanecen sin modificaciones entre dos llamadas consecutivas a procedimientos del monitor.

Procedimientos: modifican el estado interno (en E.M.)

- Pueden tener variables y parámetros locales, que toman un nuevo valor en cada activación del procedimiento.
- Algunos (o todos) constituyen la interfaz externa del monitor y podrán ser llamados por los procesos que comparten el recurso.

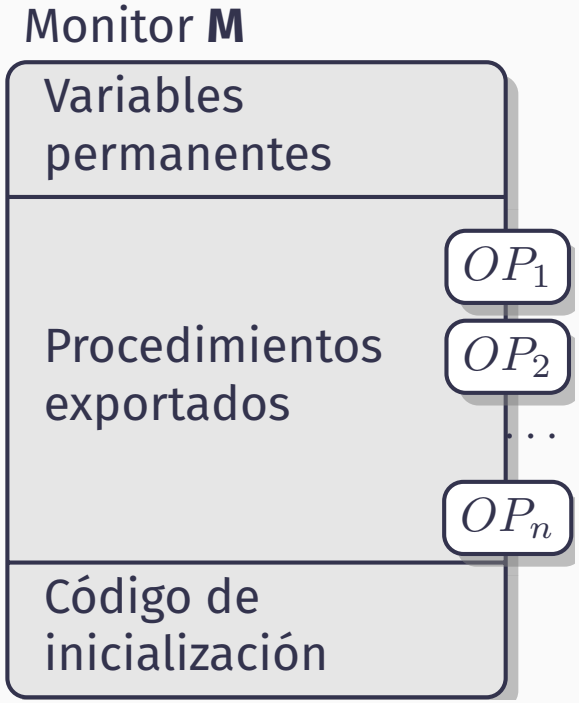
Código de inicialización: fija estado interno inicial (opcional)

- Se ejecuta una única vez, antes de cualquier llamada a procedimientos del monitor.

Diagrama de las componentes del monitor

El monitor se puede visualizar como aparece en el diagrama:

- El uso que se hace del monitor se hace **exclusivamente** usando los procedimientos exportados (constituyen el interfaz con el exterior).
- Las variables permanentes y los procedimientos no exportados **no son accesibles** desde fuera.
- **Ventaja:** la implementación de las operaciones se puede cambiar sin modificar su semántica.



Ejemplo de monitor (pseudocódigo)

Tenemos varios procesos que pueden incrementar (en una unidad) una variable compartida y poder examinar su valor en cualquier momento:

```
{ 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}
```

```
{ 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() ;
```

La variable **result** contiene el resultado en las funciones que devuelven un valor, que se declaran con **function** en lugar de **procedure**.

Subsección 3.2.
Funcionamiento de los monitores.

Funcionamiento de los monitores

Comunicación monitor-mundo exterior: Cuando un proceso necesita operar sobre un recurso compartido controlado por un monitor deberá realizar una llamada a uno de los procedimientos exportados por el monitor usando los parámetros actuales apropiados:

- Mientras el proceso está ejecutando algún procedimiento del monitor decimos que **el proceso está dentro del monitor**.

Exclusión mutua: Si un proceso P está dentro de un monitor, cualquier otro proceso Q que llame a un procedimiento de ese monitor deberá esperar hasta que P salga del mismo:

- Esta política de acceso asegura que las variables permanentes nunca son accedidas concurrentemente.
- El acceso exclusivo entre los procedimientos del monitor debe estar garantizado en la implementación de los monitores

Funcionamiento de los monitores (2)

Los monitores son objetos pasivos:
Después de ejecutarse el código de inicialización, un monitor es un objeto pasivo y el código de sus procedimientos sólo se ejecuta cuando estos son invocados por los procesos.

Instanciación de monitores:
En algunos casos es conveniente crear múltiples instancias independientes de un monitor, podemos hacerlo con esta sintaxis

```
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 ) ;
```

- Cada instancia tiene sus variables permanentes propias.
- La E.M. ocurre en cada instancia por separado.
- Esto facilita mucho escribir código reentrante.

Ejemplo de instanciación de una clase monitor

Similar a `VarCompartida`, pero ahora parametrizado:

```
{ 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
```

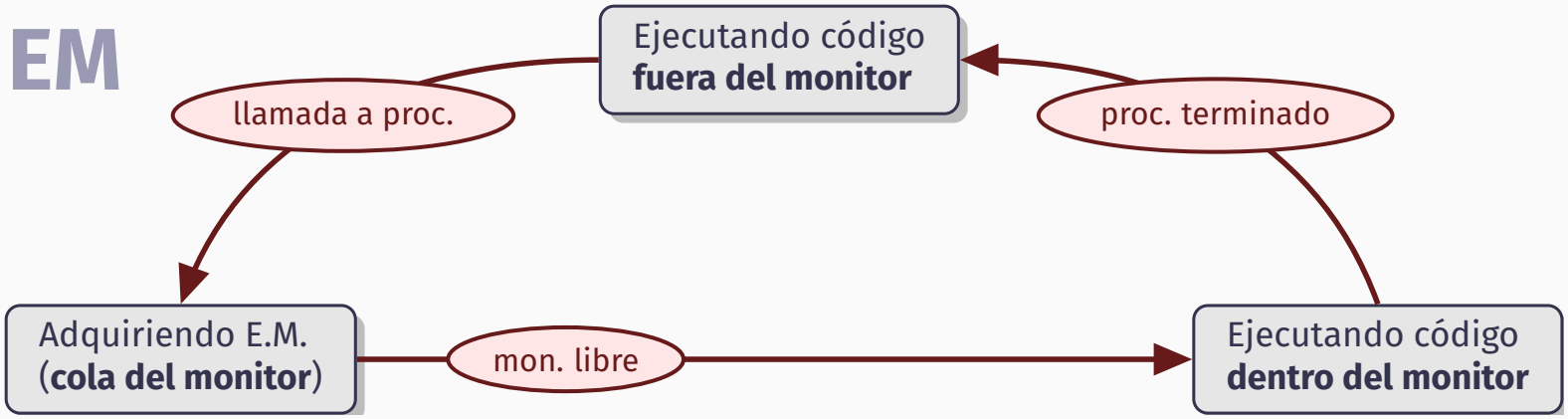
Cola del monitor para exclusión mutua

El control de la exclusión mutua se basa en la existencia de la **cola del monitor**:

- Si un proceso está *dentro del monitor* y otro proceso intenta ejecutar un procedimiento del monitor, éste último proceso queda bloqueado y se inserta en la cola del monitor
- Cuando un proceso *abandona el monitor* (finaliza la ejecución del procedimiento), se desbloquea un proceso de la cola, que ya puede entrar al monitor
- Si la cola del monitor está vacía, el monitor está *libre* y el primer proceso que ejecute una llamada a uno de sus procedimientos, entrará en el monitor
- Para garantizar la vivacidad del sistema, la planificación de la cola del monitor debe seguir una política FIFO

Diagrama de estados de un proceso

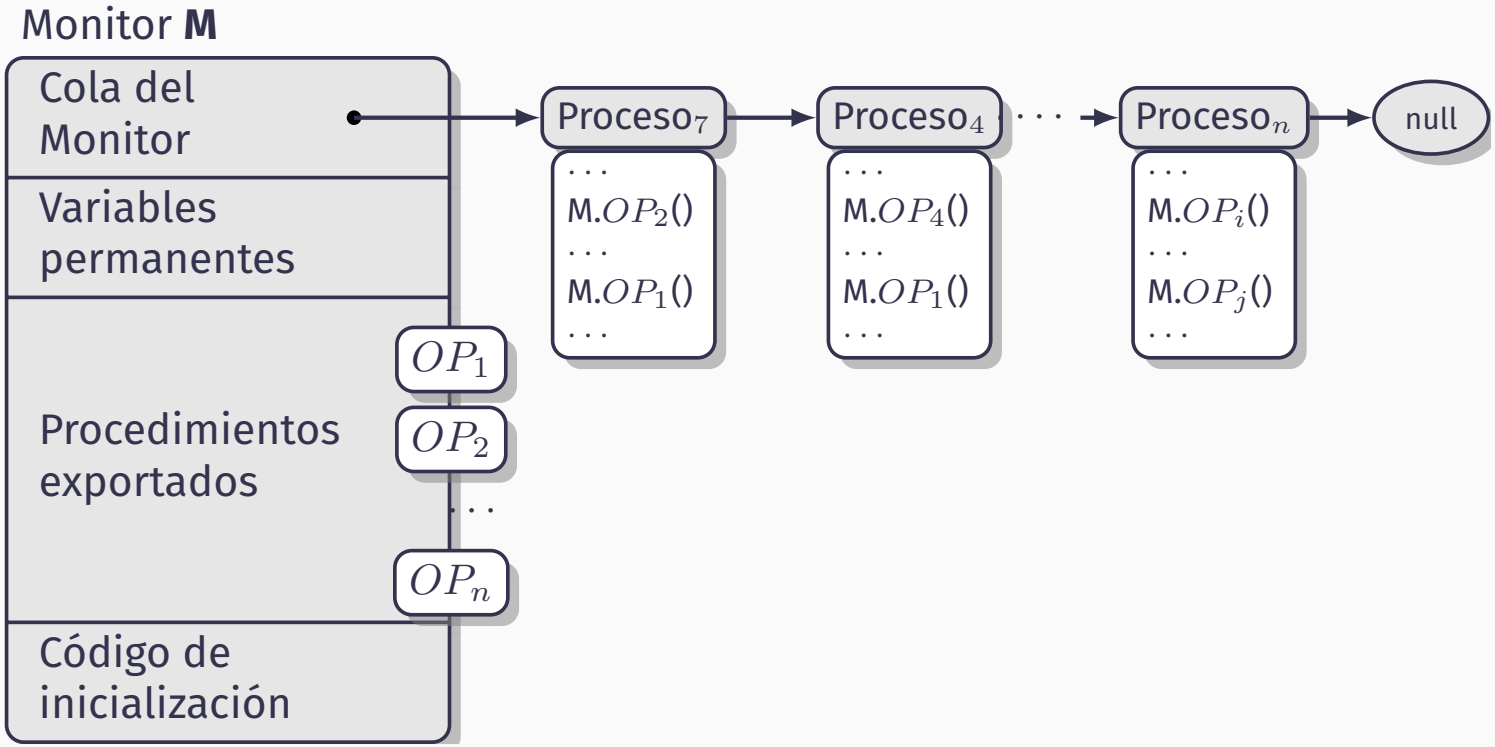
Los posibles estados de los procesos y las transiciones entre dichos estados se muestran en este diagrama:



(si el monitor está libre en el momento de la llamada al procedimiento del monitor, no habrá espera en la cola del monitor)

Estado del monitor

Por tanto, el estado del monitor incluye la cola de procesos esperando a comenzar a ejecutar el código del mismo. Se puede representar por este diagrama:



Subsección 3.3.
Sincronización en monitores..

