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

Dpt. Lenguajes y Sistemas Informáticos
ETSI Informática y de Telecomunicación
Universidad de Granada

Curso 13-14

Índice

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

Indice de la sección

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.

Entrada y salida en secciones 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 secciones 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.

de esta forma se prevee el caso más general en el cual 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 secciones críticas sea el menor posible, y que las instrucciones ejecutadas no puedan fallar.

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.

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.

Progreso

Consideremos una sección crítica en un instante en el cual no hay ningún proceso ejecutándola, pero sí hay uno o varios procesos en el PE compitiendo por entrar a la SC. La propiedad de progreso establece que

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 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.

Indice de la sección

Sección 2
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 algoritmos:

- Algoritmo de **Dekker** (para 2 procesos)
- Algoritmo de **Peterson** (para 2 y para un número arbitrario de procesos).

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 Proceso 0 y Proceso 1, cada uno de ellos ejecuta un bucle infinito conteniendo:
  1. Protocolo de entrada (PE).
  2. Sección crítica (SC).
  3. Protocolo de salida (PS).
  4. Otras sentencias del proceso (RS).

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 ;
      { seccion critica }
      p01sc := false ;
      { resto seccion }
   end
end
```

```
process P1 ;
begin
   while true do begin
      while p01sc do begin end
      p01sc := true ;
      { seccion critica }
      p01sc := false ;
      { resto seccion }
   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 protocolo de entrada, ve **p01sc** con valor **`false`**,
2. el proceso 1 accede al protocolo de entrada, 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 cual de los dos procesos tendrá prioridad para entrar SC la próxima vez que llegen 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 ; { podria ser tambien 'false' }
```

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

```
process P1 ;
begin
   while true do begin
      while turno0 do begin end
      { seccion critica }
      turno0 := true ;
      { resto seccion }
   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 ;
      { seccion critica }
      p0sc := false ;
      { resto seccion }
   end
end
```

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

Versión 3. Corrección.

De nuevo, esta versión no es correcta:

- Ahora sí se cumple la propiedad de progreso en ejecución, ya que los procesos no tienen que entrar de forma necesariamente alterna, al usar dos variables independientes. Si un proceso quiere entrar, podrá hacerlo si el otro no está en SC.
- 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 protocolo de entrada, ve **p1sc** con valor **falso**,
  - el proceso 1 accede al protocolo de entrada, ve **p0sc** con valor **falso**,
  - el proceso 0 pone **p0sc** a **verdadero** y entra en la sección crítica,
  - el proceso 1 pone **p1sc** a **verdadero** y entra en la sección crítica.

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
      { seccion critica }
      p0sc := false ;
      { resto seccion }
   end
end
```

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

Versión 4. Corrección.

De nuevo, esta versión no es correcta:

- Ahora sí se cumple la exclusión mutua. Es fácil ver que si un proceso está en SC, el otro no puede estarlo.
- 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 protocolo de entrada, pone **p0sc** a **verdadero**,
  - el proceso 1 accede al protocolo de entrada, pone **p1sc** a **verdadero**,
  - el proceso 0 ve **p1sc** a **verdadero**, y entra en el bucle de espera,
  - el proceso 1 ve **p0sc** a **verdadero**, 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       { seccion critica }
11       p0sc := false ;
12       { resto seccion }
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       { seccion critica }
11       p1sc := false ;
12       { resto seccion }
13    end
14 end
```

Versión 5. Corrección.

En este caso, se cumple exclusión mutua pero, sin embargo, no es posible afirmar que es imposible que se produzca interbloqueo en el PE. Por tanto, 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.
- En particular, y a modo de ejemplo, el interbloqueo podría ocurrir si ocurre que:
  - Ambos procesos comparten una CPU y la espera de cortesía se implementa como una 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 7, la CPU se le asigna al otro.

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       { seccion critica }
14       turno0 := false ;
15       p0sc := false ;
16       { resto seccion }
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       { seccion critica }
14       turno0 := true ;
15       p1sc := false ;
16       { resto seccion }
17    end
18 end
```

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 }
```

```
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        { seccion critica }
9        p0sc := false ;
10       { resto seccion }
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        { seccion critica }
9        p1sc := false ;
10       { resto seccion }
11    end
12 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 **verdadero**, ya que cada proceso puso la suya a **verdadero** 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** vale **verdadero** 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 y turno0** vale **verdadero**), 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 mútua.

Espera limitada

Supongamos que hay un proceso (p.ej. el 0) en espera ocupada en el PE, en un instante t, y veamos cuantas 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 while el 0 continúa en el bucle, ya que para eso el proceso 1 debería pasar antes por la asignación de **verdadero** a **turno0**, y eso provocaría que (después de un tiempo finito) forzosamente el proceso 0 entra en SC while el 1 continua 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 y no 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 **falso** 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.

