Tema2.2Sincronizacion-en-memoria...

beatrizmartin05

Sistemas Concurrentes y Distribuidos

2º Grado en Ingeniería Informática

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación
Universidad de Granada

5. MONITORES COMO MECANISMO DE ALTO NIVEL

5.1. Introducción. Definición de monitor

Al usar semáforos se generan algunos inconvenientes como operaciones dispersas y no protegidas, no puede haber un diseño modular, el uso y función de las variables no se hace explícito en el programa…
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
El **monitor** es un mecanismo de alto nivel que permite definir objetos abstractos compartidos, incluye:

- Una colección de variables encapsuladas(recurso compartido).
- Un conjunto de procedimientos (usados para manipular el recurso).

El conjunto de estos elementos garantiza el acceso en exclusión mutua a las variables encapsuladas y la sincronización requerida mediante esperas bloqueadas.

Propiedades del monitor:

1. Acceso estructurado y encapsulación
    - El proceso sólo se puede acceder al recurso mediante un conjunto de operaciones.
    - El proceso ignora la/s variable/s que representan al recurso y la implementación de las operaciones asociadas
2. Exclusión mutua en el acceso a los procedimientos
    - La exclusión mutua del monitor está garantizada por definición.
    - Garantiza que nunca dos procesos estarán ejecutando simultáneamente algún procedimiento del monitor.

Esto hace que sean preferibles respecto de los semáforos,dado que el uso de los monitores facilita el diseño e implementación de programas libres de errores.

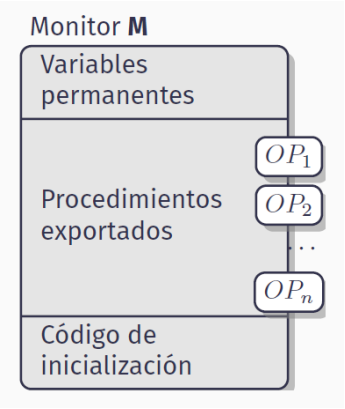
- Las variables están protegidas: sólo pueden leerse o modificarse desde el código del monitor.
- La exclusión mutua está garantizada: el programador no tiene que usar mecanismos explícitos de exclusión mutua.
- Las operaciones de esperas bloqueadas y de señalización se programan exclusivamente dentro del monitor: es más fácil verificar que el diseño es correcto.