Sincronización en monitores

Para implementar la sincronización, se requiere de una facilidad para que los procesos hagan esperas bloqueadas, hasta que sea cierta determinada condición:

En **semáforos**, existe

- la posibilidad de bloqueo (`sem_wait`) y activación (`sem_signal`)
- un valor entero (el valor del semáforo), que indica si la condición se cumple (> 0) o no ($== 0$).

En **monitores**, sin embargo

- sólo se dispone de sentencias de bloqueo y activación.
- los valores de las variables permanentes del monitor determinan si la condición se cumple o no se cumple.

Bloqueo y activación con variables condición

Para cada condición distinta que los procesos pueden eventualmente tener que esperar en un monitor, se debe de declarar una variable permanente de tipo **`condition`**. A esas variables las llamamos **señales** o **variables condición**:

- Cada variable condición tiene asociada una lista o cola (inicialmente vacía) de procesos en espera hasta que la condición se haga cierta.
- Para una cualquiera de estas variables, un proceso puede invocar estas dos operaciones:
  - **`wait`**: Estoy esperando a que alguna condición ocurra.
  - **`signal`**: Estoy señalando que una condición ocurre.

Operaciones sobre variables condición:

Dada una variable condición `cond`, se definen al menos las siguientes operaciones asociadas a `cond`:

- `cond.wait()`: bloquea **incondicionalmente** al proceso que la llama y lo introduce en la cola de la variable condición.
- `cond.signal()`: si hay procesos bloqueados por esa condición, libera uno de ellos. Si no hay ninguno esperando no hace nada. Si hay más de uno, se sigue una política **FIFO** (*first-in first-out*):
  - Se reactivará al proceso que lleve más tiempo esperando.
  - Esto **evita la inanición**: Cada proceso en cola obtendrá eventualmente su turno.
- `cond.queue()`: Función lógica que devuelve **`true`** si hay algún proceso esperando en la cola de `cond`, y **`false`** en caso contrario.

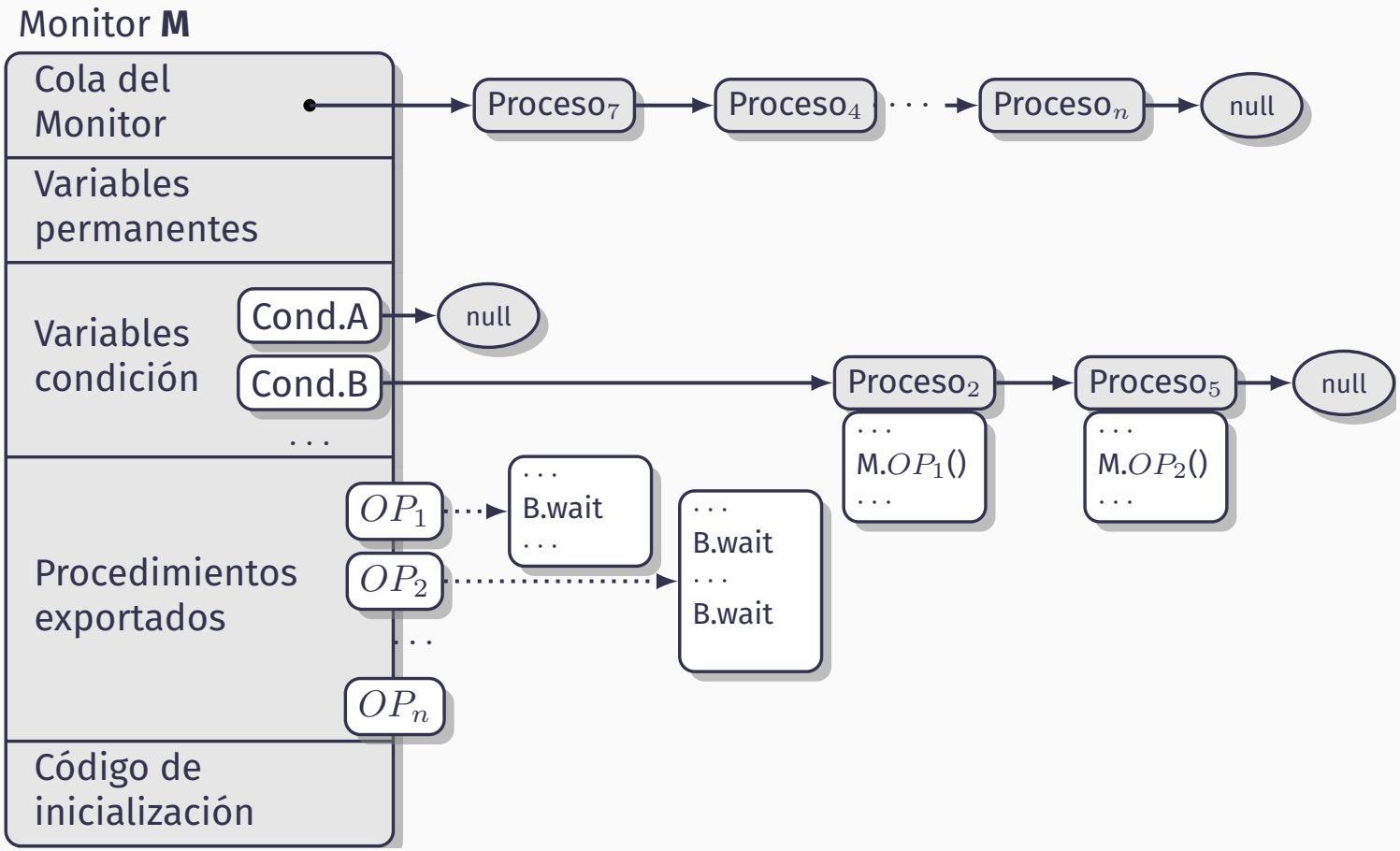
Esperas bloqueadas y E.M. en el monitor

Dado que los procesos pueden estar dentro del monitor, pero bloqueados:

- Cuando un proceso llama a **`wait`** y queda bloqueado, **se debe liberar la exclusión mutua del monitor**, si no se hiciese, se produciría interbloqueo con seguridad (ese proceso quedaría quedaría bloqueado y el resto también cuando intentasen después entrar al monitor).
- Cuando un proceso es reactivado despúes de una espera, adquiere de nuevo la exclusión mutua 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 variables condición.

Estado de un monitor con varias colas

A modo de ejemplo: los procesos 2 y 5 ejecutan las operaciones 1 y 2, ambas producen esperas de la condición B.



Subsección 3.4.
Verificación de monitores.

Verificación de programas con monitores

La verificación de la corrección de un programa concurrente con monitores requiere:

- Probar la corrección de cada monitor.
- Probar la corrección de cada proceso de forma aislada.
- Probar la corrección de la ejecución concurrente de los procesos implicados.

El programador no puede conocer *a priori* la interfoliación concreta de llamadas a los procedimientos del monitor. El enfoque de verificación que vamos a seguir utiliza un **invariante de monitor**:

- Es una propiedad que el monitor cumple siempre (parecido al invariante de un semáforo, pero específico de cada monitor diseñado por un programador)
- Unido a las propiedades de los procesos concurrentes que invocan al monitor, facilita la verificación de los programas

Características del Invariante del Monitor

El Invariante del Monitor (IM)

- Es un función lógica que se puede evaluar como **`true`** o **`false`** en cada estado del monitor a lo largo de la ejecución del programa concurrente.
- Su valor depende de la **traza** del monitor y de los valores de las **variables permanentes** de dicho monitor.
- La traza del monitor es la secuencia (ordenada en el tiempo) de llamadas a procedimientos del monitor ya completadas, desde el inicio del programa hasta llegar al estado indicado (se incluyen las llamadas de todos los procesos del monitor).
- El IM **debe ser cierto en cualquier estado del programa concurrente**, excepto cuando un proceso está ejecutando código del monitor, en E.M. (está en proceso de actualización de los valores de las variables permenentes).

Características del Invariante del Monitor

El invariante del monitor se escribe como un predicado

- Determina que trazas y/o estados son posibles en el monitor.
- Puede incluir referencias a las variables y/o a la traza, por ejemplo, al número de veces que se ha ejecutado un procedimiento, o al último procedimiento ejecutado.

El invariante del monitor debe ser cierto

- en el estado inicial, justo después de la inicialización de las variables permanentes.
- antes y después de cada llamada a un procedimiento del monitor
- antes y después de cada operación **`wait`**.
- antes y después de cada operación **`signal`**.

Justo antes de **`signal`** sobre una variable condición, además, debe ser cierta la condición lógica asociada a dicha variable.

Subsección 3.5.
Patrones de solución con monitores.