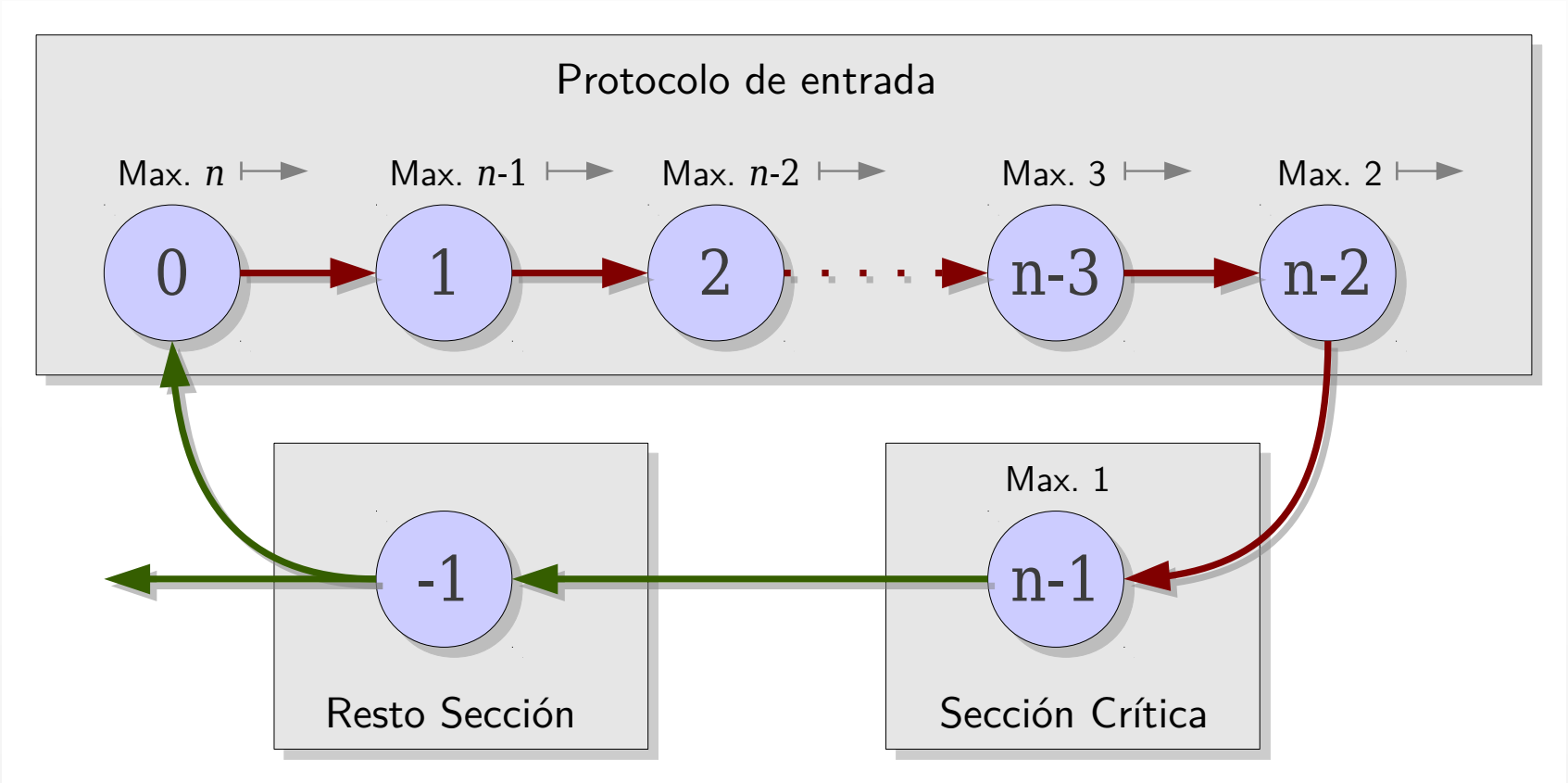
Algoritmo de Peterson para n procesos.

Peterson introdujo, junto con la versión vista, una generalización de su algoritmo para EM con n procesos.

- Cada proceso i (con $0 \leq i < n$) puede estar en uno de $n+1$ estados posibles, numerados de -1 a $n-1$:
  - El estado -1 indica que el proceso está en RS.
  - Los estados 0 al $n-2$ indican que el proceso está en PE.
  - El estado $n-1$ indica que el proceso está en SC.
- Al ingresar en PE, el proceso pasa al estado 0. Durante el PE cada proceso pasa por todos los estados $0 \leq j < n$ en orden, y en cada estado j (con $j < n-1$) hace espera ocupada antes de pasar al $j+1$.
- Las esperas están diseñadas de forma que en cada estado j con $j = 0, \ldots, n-1$, puede haber $n-j$ procesos como mucho. Esto asegura la EM pues en el estado $n-1$ (es decir, en SC) solo habrá un proceso como mucho.

Diagrama de estados.

Las transiciones en rojo suponen espera ocupada.



Variables compartidas

Las variables compartidas son dos vectores de enteros (de nombres **estado** y **ultimo**), que son escritos por cada proceso justo cuando accede a un nuevo estado:

estado: la entrada i (con $0 \leq i < n$) contiene el estado del i-ésimo proceso, es un valor entero entre -1 y $n-1$, ambos incluidos. Debe estar inicializado a -1.

ultimo: la entrada j (con $0 \leq j < n$) contiene el índice del último proceso en ingresar en el estado j (es un valor entero entre 0 y $n-1$). Su valor inicial es indiferente (ya que los procesos siempre escriben en una entrada antes de leerla).

Condición de espera en cada estado

Inicialmente, todos los procesos ingresan al estado 0 al inicio del PE. El proceso i puede pasar del estado j al estado $j+1$ tras hacer una espera ocupada, hasta que se dé una cualquiera de estas dos condiciones:

(a) No hay otros procesos en el estado j ni en estados posteriores a j (incluyendo SC). En este caso el proceso i es el más avanzado de todos los procesos en PE o SC (decimos que i es el proceso **líder**). Esto equivale a:

$$\forall\ k \in \{0,\ldots,n-1\} \text{ t.q. } k \neq i : \text{estado}[k] < \text{estado}[i]$$

(b) Después de que el proceso i ingresase al estado j, otros procesos también lo han hecho. Esto equivale a:

$$i \neq \text{ultimo}[j]$$

Pseudocódigo.

```
{ Variables compartidas, valores iniciales y funciones }
var estado : array [0..n-1] of integer ; { estado de cada proc. (inic. -1) }
    ultimo : array [0..n-2] of integer ; { ultimo proc. en cada estado del PE }
function lider( i : integer) : boolean; { ⟺ ∀ k ≠ i : estado[k] < estado[i] }

{ Procesos }
process P[ mi_numero : 0 .. n-1 ] ;
begin
   while true do begin
      for mi_etapa := 0 to n-2 do begin
         estado[mi_numero] := mi_etapa ;
         ultimo[mi_etapa] := mi_numero ;
         while ultimo[mi_etapa]==mi_numero and not lider(mi_numero) do
            begin end
      end
      estado[mi_numero] := n-1 ;
      { seccion critica }
      estado[mi_numero] := -1 ;
      { resto seccion }
end
```

La función lógica lider.