Sintaxis de un monitor(pseudo-código):

```
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 y permanecen sin modificaciones entre dos llamadas consecutivas a procedimientos del monitor).
- **Procedimientos**: modifican el estado interno (constituyen la interfaz externa del monitor y pueden tener variables y parámetros locales, que toman un nuevo valor en cada activación del procedimiento).
- **Código de inicialización**: fija estado interno inicial (opcional, se ejecuta una única vez, antes de cualquier llamada a procedimientos del monitor).



- El uso que se hace del monitor se hace exclusivamente usando los procedimientos exportados.

- 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.

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.

5.2. 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.

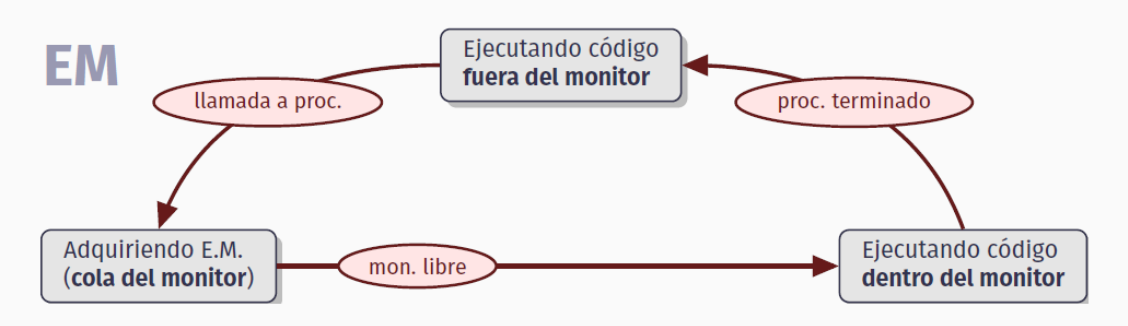
- **Los monitores son objetos pasivos**: después de ejecutarse el código de inicialización, un monitor es un objeto pasivo y el código de sus procedimientos sólo se ejecuta cuando estos son invocados por los procesos.

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:
    1. Si hay procesos en la cola, se desbloquea uno de ellos.
    2. Si no hay procesos en la cola, el monitor queda libre.
- Para garantizar la vivacidad del sistema, la planificación de la cola del monitor debe seguir una política FIFO.

Diagrama de estados de un proceso



5.3. Sincronización en monitores

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

En monitores, sin embargo

1. Sólo se dispone de sentencias de bloqueo y activación.
2. 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 lógica distinta que pueda hacer que un proceso tenga que esperar dentro de un monitor, se debe declarar una variable permanente de tipo **condition**(NO tiene valor asociado).

Esas variables, que actúan como colas de procesos bloqueados:

- 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 y signal.

Operaciones sobre variables de condición

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

- **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.
- **cond.queue()**: Función lógica que devuelve true si hay algún proceso esperando en la cola de cond, y false en caso contrario.

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.
Cuando un proceso es reactivado después 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.

5.4. Verificación de 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**. Soy una rata.

El Invariante del Monitor (IM)

Es una función lógica que se puede evaluar como true o false en cada estado del monitor a lo largo de la ejecución.
Su valor depende de la traza(secuencia ,ordenada en el tiempo, de llamadas a procedimientos del monitor ya completada) del monitor y de los valores de las variables permanentes de dicho monitor.
El IM debe ser cierto:

1. En el estado inicial, justo después de la inicialización de las variables permanentes.
2. Antes y después de cada llamada a un procedimiento del monitor
3. antes y después de cada operación wait.
4. antes y después de cada operación signal(debe ser cierta la condición lógica asociada a dicha variable).

5.5. Patrones de solución con monitores

Para ilustrar el uso de monitores, veremos los patrones de solución para los mismos tres problemas típicos:

1. **Espera única**: un proceso, antes de ejecutar una sentencia, debe esperar a que otro proceso complete otra sentencia.
2. **Exclusión mutua**: acceso en exclusión mutua a una sección crítica por parte de un número arbitrario de procesos.
3. **Problema del Productor/Consumidor**: similar a la espera única, pero de forma repetida en un bucle.

Monitor para espera única

```
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
```

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.

1/6

Este número es indicativo del riesgo del producto, siendo 1/6 indicativo de menor riesgo y 6/6 de mayor riesgo.

ING BANK NV se encuentra adherido al Sistema de Garantía de Depósitos Holandés con una garantía de hasta 100.000 euros por depositante. Consulta más información en ing.es

Que te den **10 € para gastar** es una fantasía. ING lo hace realidad.

Abre la **Cuenta NoCuenta** con el código **WUOLAH10**, haz tu primer pago y llévate 10 €.

Quiero el cash

Consulta condiciones **aquí**

do your thing

Patrón para 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
```

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
```

Sincronización tipo Productor/Consumidor

```
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
   b := PC.leer(); {copia valor en b}
   UsarValor(b) ;
 end
end
```

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

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 ;
   var v : integer ;
begin
  if not pendiente then
    cola_cons.wait();
  v := valor_com ; {copiar valor}
  pendiente := false ;
  cola_prod.signal();
  return v ; {devolver valor copiado}
end
```

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

5.6. El problema de los Lectores/Escritores

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.

Por simplicidad, suponemos que los procesos lectores y escritores ejecutan bucles en los cuales acceden repetidas veces a la estructura de datos compartida.

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

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

Para implementar la sincronización será necesario llevar la cuenta de cuántos lectores y cuántos escritores hay accediendo a la estructura de datos.

Llamamos a los procedimientos:

- Para **lectores**: ini_lectura y fin_lectura.
- Para **escritores**: ini_escritura y fin_escritura.

así que el monitor debe usarse de esta forma:

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

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

Por todo lo dicho, el monitor se implementa como vemos aquí:

```
monitor LE ;

var n_lec     : integer := 0;   { numero de lectores leyendo }
    n_esc     : integer := 0;   { numero escritores escribiendo }
    lectura   : condition; { no hay escrit. escribiendo, lectura posible }
    escritura : condition; { no hay lect. ni escrit., escritura posible }
```

```
procedure ini_lectura()
begin
  if n_esc > 0 then {si hay escrit.}
     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
```

```
procedure ini_escritura()
begin
  { si hay otros, esperar }
  if n_lec > 0 or n_esc > 0 then
    escritura.wait()

  { registrar que hay un escritor }
  n_esc := n_esc + 1 ;
end;
```

```
procedure fin_escritura()
begin
  { registrar que hay 0 escritores}
  n_esc := n_esc-1 ;

  { si hay lect., despertar uno }
  { si no hay, despertar un escritor}
  if lectura.queue() then
    lectura.signal();
  else
    escritura.signal() ;
end;
```