Patrones de uso de monitores

Para ilustrar el uso de monitores, veremos los patrones de solución para los mismos tres problemas típicos sencillos cuya soluciones con semáforos ya expusimos:

- **Espera única**: un proceso, antes de ejecutar una sentencia, debe esperar a que otro proceso complete otra sentencia (ocurre típicamente cuando un proceso debe leer una variable escrita por otro proceso, el primero se suele denominar **Consumidor** y el segundo **Productor**).
- **Exclusión mutua**: acceso en exclusión mutua a una sección crítica por parte de un número arbitrario de procesos.
- **Problema del Productor/Consumidor**: similar a la espera única, pero de forma repetida en un bucle (un proceso Productor escribe sucesivos valores en una variable, y cada uno de ellos debe ser leído una única vez por otro proceso Consumidor).

Monitor para espera única

Un proceso (consumidor) no puede ejecutar una sentencia L (típicamente una lectura) hasta que otro proceso (productor) no acabe otra sentencia E (típicamente una escritura):

```
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
```

Uso del monitor de espera única

El 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
```

- El proceso **Consumidor** espera, antes de leer, a que el **Productor** termine la sentencia de escritura.
- De esta forma nos aseguramos que ocurre la interfoliación E, L y no puede ocurrir la interfoliacion L, E.

Notación para verificación

Durante la ejecución del programa concurrente el estado va cambiando conforme se ejecutan las sentencias atómicas. En un estado cualquiera, llamamos $\#I$ al número de veces que se ha completado la ejecución de una sentencia I desde el inicio del programa. Inicialmente $\#I$ es 0. En un estado

- $\#N$ es el número de llamadas a **`notificar`** terminadas.
- $\#W$ es el número de llamadas a **`esperar`** terminadas.
- La estructura del proceso implica que $\#N \leq 1$ y $\#W \leq 1$.
- En la interfoliación E, L se cumple $\#L \leq \#E$ siempre. En la interfoliación L, E (prohibida), no se cumple.
- Luego queremos demostrar que siempre $\#L \leq \#E$. Como veremos, esto es equivalente a demostrar que $\#W \leq \#N$.
- Para demostrar que $\#W \leq \#N$, usamos el invariante del monitor.

Invariante del monitor, corrección del programa

En este ejemplo se cumple siempre este **invariante del monitor**

$$\texttt{terminado} == \texttt{false} \implies \#N = \#W = 0$$
$$\texttt{terminado} == \texttt{true} \implies \#N > 0$$

- Puesto que **`terminado`** solo puede ser **`true`** o **`false`**, en ningún caso puede ocurrir una traza en la que $\#W > 0$ y $\#N = 0$
- Esto significa que no se puede completar ninguna llamada a W hasta que no se haya completado al menos una llamada a N (este es el comportamiento esperado del monitor)
- Por lo tanto, el invariante implica $\#W \leq \#N$.
- El productor ejecuta E, N, luego $\#N \leq \#E$.
- El consumidor ejecuta W, L, luego $\#L \leq \#W$
- Encadenando las tres desigualdades, obtenemos:

$$\#L \leq \#E$$

Demostración del invariante

Para demostrar que el invariante del monitor (IM) es cierto, se verifican estas condiciones:

- **El invariante es inicialmente cierto**. En este caso, inicialmente se cumple `terminado=false` y $\#N = \#W = 0$.
- **Si es cierto antes de llamar a `notificar`, lo será también después**. Se demuestra viendo los dos casos posibles previos a la llamada. Si `terminado==true`, entonces $\#N > 0$ antes y después. Si `terminado==false`, entonces $\#N$ pasa de 0 a 1 y `terminado` se hace `true` (se cumple el IM después).
- **Si es cierto antes de llamar a `esperar`, lo será también después**. Esto se verifica viendo que en `esperar` se espera hasta que `terminado` sea `true`, y en ese caso el IM no exige nada sobre $\#W$, luego se cumple después de `esperar`.

Patrón para exclusión mutua

Este monitor sirve para ejecutar una SC en exclusión mutua.

```
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
```

Uso y propiedades del monitor

El monitor anterior puede ser usado por n procesos concurrentes:

```
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
```

- $\#E$ es el número de llamadas a **entrar** completadas.
- $\#S$ es el número de llamadas a **salir** completadas.
- El número de procesos en SC es $\#E - \#S$.
- La única interfoliación correcta es $E, S, E, S, \ldots$.
- Se debe cumplir $0 \leq \#E - \#S \leq 1$

Invariante del monitor

El **invariante del monitor**, es la conjunción de estas dos condiciones:

$$\texttt{ocupada} == \texttt{true} \iff U == E$$

E no aparece dos o más veces seguidas en la traza

Donde U designa al último procedimiento del monitor ejecutado en la traza (solo puede ser E o S). El IM implica que no pueden entrar dos o más procesos a la SC, ya que eso implicaría una traza con dos o más E consecutivas. El IM es fácil de demostrar:

- Al inicio es cierto (la traza está vacía, luego $U \neq E$ y `ocupada == false`).
- Al completar S, `ocupada` toma el valor `false` (se cumple el IM)
- Antes de completar E, se espera a que `ocupada` sea `false`, es decir, a que $U \neq E$, finalmente se pone `ocupada` a `true`. Por tanto, se cumple el invariante y se impide que E se ejecute dos veces seguidas.

Correción del programa

Queremos probar que el programa concurrente es correcto siempre, usamos las propiedades de los procesos de forma conjunta con el invariante del monitor:

- Por la estructura de los procesos, sabemos que S se ejecuta una única vez después de cada E, luego se cumple $\#S \leq \#E$, es decir $0 \leq \#E - \#S$
- Por el invariante, sabemos que E no se puede ejecutar dos veces seguidas sin que se ejecute S entre ellas, luego se cumple $\#E - \#S \leq 1$
- Como consecuencia, se cumple la propiedad deseada del programa concurrente:

$$0 \leq \#E - \#S \leq 1$$

Sincronización tipo Productor/Consumidor

El problema del Productor-Consumidor con las lecturas y escrituras (E y L) repetidas en un bucle puede solucionarse con un monitor (lo llamamos **PC**) sencillo, que encapsula el valor compartido:

```
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 el parámetro en la variable compartida
- El procedimiento **`leer`** lee el valor que hay en la variable compartida

Sincronización tipo Productor/Consumidor

La forma del monitor **PC** es esta:

```
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
```

Verificación del monitor

Al igual que en los otros casos, podemos verificar que el monitor funciona bien

- $\#E =$ número de llamadas a **`escribir`** completadas.
- $\#L =$ número de llamadas a **`leer`** completadas.
- El monitor es correcto solo si en cualquier estado

$$0 \leq \#E - \#L \leq 1$$

- El invariante del monitor es:

$$\#E - \#L = \begin{cases} 0 & \text{si pendiente == false} \\ 1 & \text{si pendiente == true} \end{cases}$$

Si el invariante es cierto, el monitor es correcto (ya que **`pendiente`** solo puede ser **`true`** o **`false`**). Solo quedaría probar el invariante.

Demostración del invariante

Se cumplen las siguientes condiciones lógicas:

- Al inicio, se cumple el invariante, ya que $\#E$ y $\#L$ son 0, y **`pendiente`** es **`false`**.
- Si se cumple el invariante y se ejecuta **`escribir`**, se sigue cumpliendo el invariante después, ya que se espera a que **`pendiente`** sea **`false`**, es decir, a que $\#E - \#L = 0$, y luego se pone **`pendiente`** a **`true`**, por tanto al acabar $\#E - \#L = 1$ y **`pendiente==true`**.
- Si se cumple el invariante y se completa **`leer`**, se sigue cumpliendo el invariante después, ya que se espera a que **`pendiente`** sea **`true`**, es decir, a que $\#E - \#L = 1$, y luego se pone **`pendiente`** a **`false`**, por tanto al acabar $\#E - \#L = 0$ y **`pendiente==false`**.

Subsección 3.6.
Colas de prioridad.

Colas de condición con prioridad

Por defecto, se usan colas de espera FIFO. Sin embargo a veces resulta útil disponer de un mayor control sobre la estrategia de planificación, dando la prioridad del proceso en espera como un parámetro entero de **`wait`**

- La sintaxis de la llamada es: `cond.wait(`p`)`, donde p es un entero no negativo que refleja la prioridad.
- `cond.signal()` reanudará un proceso que especificó el valor mínimo de p de entre todos los que esperan (si hay más de uno con prioridad mínima, se usa política FIFO).
- Se deben evitar riesgos como la inanición.
- No tiene ningún efecto sobre la lógica del programa: el funcionamiento es similar con y sin colas de prioridad.
- Sólo mejoran las características dependientes del tiempo.

Ejemplo de cola con prioridad: uso de un recurso

Este monitor es similar al anterior, solo que ahora los procesos, cuando adquieren un recurso, especifican un valor entero de duración de uso del recurso. Se da prioridad a los que lo van a usar durante 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
```

Ejemplo de cola con prioridad (2): Reloj con alarma

El proceso llamador se retarda **n** unidades de tiempo:

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

```
procedure despiertame( 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
```

Subsección 3.7.
El problema de los Lectores/Escritores.

El problema de los Lectores/Escritores (LE)

Dos tipos de procesos acceden concurrentemente a datos compartidos:

- **Escritores**: procesos que modifican la estructura de datos (escriben en ella). El código de escritura no puede ejecutarse concurrentemente con ninguna otra escritura ni lectura, ya que está formado por una secuencia de instrucciones que temporalmente ponen la estructura de datos en un estado no usable por otros procesos.
- **Lectores**: procesos que leen la estructura de datos, pero no modifican su estado en absoluto. El código de lectura puede (y debe) ejecutarse concurrentemente por varios lectores de forma arbitraria, pero no puede hacerse a la vez que la escritura.

La solución de este problema usando semáforos es compleja, veremos que con monitores es sencillo.

Vars. permanentes y procedimientos para lectores

El monitor se puede implemenar como vemos aquí:

```
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
```

Procedimientos para escritores

Los procedimientos para escritores son estos dos:

```
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
```

Uso del monitor

Los procesos lectores y escritores usan el monitor de esta forma:

```
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
```

- En esta implementación se ha dado 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.

Subsección 3.8.
Semántica de las señales de los monitores.

La exclusión mutua tras una señal. Semántica.

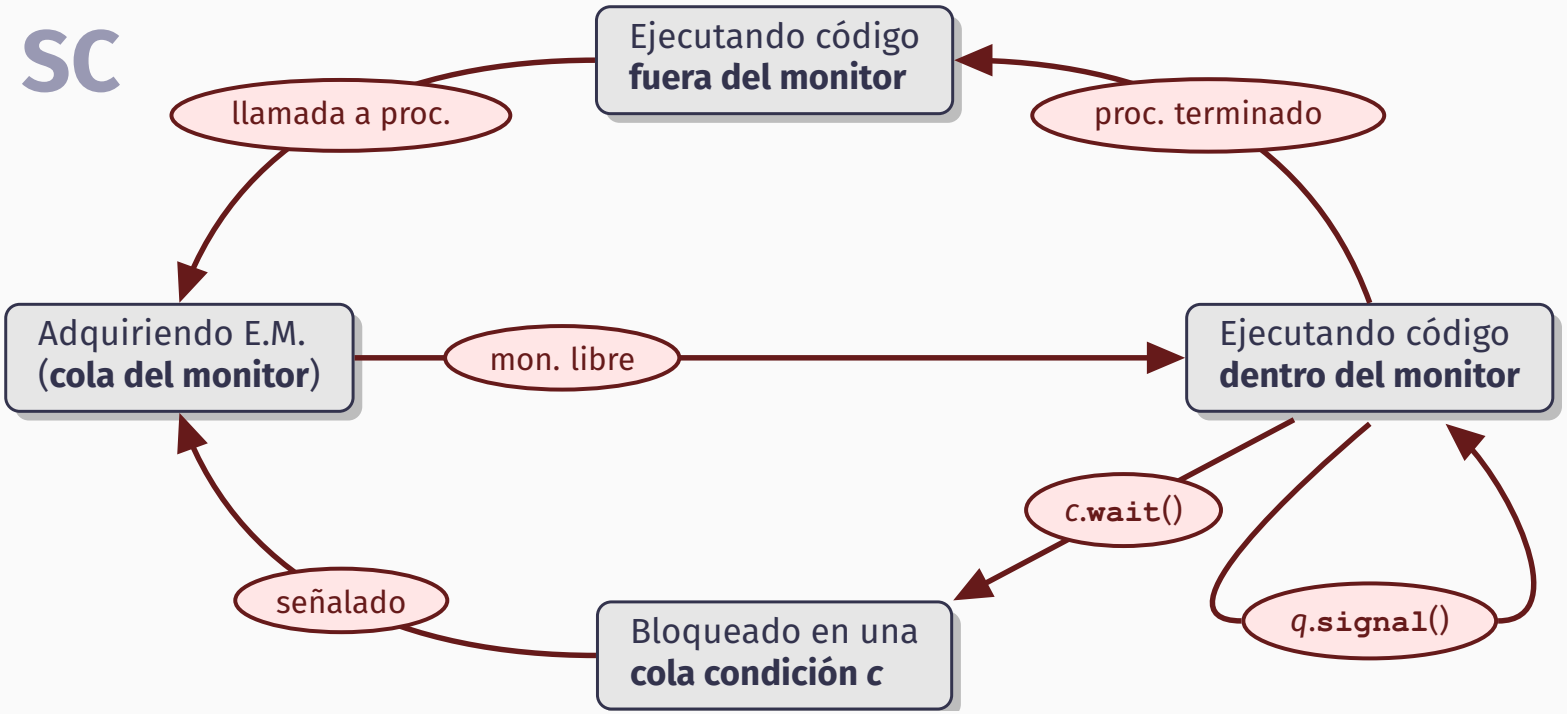
Cuando un proceso hace **`signal`** en una cola no vacía, se denomina proceso **señalador**. El proceso que esperaba en la cola y que se reactiva se denomina **señalado**:

- Suponemos que hay código restante del monitor tras el **`wait`** y tras el **`signal`**.
- Inmediatamente después de señalar, **no es posible** que ambos (señalador y señalado) continuen la ejecución de su código restante, ya que no se cumpliría la exclusión mutua del monitor.
- Uno de los dos puede inmediatamente ejecutar su código restante, pero entonces el otro no puede hacerlo.
- Se denomina **semántica de señales** a la política que establece la forma concreta en que se resuelve el conflicto tras hacerse un **`signal`** en una cola no vacía.

Posibles semánticas de las señales. Esquema.

- **El proceso señalador continua la ejecución** tras el **`signal`**. El señalado espera bloqueado hasta que puede adquirir la E.M. de nuevo (SC: **señalar y continuar**).
- **El proceso señalado se reactiva inmediatamente**. El señalador
  - abandona el monitor tras hacer **`signal`** sin ejecutar el código que haya después de dicho **`signal`** (SS: **señalar y salir**).
  - queda bloqueado a la espera en
    - la cola del monitor, junto con otros posibles procesos que quieren comenzar a ejecutar código del monitor (SE: **señalar y esperar**).
    - una cola específica para esto, con mayor prioridad que esos otros procesos (SU: **señalar y espera urgente**).

Señalar y continuar (SC): diagrama de estados del proceso



- **Señalador:** continúa inmeditamente la ejecución de código del monitor tras **`signal`**.
- **Señalado:** abandona la cola condición y espera en la **cola del monitor** hasta readquirir la E.M. y ejecutar código tras **`wait`**.

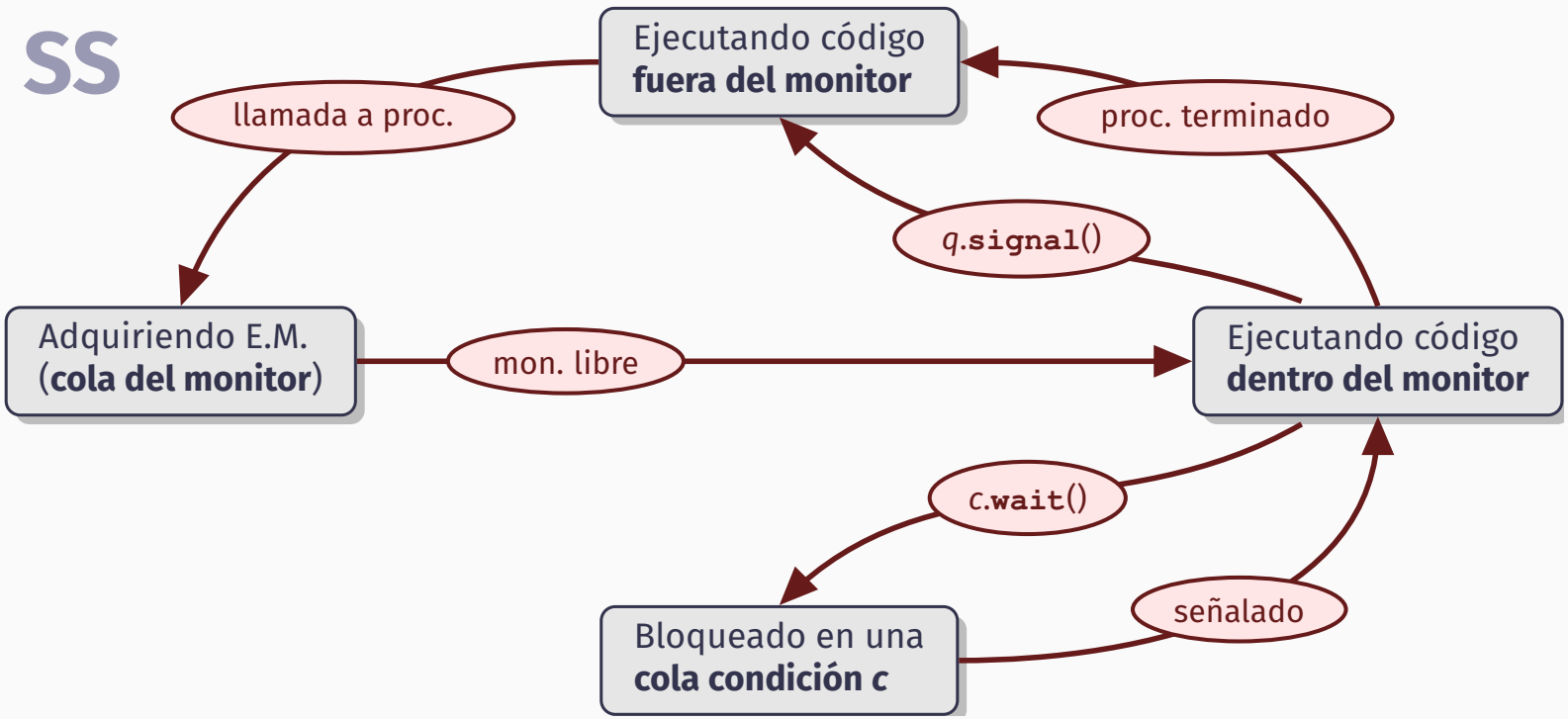
Señalar y continuar (SC): características

El proceso señalador continúa su ejecución dentro del monitor después del signal. El proceso señalado abandona la cola condición y espera en la cola del monitor para readquirir la E.M.

- Tanto el señalador como otros procesos pueden hacer falsa la condición despues de que el señalado abandone la cola condición.
- Por tanto, en el proceso señalado no se puede garantizar que la condición asociada a `cond` es cierta al terminar `cond.wait()`, y lógicamente es necesario volver a comprobarla entonces.
- Esta semántica obliga a programar la operación **`wait`** en un bucle, de la siguiente manera:

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

Señalar y salir (SS): diagrama de estados del proceso



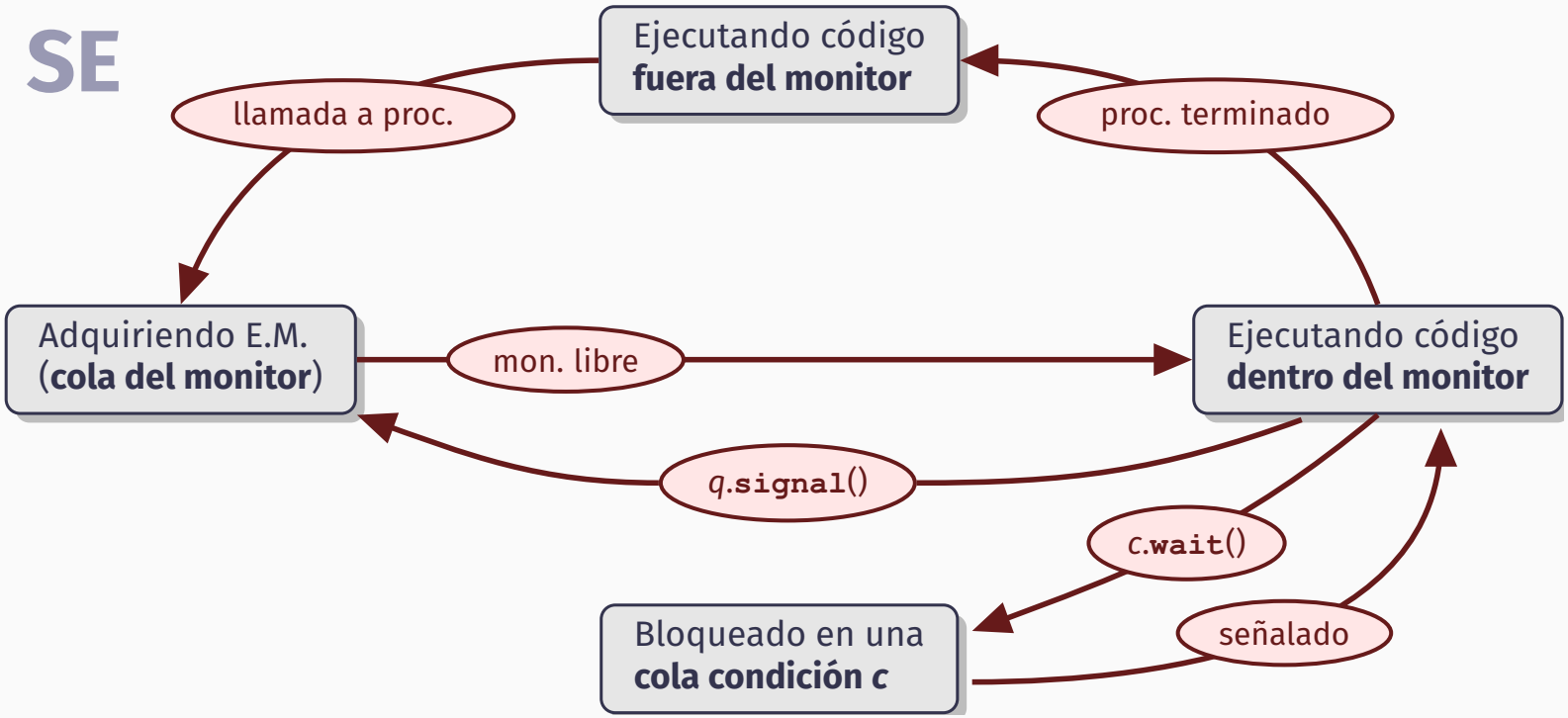
- **Señalador:** abandona el monitor (ejecuta código tras la llamada al procedimiento del monitor). Si hay código en el monitor tras **`signal`**, no se ejecuta.
- **Señalado:** reanuda inmediatamente la ejecución de código del monitor tras **`wait`**.

Señalar y salir (SS): características

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

- En ese caso, la operación **`signal`** conlleva:
  - Liberar al proceso señalado.
  - Terminación del procedimiento del monitor que estaba ejecutando el proceso señalador.
- Está asegurado el estado que permite al proceso señalado continuar la ejecución del procedimiento del monitor en el que se bloqueó (la condición de desbloqueo se cumple).
- Esta semántica condiciona el estilo de programación ya que obliga a colocar **siempre la operación `signal` como última instrucción** de los procedimientos de monitor que la usen.

Señalar y esperar (SE): diagrama de estados del proceso



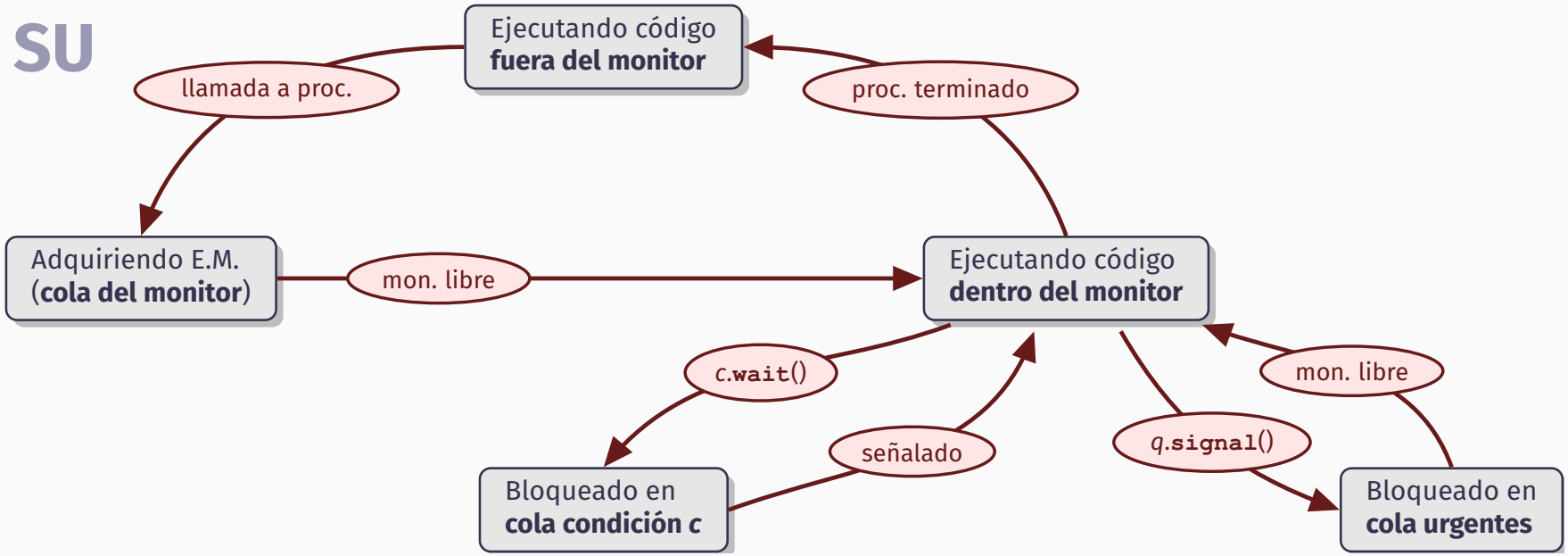
- **Señalador:** se bloquea en la **cola del monitor** hasta readquirir E.M. y ejecutar el código del monitor tras **`signal`**.
- **Señalado:** reanuda inmediatamente la ejecución de código del monitor tras **`wait`**.

Señalar y esperar (SE): características

El proceso señalador se bloquea en la cola del monitor justo después de ejecutar **`signal`**. El proceso señalado entra de forma inmediata en el monitor.

- Está asegurado el estado que permite al proceso señalado continuar la ejecución del procedimiento del monitor en el que se bloqueó.
- El proceso señalador entra en la cola de procesos del monitor, por lo que está al mismo nivel que el resto de procesos que compiten por la exclusión mutua del monitor.
- Puede considerarse una semántica *injusta* respecto al proceso señalador ya que dicho proceso ya había obtenido el acceso al monitor por lo que debería tener prioridad sobre el resto de procesos que compiten por el monitor.

Señalar y espera urgente (SU): diagrama de estados del proc.



- **Señalador:** se bloquea en la **cola de urgentes** hasta readquirir la E.M. y ejecutar código del monitor tras **`signal`**. Para readquirir E.M. tiene más prioridad que los procesos en la cola del monitor.
- **Señalado:** reanuda inmediatamente la ejecución de código del monitor tras **`wait`**.

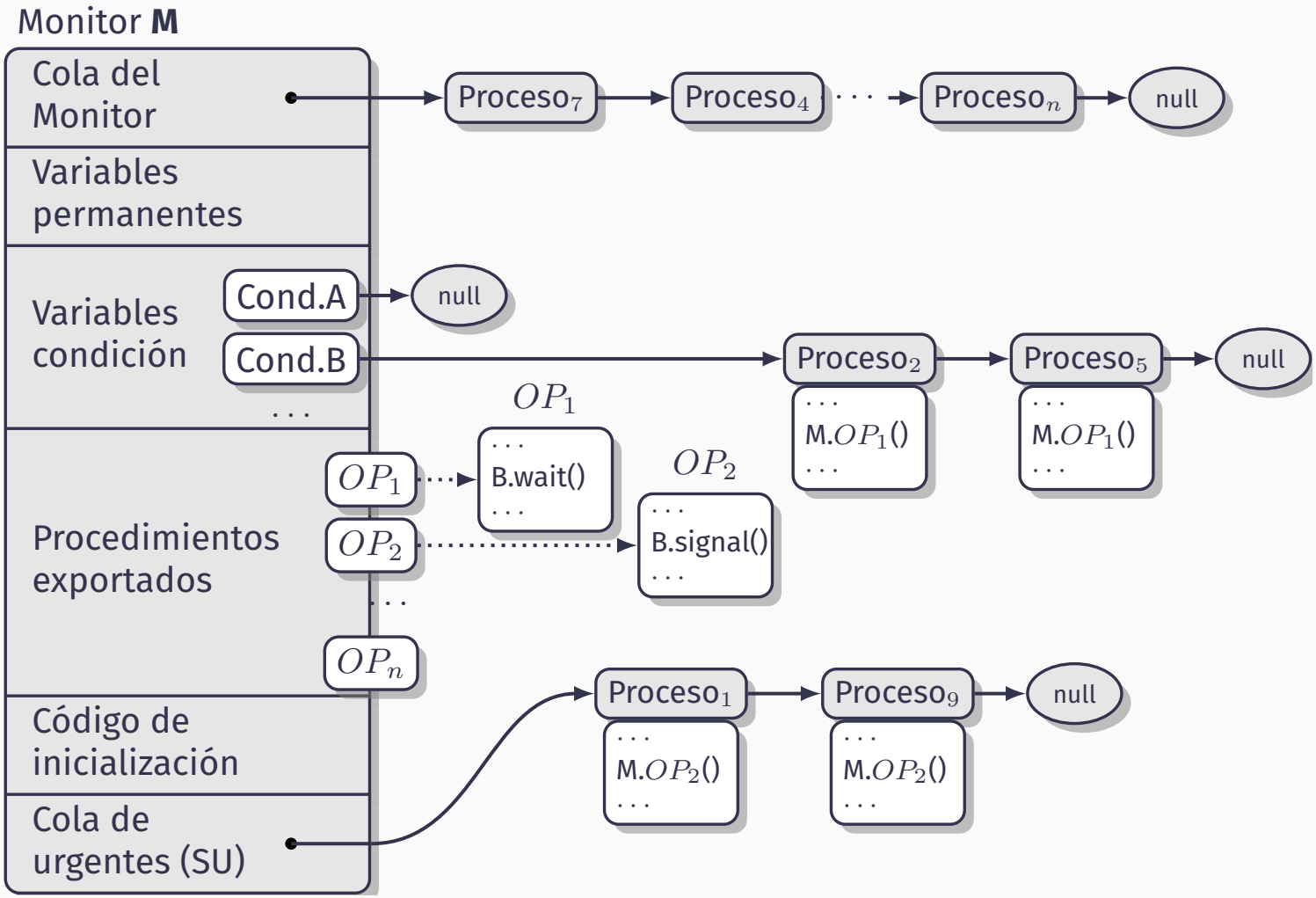
Señalar y espera urgente (SU): características

Es similar la semántica SE, pero se intenta corregir el problema de falta de equitatividad indicado:

- El proceso señalador se bloquea justo después de ejecutar la operación **`signal`**.
- El proceso señalado entra de forma inmediata en el monitor.
- Está asegurado el estado que permite al proceso señalado continuar la ejecución del procedimiento del monitor en el que se bloqueó.
- El proceso señalador entra en una nueva cola de procesos que esperan para acceder al monitor, que podemos llamar *cola de procesos urgentes*.
- Los procesos de la cola de procesos urgentes tienen preferencia para acceder al monitor frente a los procesos que esperan en la cola del monitor.
- Es la semántica que se supone en los ejemplos vistos.

Procesos en la cola de urgentes.

El proceso 1 y el 9 han ejecutado la op.2, que hace **`signal`** de la cond. B.



Análisis comparativo de las diferentes semánticas

- **Potencia expresiva**: todas las semánticas son capaces de resolver los mismos problemas.
- **Facilidad de uso**: La semántica **SS** condiciona el estilo de programación y puede llevar a aumentar de forma *artificial* el número de procedimientos.
- **Eficiencia**:
  - Las semánticas **SE** y **SU** resultan ineficientes cuando no hay código tras **`signal`**, ya que en ese caso implican que el señalador emplea tiempo en bloquearse y después reactivarse, pero justo a continuación abandona el monitor sin hacer nada.
  - La semántica **SC** también es un poco ineficiente al obligar a usar un bucle para cada instrucción **`signal`**.

Ejemplo comparativo de semánticas: barrera parcial

Como ejemplo de las diferencias entre las distintas semánticas, veremos un ejemplo de un monitor que llamamos **barrera parcial**.

- El monitor tiene un único procedimiento público llamado **`cita`**.
- Hay p procesos ejecutando un bucle infinito, en cada iteración realizan una actividad de duración arbtraria y después llaman a **`cita`**.
- Ningún proceso termina **`cita`** antes de que haya al menos n de ellos que la hayan inciado (donde $1 < n < p$). Después de esperar en **`cita`**, pero antes de terminarla, el proceso imprime un mensaje.
- Cada vez que un grupo de n procesos llegan a la cita, esos n procesos imprimen su mensaje antes de que lo haga ningún otro proceso que haya llegado después de todos ellos a dicha cita (que sea del siguiente grupo de n).

Diseño del monitor de barrera parcial

Cuando un proceso comienza a ejecutar **`cita`**, debe esperar hasta que haya otros $n-1$ que tambien lo hayan hecho:

- Eso supone usar una variable condición (la llamamos **`cola`**)
- Para saber si hay que esperar o no, es necesario saber cuantos procesos han llegado a la cita pero no la han terminado todavía, para ello usamos una variable entera (**`contador`**), inicialmente a 0. Al entrar en la cita, debe incrementarse.
- Por tanto, la condición lógica asociada a la variable condición **`cola`** es **`contador`**`==`n.
- El proceso que, tras llegar a la cita e incrementar, observa que **`contador`**`==`n, debe de encargarse de que los procesos en espera abandonan todos dicha espera y terminen **`cita`** (puesto que ya se cumple la condición que esperan).

Una posible implementación del monitor

Del diseño que hemos visto se deduce esta posible implementación:

```
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
```

Comportamiento del monitor en distintas semánticas (1)

Se puede analizar el comportamiento de este monitor en las distintas semánticas. Para ello, llamamos *último* al último proceso en llegar a la cita de cada grupo de n (el que observa `contador==`n).

- **Señalar y Continuar**: el proceso último hace todos los `signal` seguidos sin esperar entre ellos, y pone `contador` a 0. Los $n-1$ procesos señalados abandonan el `wait`, pero pasan a la cola del monitor. Por tanto, antes de que esos señalados puedan terminar de ejecutar `cita`, todos los procesos del siguiente grupo de n podrían iniciar y terminar dicha cita, y no se cumple el segundo requerimiento.
- **Señalar y Salir**: en este caso, el proceso último abandona el monitor tras el primer `signal`, por tanto no pone `contador` a 0, no hace el resto de `signal` y el siguiente proceso en llegar (primero del siguiente grupo), no hace la espera requerida.

Comportamiento del monitor en distintas semánticas (2)

Para las otras dos semánticas restantes, hacemos este análisis:

- **Señalar y Esperar**: el proceso último, después del primer `signal`, va a la cola del monitor. Por tanto, podría entrar a la cita el primer proceso del siguiente grupo y observar `contador`$==n$, con lo cual ese primero no hace la espera requerida.
- **Señalar y Espera Urgente**: el proceso último hace todos los signals. Entre cada dos de ellos, espera en la cola de urgentes a que el señalado abandone el monitor. Los procesos del siguiente grupo esperan en la cola del monitor hasta que todos los señalados lo abandonen y el último ponga el contador a cero. Luego esta solución **es correcta con semántica SU**.

Una implementación alternativa

Ahora los procesos señalados se despiertan en cadena entre ellos, y **contador** es el número de procesos en la cola:

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

 procedure cita() ;
 begin
   contador := contador+1 ;       { registrar un proceso más esperando en cola }
   if contador < n then           { si todavía no hay n procesos: }
      cola.wait();                {    esperar a que los haya  }
   contador := contador-1;        { registrar un proceso menos esperando en cola }
   print("salgo de la cita");     { mensaje de salida }
   if contador > 0 then           { si hay otros procesos en la cola }
         cola.signal();           {     despertar al siguiente }
 end
begin                             { inicialización: }
 contador := 0 ;                  {    inicialmente, no hay procesos en la cola }
end
```