La función **lider** lee la variable compartida **estado**, y devuelve verdadero solo si el i-ésimo proceso es el más avanzado de todos los que están en PE y SC:

```
function lider( i : integer ) : boolean ;
begin
   for k := 0 to n-1 do
      if i ≠ k and estado[i] ≤ estado[k] then
         return false ;
   return true ;
end
```

Indice de la sección

Sección 3
Soluciones hardware con espera ocupada (cerrojos) para E.M.

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 dos de ellas: **LeerAsignar** (**TestAndSet**) e **Intercambia** (**Swap**).

La instrucción LeerAsignar.

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

- admite como argumento la dirección de memoria de la variable lógica que actua 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 **verdadero**
  3. devuelve el valor anterior del cerrojo
- al ser una única instrucción máquina su ejecución no puede entremezclarse con ninguna otra instrucción ejecutada en el mismo procesador
- bloquea el bus del sistema y la memoria donde está el cerrojo de forma que ninguna otra instrucción similar de otro procesador puede entremezclarse con ella.

Protocolos de entrada y salida con LeerAsignar

La forma adecuada de usar **LeerAsignar** 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 LeerAsignar( 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 **LeerAsignar**) ve el cerrojo (**sc_ocupada**) a **falso**, lo pone a **verdadero** y logra entrar a SC.

La instrucción Intercambia.

Constituye otra instrucción hardware que, al igual que **LeerAsignar**, se ejecuta de forma atómica en entornos con varios procesos ejecutándose en uno o varios procesadores compartiendo memoria.

- actua sobre un cerrojo (un valor lógico) en memoria compartida
- admite dos argumentos:
  - la dirección de memoria del cerrojo (variable global en memoria compartida)
  - la dirección de memoria de otra variable lógica (no necesariamente compartida, típicamente es una variable local del proceso que llama a **Intercambia**)
- el efecto de la instrucción es intercambiar los dos valores lógicos almacenados en las posiciones de memoria indicadas
- se invoca como un procedimiento

Protocolos de entrada y salida con Intercambia

La forma de usar **Intercambia** es como se indica aquí:

```
{ variables compartidas y valores iniciales }
var sc_libre : boolean := true ; { verdadero solo si la SC esta libre }

{ procesos }
process P[ i : 1 .. n ];
var { variable no compartida: true solo si este proceso ocupa la SC }
   sc_ocupada_proc : boolean := falso ;
begin
   while true do begin
      repeat
            Intercambia( sc_libre, sc_ocupada_proc  ) ;
      until sc_ocupada_proc ;
      { seccion critica }
      intercambia( sc_libre, sc_ocupada_proc ) ;
      { resto seccion }
   end
end
```

Validez de Intercambia para exclusión mutua

Se puede verificar que funciona:

- si hay n procesos conteniendo una sección crítica, habrá $n+1$ variables lógicas involucradas:
  - n variables locales (**sc_ocupada_proc**), una por proceso
  - el cerrojo (**sc_libre**)
- en cada instante de tiempo, exactamente una de esas $n+1$ variables lógicas vale **verdadero**, y el resto valen **falso**
- este invariante se cumple debido a que:
  - es inicialmente cierto (ver los valores iniciales)
  - las llamadas a **Intercambia** no lo hacen falso

por tanto se cumple la exclusión mutua ya que en cada instante solo el proceso cuya variable local este a **verdadero** (si es que hay alguno) está en SC.

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

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).

Indice de la sección

Sección 4
Semáforos para sincronización

Semáforos. Introducción

Los **semáforos** constituyen un mecanismo que soluciona o aminora los problemas indicados para los cerrojos, 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 que deseen acceder a una SC no ocupen la CPU mientras esperan a que otro proceso o hebra abandone dicha SC, 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), mediante una interfaz bien definida.

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

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)
  - Si el valor de s es mayor que cero, decrementar en una unidad dicho valor
  - Si el valor de s es cero, bloquear el proceso que invoca en el conjunto de procesos bloqueados asociado a s
- **sem_signal**(s)
  - Si el conjunto de procesos bloqueados asociado a s no está vacío, desbloquear uno de dichos procesos.
  - Si el conjunto de procesos bloqueados asociado a s está vacío, incrementar en una unidad el valor de s.

Invariante de un semáforo

Dado un semáforo s, en un instante de tiempo cualquiera t (en el cual el valor de s no esté en proceso de actualización) se cumplirá que:

$$v_0 + n_s = v_t + n_w$$

todos estos símbolos desginan números enteros no negativos, en concreto:

- v_0 = valor inicial de s
- v_t = valor de s en el instante t.
- n_s = número total de llamadas a **sem_signal**(s) completadas hasta el instante t.
- n_w = número total de llamadas a **sem_wait**(s) completadas hasta el instante t (no se incluyen los **sem_wait** no completados por haber dejado el proceso en espera).

Implementación de sem_wait y sem_signal.

Se pueden implementar con este esquema:

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

```
procedure sem_signal(var s:semaphore);
begin
   if vacio( s.procesos ) then
      s.valor := s.valor + 1 ;
   else
      desbloquear( s.procesos ) ;
end
```

En un semáforo cualquiera, estas operaciones se ejecutan de forma atómica, 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**).

Uso de semáforos para exclusión mutua

Los semáforos se pueden usar para EM usando un semáforo inicializado a 1, y haciendo **sem_wait** en el protocolo de entrada y **sem_signal** en el protocolo de salida:

```
{ variables compartidas y valores iniciales }
var sc_libre : semaphore := 1 ; { vale 1 si SC esta libre, 0 si SC esta ocupada }

{ procesos }
process P[ i : 0..n ];
begin
   while true do begin
      sem_wait( sc_libre ) ; { esperar bloqueado hasta que sc_libre sea 1 }
         { seccion critica: ..... }
      sem_signal( sc_libre ) ; { desbloquear proc. en espera o poner sc_libre a 1 }
      { resto seccion: ..... }
   end
end
```

En cualquier instante de tiempo, la suma del valor del semáforo más el número de procesos en la SC es la unidad. Por tanto, solo puede haber 0 o 1 procesos en SC, y se cumple la exclusión mutua.