Comportamiento de la versión alternativa

Analizamos el comportamiento en todas las semánticas:

- **No funciona con la semántica SC**, ya que al salir del `wait` los señalados vuelven a la cola del monitor, donde ya podría haber esperando procesos del siguiente grupo que entrarían a la cita antes de que los del grupo actual puedan completar su ejecución tras `wait`.
- **Sí funciona con el resto de semánticas (SE,SS,SU)**: en todos los casos, los procesos señalados completan la ejecución de `cita` inmediatamente después de salir del `wait`. En ningún caso los procesos del siguiente grupo tienen opción de *colarse* en la cita antes de tiempo.

En general, hay que ser cuidadoso con la semántica en uso, especialmente si el monitor tiene código tras `signal`.
Generalmente, la semántica SC puede complicar mucho los diseños.

Subsección 3.9.
Implementación de monitores.

Implementación de monitores con semáforos

Es posible implementar cualquier monitor usando exclusivamente semáforos. Cada cola tendrá un semáforo asociado:

- **Cola del monitor**: se implementa con un semáforo tipo mutex (vale 0 si algún proceso está ejecutando código, 1 en otro caso)
- **Colas de variables condición**: para cada variable condición será necesario definir un semáforo (que está siempre a 0) y una variable entera que indica cuantos procesos hay esperando.
- **Cola de procesos urgentes**: en semántica SU, debe haber un entero y un semáforo (siempre a 0) adicionales.

Limitación: esta implementación no permite llamadas recursivas a los procedimientos del monitor y no asegura orden FIFO en las colas.

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

Implementación de monitores con semáforos

La exclusión mutua en el acceso a los procs. del monitor se puede implementar con un único semáforo **mutex** inicializado a 1:

```
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 entero **n_urgentes** (núm. de 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 ;
```

Implementación de monitores con semáforos

Para variable condición (de nombre, por ejemplo, **cond**) definimos un semáforo asociado (lo llamamos **cond_sem**, su valor siempre es 0) y una variable para contar los procesos bloqueados en ese semáforo (la llamamos **n_cond**, inicializada a 0):

Implementación de **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 ;
```

Implementación de **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
```

Sección 4.
Soluciones software con espera ocupada para E.M..

Introducción

En esta sección veremos diversas soluciones para lograr exclusión mutua en una sección crítica usando variables compartidas entre los procesos o hebras involucrados.

- Estos algoritmos usan dichas variables para hacer espera ocupada cuando sea necesario en el protocolo de entrada.
- Los algoritmos que resuelven este problema no son triviales, y menos para más de dos procesos. En la actualidad se conocen distintas soluciones con distintas propiedades. Veremos estos dos algoritmos:
  - **Algoritmo de Dekker** (para 2 procesos)
  - **Algoritmo de Peterson** (para 2 y para un número arbitrario de procesos).
- Previamente a esos algoritmos, veremos la estructura de los procesos con secciónes críticas y las propiedades que deben cumplir los algoritmos.

Subsección 4.1.
Estructura de los procesos con secciones críticas.

Entrada y salida en secciónes críticas

Para analizar las soluciones a EM asumimos que un proceso que incluya un bloque considerado como sección crítica (SC) tendrá dicho bloque estructurado en tres etapas:

1. **Protocolo de entrada (PE):** una serie de instrucciones que incluyen posiblemente espera, en los casos en los que no se pueda conceder acceso a la sección crítica.
2. **Sección crítica (SC):** instrucciones que solo pueden ser ejecutadas por un proceso como mucho.
3. **Protocolo de salida (PS):** instrucciones que permiten que otros procesos puedan conocer que este proceso ha terminado la sección crítica.

Todas las sentencias que no forman parte de ninguna de estas tres etapas se denominan **Resto de Sentencias (RS)** .

Acceso repetitivo a las seccióones críticas

En general, un proceso puede contener más de una sección crítica, y cada sección crítica puede estar desglosada en varios bloques de código separados en el texto del proceso. Para simplificar el análisis, suponemos, sin pérdida de generalidad, que:

- Cada proceso tiene una única sección crítica.
- Dicha sección crítica está formada por un único bloque contiguo de instrucciones.
- El proceso es un bucle infinito que ejecuta en cada iteracción dos pasos:
  - Sección crítica (con el PE antes y el PS después)
  - Resto de sentencias: se emplea un tiempo arbitrario no acotado, e incluso el proceso puede finalizar en esta sección, de forma prevista o imprevista.

es el caso más general: no se supone nada acerca de cuantas veces un proceso puede intentar entrar en una SC.

Condiciones sobre el comportamiento de los procesos.

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

> *Los procesos siempre terminan una sección crítica y emplean un intervalo de tiempo finito desde que la comienzan hasta que la terminan.*

es decir, durante el tiempo en que un proceso se encuentra en una sección crítica nunca

- Finaliza o aborta.
- Es finalizado o abortado externamente.
- Entra en un bucle infinito.
- Es bloqueado o suspendido indefinidamente de forma externa.

En general, es deseable que el tiempo empleado en las seccióones críticas sea el menor posible.

Subsección 4.2.
Propiedades para exclusión mutua.

Propiedades requeridas para las soluciones a EM

Para que un algoritmo para EM sea **correcto**, se deben cumplir cada una de 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.

Propiedad de exclusión mutua

Es la propiedad fundamental para el problema de la sección crítica. Establece que

> *En cada instante de tiempo, y para cada sección crítica existente, habrá como mucho un proceso ejecutando alguna sentencia de dicha región crítica.*

En esta sección veremos soluciones de memoria compartida que permiten un único proceso en una sección crítica.

Si bien esta es la propiedad fundamental, no puede conseguirse de cualquier forma, y para ello se establecen las otras dos condiciones mínimas que vemos a continuación.

Propiedad de progreso

Consideremos una SC en un instante en el cual no hay ningún proceso ejecutándola, pero sí hay procesos en el PE compitiendo por entrar a la SC. La **propiedad de progreso** establece:

Un algoritmo de EM debe estar diseñado de forma tal que:

1. Después de un intervalo de tiempo finito desde que ingresó el primer proceso al PE, uno de los procesos en el mismo podrá acceder a la SC.
2. La selección del proceso anterior es completamente independiente del comportamiento de los procesos que durante todo ese intervalo no han estado en SC ni han intentado acceder.

Cuando la condición (1) no se da, se dice que ocurre un **interbloqueo**, ya que todos los procesos en el PE quedan en espera ocupada indefinidamente sin que ninguno pueda avanzar.

Espera limitada

Supongamos que un proceso emplea un intervalo de tiempo en el PE intentando acceder a una SC. Durante ese intervalo de tiempo, cualquier otro proceso activo puede entrar un número arbitrario de veces n a ese mismo PE y lograr acceso a la SC (incluyendo la posibilidad de que $n = 0$). La propiedad de **espera limitada** establece que:

> *Un algoritmo de exclusión mutua debe estar diseñado de forma que n nunca será superior a un valor máximo determinado.*

Esto implica que las esperas en el PE siempre serán finitas (suponiendo que los procesos emplean un tiempo finito en la SC).

Propiedades deseables: eficiencia y equidad.

Las propiedades deseables son estas dos:

- **Eficiencia**:
  Los protocolos de entrada y salida deben emplear poco tiempo de procesamiento (excluyendo las esperas ocupadas del PE), y las variables compartidas deben usar poca cantidad de memoria.
- **Equidad**:
  En los casos en que haya varios procesos compitiendo por acceder a una SC (de forma repetida en el tiempo), no debería existir la posibilidad de que sistemáticamente se perjudique a algunos y se beneficie a otros.

Subsección 4.3.
Refinamiento sucesivo de Dijkstra.

Introducción al refinamiento sucesivo de Dijkstra

El **Refinamiento sucesivo de Dijskstra** hace referencia a una serie de algoritmos que intentan resolver el problema de la exclusión mutua.

- Se comienza desde una versión muy simple, incorrecta (no cumple alguna de las propiedades), y se hacen sucesivas mejoras para intentar cumplir las tres propiedades. Esto ilustra muy bien la importancia de dichas propiedades.
- La versión final correcta se denomina **Algoritmo de Dekker**.
- Por simplicidad, veremos algoritmos para 2 procesos únicamente.
- Se asume que hay dos procesos, denominados `P0` y `P1`, cada uno de ellos ejecuta un bucle infinito conteniendo: protocolo de entrada, sección crítica, protocolo de salida y otras sentencias del proceso.

Versión 1. Pseudocódigo.

En esta versión se usa una variable lógica compartida (**p01sc**) que valdrá **true** solo si el proceso 0 o el proceso 1 están en SC, y valdrá **false** si ninguno lo está:

```
{ 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
```

Versión 1. Corrección.

Sin embargo, esa versión **no es correcta**. El motivo es que no cumple **la propiedad de exclusión mutua**, pues ambos procesos pueden estar en la sección crítica. Esto puede ocurrir si ambos leen `p01sc` y ambos la ven a `false`. es decir, si ocurre la siguiente secuencia de eventos:

1. el proceso 0 accede al PE, ve `p01sc` con valor `false`,
2. el proceso 1 accede al PE, ve `p01sc` con valor `false`,
3. el proceso 0 pone `p01sc` a `true` y entra en la sección crítica,
4. el proceso 1 pone `p01sc` a `true` y entra en la sección crítica.

Versión 2. Pseudocódigo.

Para solucionar el problema se usará una única variable lógica (**turno0**), cuyo valor servirá para indicar cuál de los dos procesos tendrá prioridad para entrar SC la próxima vez que lleguen al PE. La variable valdrá **true** si la prioridad es para el proceso 0, y **false** si es para el proceso 1:

```
{ 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
```

Versión 2. Corrección.

Esta segunda versión no es tampoco correcta, el motivo es distinto. Se dan estas circunstancias:

- Se cumple la propiedad de exclusión mutua. Esto es fácil de verificar, ya que si un proceso está en SC ha logrado pasar el bucle del protocolo de entrada y por tanto la variable turno0 tiene un valor que forzosamente hace esperar al otro.
- No se cumple la propiedad de **progreso en la ejecución**. El problema está en que este esquema obliga a los procesos a acceder de forma alterna a la sección crítica. En caso de que un proceso quiera acceder dos veces seguidas sin que el otro intente acceder más, la segunda vez quedará esperando indefinidamente.

Este es un buen ejemplo de la necesidad de tener en cuenta cualquier secuencia posible de mezcla de pasos de procesamiento de los procesos a sincronizar.

Versión 3. Pseudocódigo.

Para solucionar el problema de la alternancia, ahora usamos dos variables lógicas (**p0sc**, **p1sc**) en lugar de solo una. Cada variable vale **true** si el correspondiente proceso está en la sección crítica:

```
{ 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
```

Versión 3. Corrección.

De nuevo, esta versión no es correcta:

- Sí se cumple la propiedad de progreso: los procesos no tienen que entrar de forma necesariamente alterna, al usar dos variables independientes.
- No se cumple, sin embargo, la **exclusión mutua**, por motivos parecidos a la primera versión, ya que se pueden producir secuencias de acciones como esta:
  - el proceso 0 accede al PE, ve `p1sc` con valor `false`,
  - el proceso 1 accede al PE, ve `p0sc` con valor `false`,
  - el proceso 0 pone `p0sc` a `true` y entra en SC,
  - el proceso 1 pone `p1sc` a `true` y entra en SC.

Versión 4. Pseudocódigo.

Para solucionar el problema anterior se puede cambiar el orden de las dos sentencias del PE. Ahora las variables lógicas **p0sc** y **p1sc** están a **true** cuando el correspondiente proceso está en SC, pero también cuando está intentando entrar (en el 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
```

Versión 4. Corrección.

De nuevo, esta versión no es correcta:

- Ahora es fácil demostrar que sí se cumple la E.M.
- También se permite el entrelazamiento con regiones no críticas, ya que si un proceso accede al PE cuando el otro no está en el PE ni en el SC, el primero logrará entrar a la SC.
- Sin embargo, no se cumple el **progreso en la ejecución**, ya que puede ocurrir **interbloqueo**. En este caso en concreto, eso puede ocurrir si se produce una secuencia de acciones como esta:
  - el proceso 0 accede al PE, pone `p0sc` a `true`,
  - el proceso 1 accede al PE, pone `p1sc` a `true`,
  - el proceso 0 ve `p1sc` a `true`, y entra en el bucle de espera,
  - el proceso 1 ve `p0sc` a `true`, y entra en el bucle de espera.

Versión 5. Pseudocódigo.

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
```

Versión 5. Corrección.