Uso de semáforos para sincronización

El problema del Productor-Consumidor que vimos se puede resolver fácilmente con semáforos:

```
{ variables compartidas y valores iniciales }
var x                : integer ;          { contiene cada valor producido }
    producidos       : integer :=0 ;      { numero de valores producidos }
    consumidos       : integer :=0 ;      { numero de valores consumidos }
    puede_leer       : semaphore := 0 ;   { 1 si se puede leer 'x', 0 en otro caso }
    puede_escribir   : semaphore := 1 ;   { 1 si se puede escribir 'x', 0 en otro caso }
```

```
process Productor ; { calcula 'x' }
var a : integer ;
begin
   while true do begin
      a := ProducirValor() ;
      sem_wait( puede_escribir );
      x := a ; { sentencia E }
      producidos := producidos+1 ;
      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 }
      consumidos := consumidos+1 ;
      sem_signal( puede_escribir ) ;
      UsarValor(b) ;
   end
end
```

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).

Indice de la sección

Sección 5
Monitores como mecanismo de alto nivel

Fundamento teórico de los monitores

- **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).
- 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

Definición de monitor

- *C.A.R. Hoare (1974)* **Monitor**: mecanismo de **alto nivel** que permite:
  - Definir **objetos abstractos compartidos**:
    - Colección de variables encapsuladas (datos) que representan un recurso compartido por varios procesos.
    - Conjunto de procedimientos para manipular el recurso: afectan a las variables encapsuladas.
  - Garantizar el **acceso exclusivo** a las variables e implementar **sincronización**.
- El **recurso compartido** se percibe 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.

Propiedades de un monitor

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 (solución no estructurada)

- Las variables están protegidas, evitando interferencias exteriores.
- **Acceso estructurado**: Un proceso cualquiera solo puede acceder a las variables del monitor usando los procedimientos exportados por el monitor. Toda la espera y señalización se programan dentro del monitor. Si el monitor es correcto, lo será cada instancia usada por los procesos.
- **Exclusión mutua garantizada**: evita errores.

Sintaxis de un monitor

Estructura de una declaración de un monitor, de nombre *nombre-monitor*, con N procedimientos (con nombres: *nom_proc_1*,*nom_proc_2*,...,*nom_proc_n*), de los cuales algunos son exportados (*nom_exp_1*,*nom_exp_2*,....):

```
monitor nombre_monitor ;

   var                                  { declaracion de variables permanentes (privadas)  }
      ..... ;                           {     (se conservan entre llamadas al monitor)  }
   export                               { declaracion de procedimientos publicos     }
     nom_exp_1, nom_exp_2 .... ;        {     (se indican solo los nombres)  }

   procedure nom_proc_1( ... );         { decl. de procedimiento y sus parametros formales }
      var ...    ;                      { declaracion de variables locales a nom_proc_1 }
   begin
      ...                               { codigo que implementa  nom_proc_1 }
   end
   ...                                  { resto de procedimientos del monitor }

begin
   ....                                 { codigo de inicializacion de vars. permanentes }
end
```

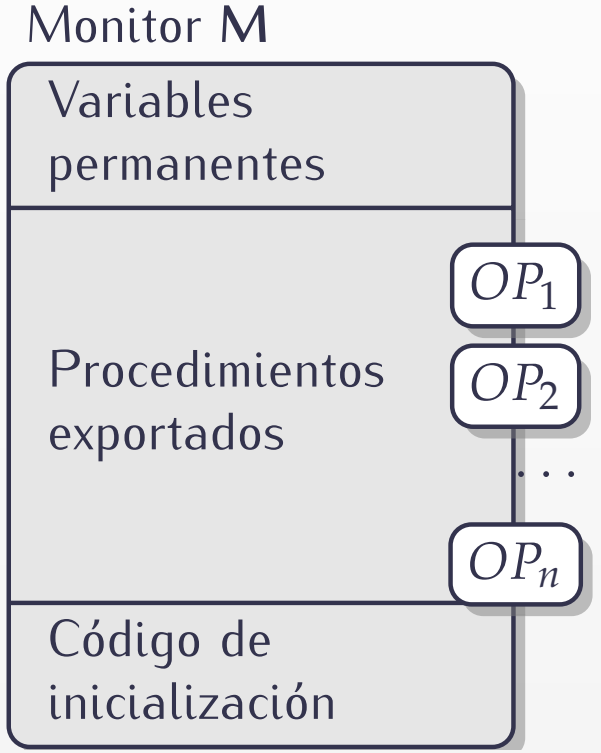
Componentes de un monitor

1. Un conjunto de **variables permanentes**:
   - Almacenan el estado interno del recurso compartido que está siendo representado por el 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.
2. Un conjunto de **procedimientos internos**:
   - Manipulan las variables permanentes.
   - 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.
3. Un **código de inicialización**:
   - Inicializa las variables permanentes del monitor.
   - Se ejecuta una única vez, antes de cualquier llamada a procedimiento 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 VariableCompartida ;

  var x : integer; { permanente }

  export incremento, valor;

  procedure incremento( );
  begin
       x := x+1 ; { incrementa valor actual }
  end;
  procedure valor(var v:integer);
  begin
       v := x ; { copia sobre v valor actual }
  end;

begin
  x := 0 ; { inicializa valor }
end
```

```
{ ejemplo de uso del monitor          }
{ (debe aparecer en el ambito de la }
{  declaracion del monitor)           }


VariableCompartida.incremento();
VariableCompartida.valor( k ) ;
```

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 reales apropiados.
  - Cuando un proceso está ejecutando un procedimiento exportado del monitor decimos que está dentro del monitor.
- **Exclusión mutua**: Si un proceso está ejecutando un procedimiento del monitor (está dentro del mismo), ningún otro proceso podrá entrar al monitor (a cualquier procedimiento del mismo) hasta que el proceso que está dentro del monitor termine la ejecución del procedimiento que ha invocado.
  - 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 ejecutar 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**: Para cada recurso compartido se tendría que definir un monitor. Para recursos con estructura y comportamiento idénticos podemos instanciar (de forma parametrizada) el monitor tantas veces como recursos se quieran controlar.
  - Declaración de un tipo monitor e instanciación:

```
class monitor nombre_clase_monitor( parametros_formales ) ;
     .... .......... { cuerpo del monitor semejante a los anteriores }
end
var nombre_instancia : nombre_clase_monitor( parametros_actuales ) ;
```

  - **Implementaciones reentrantes**: poder crear una nueva instancia independiente de un monitor para una tarea determinada permite escribir código reentrante que realiza dicha tarea.

Ejemplo de instanciación de un monitor

Podemos escribir un ejemplo similar a `VariableCompartida`, pero ahora parametrizado:

```
{ declaracion del monitor }

class monitor VarComp(pini,pinc : integer);

   var x, inc : integer;

   export incremento, valor;

   procedure incremento( );
   begin
      x := x+inc ;
   end;
   procedure valor(var v : integer);
   begin
      v := 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() ;
   mv1.valor(i1) ; { i1==1 }
   mv2.incremento() ;
   mv2.valor(i2) ; { i2==14 }
end
```

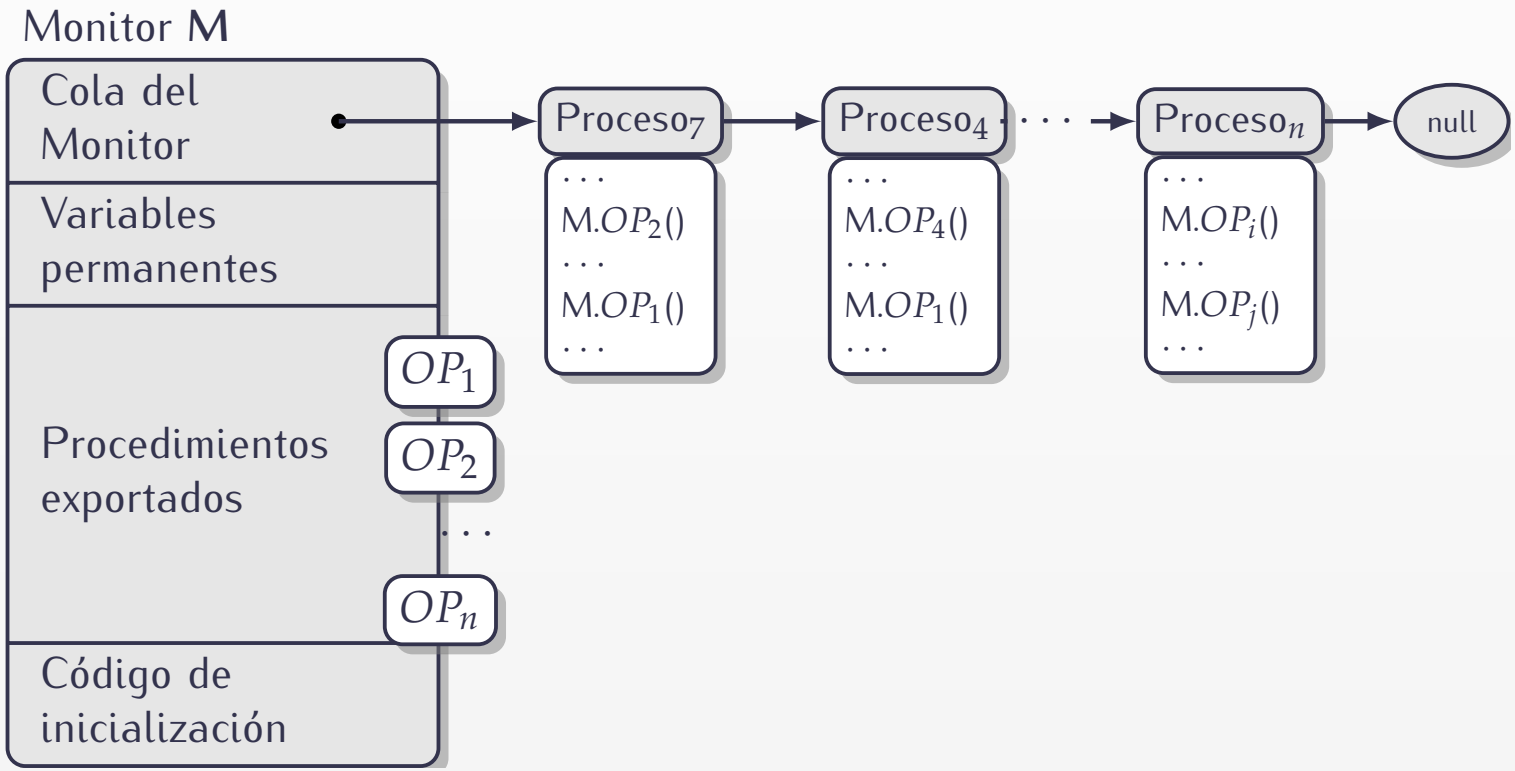
Cola del monitor para exclusión mutua

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

Cola del monitor

Esta cola forma parte del estado del monitor:



Sincronización en monitores

- Para implementar la sincronización, se requiere de una facilidad para bloqueo-activación de acuerdo a una condición.
- En **semáforos**, las instrucciones de sincronización presentan:
  - Bloqueo-activación
  - Cuenta, para representar la condición.
- En **monitores**:
  - Sólo se dispone de sentencias Bloqueo-activación.
  - La condición se representa mediante los valores de las variables permanentes del monitor.

Primitivas de bloqueo-activación en monitores

- **`wait`**: Estoy esperando a que alguna condición ocurra.

  Bloquea al proceso llamador.
- **`signal`**: Estoy señalando que una condición ocurre.

  Reactiva a un proceso esperando a dicha condición (si no hay ninguno no hace nada).

Ejemplo de monitor para planificar un único recurso

Este monitor permite gestionar el acceso a un recurso que debe usarse en E.M. por parte de varios procesos:

```
monitor recurso;

   var    ocupado    : boolean;
          noocupado  : condition;
   export adquirir, liberar ;
```

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

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

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

Variables condición o señales (1)

Para gestionar las condiciones de sincronización en monitores se utilizan las **señales** o **variables de condición**:

- Debe declararse una variable condición para cualquier condición lógica de sincronización que se pueda dar entre los procesos del programa (por cada razón de bloqueo basada en los valores de las variables permanentes del monitor).
- No tienen valor asociado (no requieren inicialización).
- Representan colas (inicialmente vacías) de procesos esperando la condición.
- Más de un proceso podrá estar dentro del monitor, aunque solo uno estará ejecutándose (el resto estarán bloqueados en variables condición).

Colas de condición o señales (2)

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

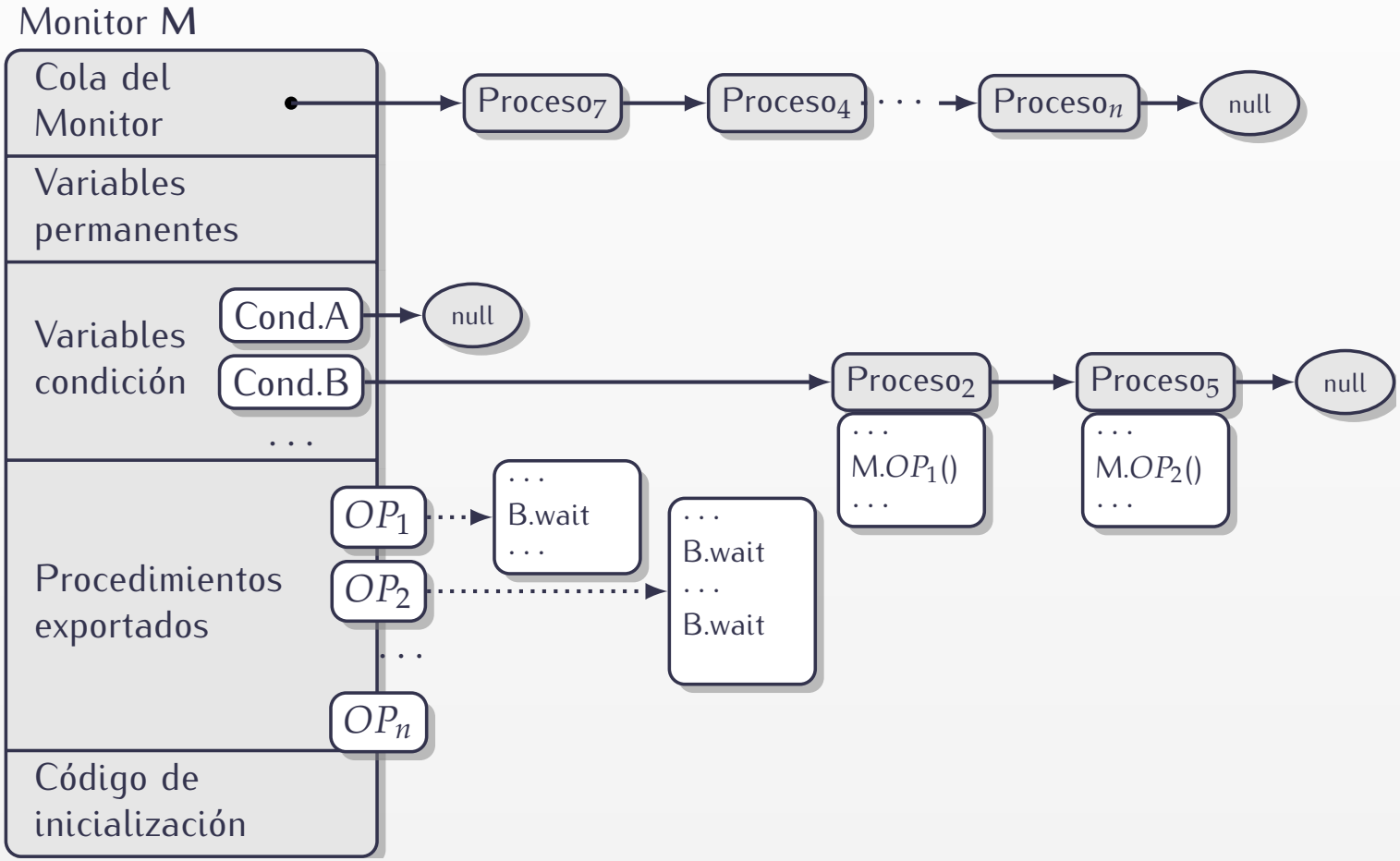
Operaciones sobre variables condición

- `cond.`**`wait`**`()`: Bloquea 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.
- `cond.`**`queue`**`()`: Función booleana que devuelve `verdadero` si hay algún proceso esperando en la cola de `cond` y `falso` en caso contrario.

- Cuando hay más de un proceso esperando en una condición `cond`:
  - **Propiedad FIFO**: `cond.`**`signal`**`()` reactivará al proceso que lleve más tiempo esperando.
  - **Evita la inanición**: Cada proceso en cola obtendrá eventualmente su turno.

Variables condición y colas asociadas.

Los procesos 2 y 5 ejecutan las operaciones 1 y 2, ambas producen esperas de la condición B.



Colas de condición con prioridad

- Por defecto, se usan colas de espera FIFO.
- A veces resulta útil disponer de un mayor control sobre la estrategia de planificación, dando la prioridad del proceso en espera como un nuevo parámetro:

cond.wait (prioridad:integer)

- Tras bloqueo, ordena los procesos en cola en base al valor de `prioridad`
- `cond.`**`signal`**`()` reanuda proceso que especificó el valor más bajo de `prioridad`.

- Precaución al diseñar el monitor: Evitar riesgos como la inanición.
- Ningún efecto sobre la lógica del programa: Funcionamiento similar con/sin colas de prioridad.
- Sólo mejoran las características dependientes del tiempo.

Ejemplo de cola con prioridad(1): Asignador

Asigna un recurso al siguiente trabajo más corto:

```
monitor asignador;

   var    libre : boolean ;
          turno : condition ;
   export peticion, liberar ;
```

```
procedure peticion(tiempo: integer);
begin
   if not libre then
      turno.wait( tiempo );
   libre := false ;
end
```

```
procedure liberar() ;
begin
   libre := true ;
   turno.signal();
end
```

```
{ inicializacion }
begin
   libre := true ;
end
```

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

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

```
monitor despertador;

   var    ahora      : integer ;
          despertar  : condition ;
   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
```

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

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

El problema de los lectores-escritores (LE)

En este ejemplo, hay dos tipos de procesos que acceden concurrentemente a una estructura de datos en memoria compartida:

escritores son 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 son procesos que leen a 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.

LE: vars. permanentes y procedimientos para lectores

```
monitor Lec_Esc ;

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

   export inicio_lectura, fin_lectura,
          inicio_escritura, fin_escritura ;
```

```
procedure inicio_lectura()
begin
   if escribiendo then { si hay escritor }
      lectura.wait() ; { se bloquea }
   n_lec := n_lec + 1 ;
   { desbloqueo en cadena de posibles }
   { procesos lectores bloqueados     }
   lectura.signal()
end
```

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

LE: procedimientos para escritores

```
procedure inicio_escritura()
begin
 {si hay procs. accediendo: esperar}
 if n_lec > 0 or escribiendo then
   escritura.wait()
 escribiendo:= true;
end;
```

```
procedure fin_escritura()
begin
 escribiendo:= false;
 if lectura.queue() then {si hay lect.:}
  lectura.signal();      {despertar uno }
 else                    {si no hay lect:}
  escritura.signal() ;   {desp. escr.}
end;
```

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

LE: uso del monitor

Los procesos lectores y escritores tendrían el siguiente aspecto:

```
monitor Lec_Esc ;
   ....
end
```

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

```
process Escritor[ i:1..m ] ;
begin
   while true do begin
      ..........
      Lec_Esc.inicio_escritura() ;
        { codigo 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).

Efectos de las operaciones sobre la E.M.

- Es necesario liberar la E.M. del monitor justo antes de ejecutar una operación **`cond.wait()`** para no generar un interbloqueo en la cola de procesos del monitor.
- Cuando se libera un proceso que está esperando por una condición **`cond`** (proceso señalado) es porque otro proceso (proceso señalador) ha ejecutado una operación **`cond.signal()`**:
  - El proceso señalado tiene preferencia para acceder al monitor frente a los que esperan en la cola del monitor (ya que éstos podrían cambiar el estado que ha permitido su liberación).
  - Si el proceso señalador continuase "dentro" del monitor, tendríamos una violación de la E.M. del monitor
  - El comportamiento concreto del proceso señalador dependerá de la semántica de señales que se haya establecido en la implementación del monitor.
- Algunos lenguajes implementan una operación que permite liberar a todos los procesos esperando por una condición (**`signal_all`**).

Señalar y continuar (SC)

- El proceso señalador continúa su ejecución dentro del monitor hasta que sale del mismo (porque termina la ejecución del procedimiento) o se bloquea en una condición. En ese momento, el proceso señalado se reactiva y continúa ejecutando código del monitor.
- Al continuar dentro del monitor, el proceso señalador podría cambiar el estado del monitor y hacer falsa la condición lógica por la que esperaba el proceso señalado.
- 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)

- Si hay procesos bloqueados por la condición **`cond`**, el proceso señalador sale del monitor después de ejecutar **`cond.signal()`**.
- 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)