Ahora se cumple exclusión mutua pero no es posible afirmar que es imposible que se produzca interbloqueo en el PE: no se cumple la propiedad de **progreso**.

- La posibilidad de interbloqueo es pequeña, y depende de cómo se seleccionen las duraciones de los tiempos de la *espera de cortesía*, de cómo se implemente dicha espera, y de la metodología usada para asignar la CPU a los procesos o hebras a lo largo del tiempo.
- Por ejemplo, el interbloqueo podría ocurrir si ocurre que:
  - Hay una CPU y la espera de cortesía es espera ocupada.
  - Los dos procesos acceden al bucle de las líneas 4-8 (ambas variables están a verdadero).
  - Sistemáticamente, cuando un proceso está en la CPU y ha terminado de ejecutar la asignación de la línea 8, la CPU se le asigna al otro.

Subsección 4.4.
Algoritmo de Dekker.

Algoritmo de Dekker. Descripción.

El algoritmo de Dekker debe su nombre a su inventor, es un algoritmo correcto (es decir, cumple las propiedades mínimas establecidas), y se puede interpretar como el resultado final del refinamiento sucesivo de Dijkstra:

- Al igual que en la versión 5, cada proceso incorpora una *espera de cortesía* durante la cual le cede al otro la posibilidad de entrar en SC, cuando ambos coinciden en el PE.
- Para evitar interbloqueos, la espera de cortesía solo la realiza uno de los dos procesos, de forma alterna, mediante una variable de turno (parecido a la versión 2).
- La variable de turno permite también saber cuando acabar la espera de cortesía, que se implementa mediante un bucle (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 cortesia }
```

```
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
```

Subsección 4.5.
Algoritmo de Peterson.

Algoritmo de Peterson. Descripción.

Este algoritmo (que también debe su nombre a su inventor), es otro algoritmo correcto para EM, que además es más simple que el algoritmo de Dekker.

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

Pseudocódigo para 2 procesos.

El esquema del algoritmo queda como sigue:

```
{ 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 cortesia }
```

```
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
```

Demostración de exclusión mutua. (1/2)

Supongamos que en un instante de tiempo t ambos procesos están en SC, entonces:

(a) La última asignación (atómica) a la variable **`turno0`** (línea 5), previa a t, finalizó en un instante s (se cumple $s < t$).

(b) En el intervalo de tiempo $(s, t]$, ninguna variable compartida ha podido cambiar de valor, ya que en ese intervalo ambos procesos están en espera ocupada o en la sección crítica y no escriben esas variables.

(c) Durante el intervalo $(s, t]$, las variables **`p0sc`** y **`p1sc`** valen **`true`**, ya que cada proceso puso la suya a **`true`** antes de s, sin poder cambiarla durante $(s, t]$.

Demostración de exclusión mutua. (2/2)

de las premisas anteriores de deduce que

(d) si el proceso 0 ejecutó el último la línea 5 en el instante s, entonces no habría podido entrar en SC entre s y t (la condición de espera del proc.0 se cumpliría en el intervalo $(s,t]$), y por tanto en s forzosamente fue el proceso 1 el último que asignó valor a `turno0`, luego `turno0`b vale **`true`** durante el intervalo $(s,t]$.

(e) la condición anterior implica que el proceso 1 no estaba en SC en s, ni ha podido entrar a SC durante $(s,t]$ (ya que `p0sc and turno0` vale **`true`**), luego el proceso 1 no está en SC en t.

Vemos que se ha llegado a una contradicción con la hipótesis de partida, que por tanto debe ser falsa, luego no puede existir ningún instante en el cual los dos procesos estén en SC, es decir, se cumple la exclusión mutua.

Espera limitada

Supongamos que hay un proceso (p.ej. el 0) en espera ocupada en el PE, en un instante t, y veamos cuántas veces m puede entrar a SC el proceso 1 antes de que el 0 logre hacerlo:

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

En cualquiera de los dos casos, el proceso 1 no puede después llegar (o volver) a SC mientras el 0 continúa en el bucle, ya que para eso el proceso 1 debería pasar antes por la asignación de **`true`** a **`turno0`**, y eso provocaría que (después de un tiempo finito) forzosamente el proceso 0 entra en SC mientras el 1 continúa en su bucle.

Por tanto, la cota que requiere la propiedad es $n = 1$.

Progreso en la ejecución

Para asegurar el progreso es necesario asegurar

- Ausencia de interbloqueos en el PE:
  Esto es fácil de demostrar pues si suponemos que hay interbloqueo de los dos procesos, eso significa que son indefinida y simultáneamente verdaderas las dos condiciones de los bucles de espera, y eso implica que es verdad `turno0 and not turno0`, lo cual es absurdo.
- Independencia de procesos en RS:
  Si un proceso (p.ej. el 0) está en PE y el otro (el 1) está en RS, entonces `p1sc` vale `false` y el proceso 0 puede progresar a la SC independientemente del comportamiento del proceso 1 (que podría terminar o bloquearse estando en RS, sin impedir por ello el progreso del proc.0). El mismo razonamiento puede hacerse al revés.

Luego es evidente que el algoritmo cumple la condición de progreso.

Sección 5.
Soluciones hardware con espera ocupada (cerrojos) para E.M..

Subsección 5.1. Introducción.

Introducción

Los cerrojos constituyen una solución hardware basada en espera ocupada que puede usarse en procesos concurrentes con memoria compartida para solucionar el problema de la exclusión mutua.

- La espera ocupada constituye un bucle que se ejecuta hasta que ningún otro proceso esté ejecutando instrucciones de la sección crítica
- Existe un valor lógico en una posición de memoria compartida (llamado **cerrojo**) que indica si algún proceso está en la sección crítica o no.
- En el protocolo de salida se actualiza el cerrojo de forma que se refleje que la SC ha quedado libre

Veremos una solución elemental que sin embargo es incorrecta e ilustra la necesidad de instrucciones hardware específicas (u otras soluciones más elaboradas).

Una posible solución elemental

Vemos un esquema para procesos que ejecutan SC y RS repetidamente:

```
{ variables compartidas y valores iniciales }
var sc_ocupada : boolean := false ; { cerrojo: verdadero solo si SC ocupada }

{ 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
```

Errores de la solución simple

La solución anterior no es correcta, ya que no garantiza exclusión mutua al existir secuencias de mezclado de instrucciones que permiten a más de un proceso ejecutar la SC a la vez:

- La situación se da si n procesos acceden al protocolo de entrada y todos ellos leen el valor del cerrojo a **`false`** (ninguno lo escribe antes de que otro lo lea).
- Todos los procesos registran que la SC está libre, y todos acceden a ella.
- El problema es parecido al que ya vimos de acceso simultáneo a una variable en memoria compartida: la lectura y posterior escritura del cerrojo se hace en varias sentencias distintas que se entremezclan.

una solución es usar instrucciones máquina atómicas (indivisibles) para acceso a la zona de memoria donde se aloja el cerrojo. Veremos una de ellas: **`TestAndSet`**.

Subsección 5.2.
La instrucción `TestAndSet`.

La instrucción `TestAndSet`.

Es una instrucción máquina disponible en el repertorio de algunos procesadores.

- Admite como argumento la dirección de memoria de la variable lógica que actúa como cerrojo.
- Se invoca como una función desde LLPP de alto nivel, y ejecuta estas acciones:
  1. lee el valor anterior del cerrojo
  2. pone el cerrojo a **`true`**
  3. devuelve el valor anterior del cerrojo
- Durante su ejecución, ninguna otra intrucción ejecutada por otro proceso puede leer ni escribir la variable lógica: por tanto, se ejecuta de forma atómica.

Protocolos de entrada y salida con TestAndSet.

La forma adecuada de usar **TestAndSet** es la que se indica en este esquema:

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

cuando hay más de un proceso intentando entrar en SC (estando SC libre), solo uno de ellos (el primero en ejecutar **TestAndSet**) ve el cerrojo (**sc_ocupada**) a **false**, lo pone a **true** y logra entrar a SC.

Subsección 5.3.
Desventajas de los cerrojos..

Desventajas de los cerrojos.

Los cerrojos constituyen una solucion válida para EM que consume poca memoria y es eficiente en tiempo (excluyendo las esperas ocupadas), sin embargo:

- las 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.
- en la forma básica que hemos visto no se cumplen ciertas condiciones de equidad

Subsección 5.4.
Uso de los cerrojos..

Uso de los cerrojos.

Las desventajas indicadas hacen que el uso de cerrojos sea restringido, en el sentido que:

- por seguridad, normalmente solo se usan desde componentes software que forman parte del sistema operativo, librerías de hebras, de tiempo real o similares (estas componentes suelen estar bien comprobadas y por tanto libres de errores o código malicioso).
- para evitar la pérdida de eficiencia que supone la espera ocupada, se usan solo en casos en los que la ejecución de la SC conlleva un intervalo de tiempo muy corto (por tanto las esperas ocupadas son muy cortas, y la CPU no se desaprovecha).

Bibliografía del tema 2.

Para más información más detallada, ejercicios y bibliografía adicional, se puede consultar:

2.1. **Introducción**
Palma (2003) capítulo 3.

2.2. **Monitores como mecanismo de alto nivel.**
Palma (2003) capítulo 6, Andrews (2000) capítulo 5, Ben Ari (2006) capítulo 7

2.3. **Soluciones software con Espera Ocupada para E.M.**
Palma (2003) capítulo 3, Ben Ari (2006) capítulo 3

2.4. **Soluciones hardware con Espera Ocupada para E.M.**
Palma (2003) capítulo 3, Andrews (2000) capítulo 3

2.5. **Semáforos para sincronización.**
Palma (2003) capítulo 4, Andrews (2000) capítulo 4, Ben Ari (2006) capítulo 6

Fin de la presentación.