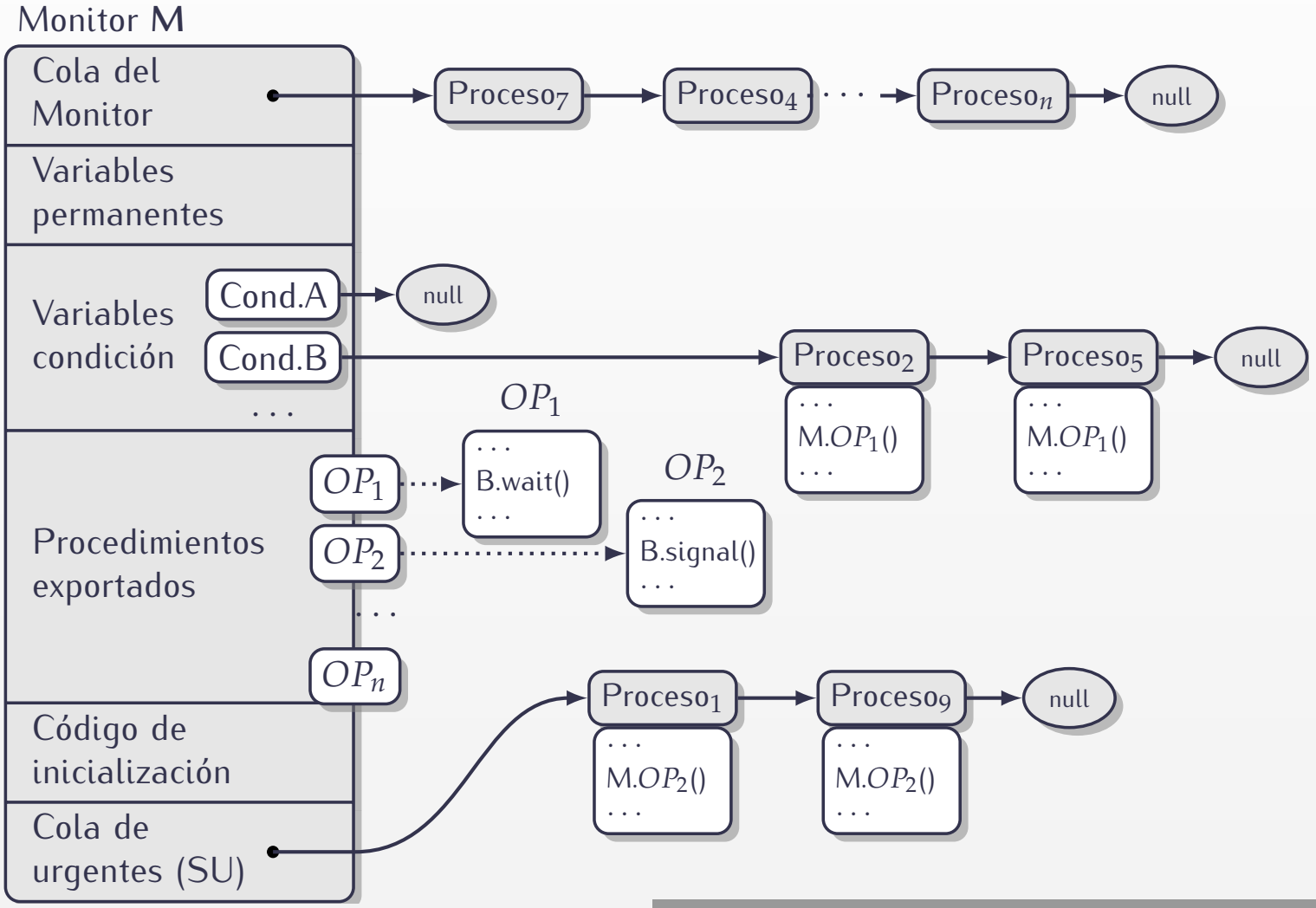
- El proceso señalador se bloquea justo después de ejecutar la operación **`signal`**.
- El proceso señalado entra de forma inmediata en el monitor.
- Está asegurado el estado que permite al proceso señalado continuar la ejecución del procedimiento del monitor en el que se bloqueó.
- El proceso señalador entra en 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)

- Esta solución intenta corregir el problema de falta de equitatividad de la solución anterior.
- 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 la operación signal se ejecuta al final de los procedimientos del monitor.
  El proceso señalador se bloquea dentro del monitor cuando ya no le quedan más instrucciones que ejecutar en el procedimiento, suponiendo un bloqueo innecesario que conlleva un innecesario doble cambio de contexto en CPU (reactivación señalado + salida).
- La semántica **SC** también es un poco ineficiente al obligar a usar un bucle while para cada instrucción **`signal`**.

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(...)
begin
   sem_wait(mutex);
   { cuerpo del procedimiento }
   sem_signal(mutex);
end
```

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

Para implementar la semántica **SU** necesitamos además un semáforo **next**, para implementar la cola de urgentes, y una variable entera **next_count** para contar los procesos bloqueados en esa cola:

```
procedure P1(...)
begin
   sem_wait(mutex);
   { cuerpo del procedimiento }
   if next_count > 0 then sem_signal(next);
                     else sem_signal(mutex);
end
```

```
{ inicializacion }
next := 0 ;
next_count :=0 ;
```

Implementación de monitores con semáforos

Para implementar las variables condición le asociamos un semáforo a cada una de ellas **x_sem** (inicializado a 0) y una variable para contar los procesos bloqueados en cada una de ellas (**x_sem_count** inicializada a 0):

Implementación de **x.wait()**

```
x_sem_count := x_sem_count + 1 ;
if next_count <> 0 then
   sem_signal(next) ;
else
   sem_signal(mutex);
sem_wait(x_sem);
x_sem_count := x_sem_count - 1 ;
```

Implementación de **x.signal()**

```
if x_sem_count <> 0 then begin
   next_count := next_count + 1 ;
   sem_signal(x_sem);
   sem_wait(next);
   next_count := next_count - 1 ;
end
```

Implementación de monitores con semáforos

Cada monitor tiene asociadas las siguientes colas de procesos:

- **Cola del monitor**: controlada por el semáforo **`mutex`**.
- **Cola de procesos urgentes**: controlada por el semáforo **`next`** y el número de procesos en cola se contabiliza en la variable **`next_count`**. Esta cola solo es necesaria para implementar la semántica SU.
- **Colas de procesos bloqueados en cada condición**: controladas por el semáforo asociado a cada condición (**`x_sem`**) y el número de procesos en cada cola se contabiliza en una variable asociada a cada condición (**`x_sem_count`**).

Esta implementación no permite llamadas recursivas a los procedimientos del monitor y no asegura la suposición FIFO sobre las colas de exclusión mutua y de las señales.
Los semáforos y monitores son equivalentes en potencia expresiva pero los monitores facilitan el desarrollo.

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 el orden en que se ejecutarán los procedimientos del monitor.
- El enfoque de verificación que vamos a seguir utiliza un **invariante de monitor**:
  - Establece una relación constante entre los valores permitidos de las variables del monitor.
  - Debe ser cierto cuando un procedimiento empieza a ejecutarse.
  - Debe ser cierto cuando un procedimiento termine de ejecutarse.
  - Debe ser cierto cuando un procedimiento llegue a una llamada a **`wait`**.

Invariante del monitor (IM)

- IM se ha de cumplir después de la inicialización de las variables permanentes.
- IM se ha de cumplir antes y después de cada acción.
- IM se ha de cumplir antes y después de cada operación **`wait`**. Una vez el proceso ha sido liberado (por una operación **`signal`** ejecutada por otro proceso), se debe cumplir la condición que permite liberarlo.
- IM se ha de cumplir antes y después de cada operación **`signal`** sobre una condición. También se debe cumplir antes la condición que permite liberar a un proceso.

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

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

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

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

Fin de la presentación del tema 2.