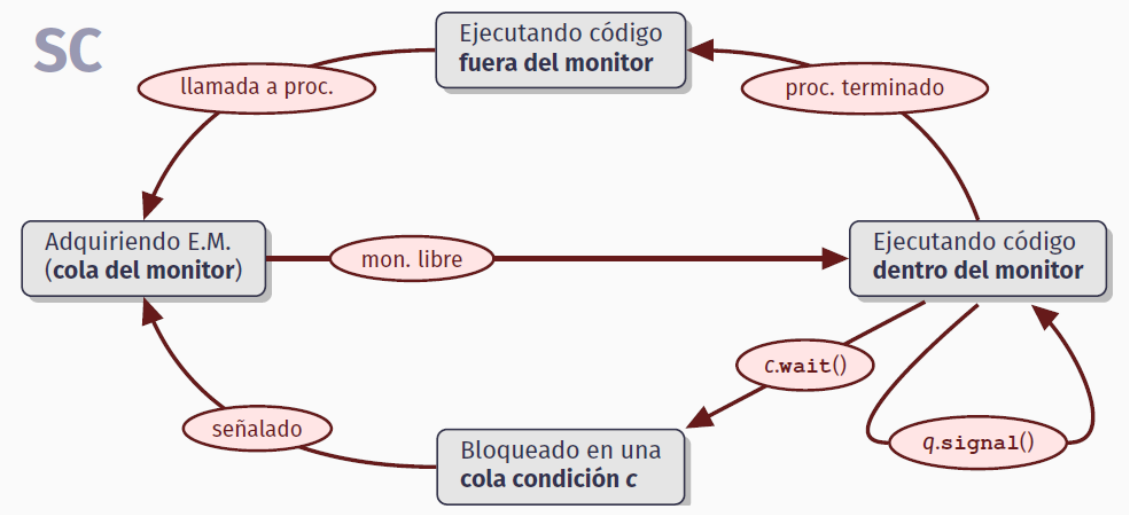
5.7. Semántica de las señales de los monitores

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) continúen la ejecución de su código restante, ya que no se cumpliría la exclusión mutua del monitor.
- 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

Señalar y continuar (**SC**)

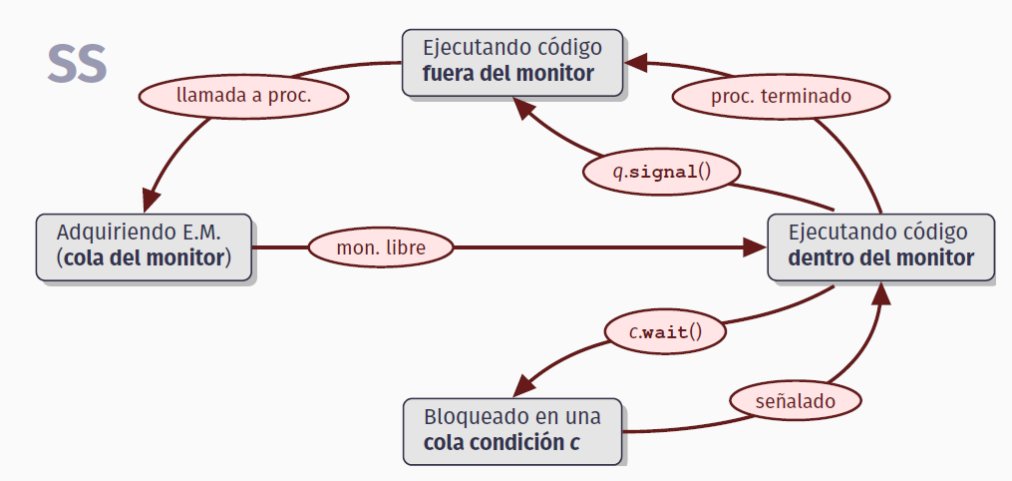


Señalador: continúa inmediatamente la ejecución de código del monitor tras signal.
Señalado: abandona la cola condición y espera en la cola del monitor hasta readquirir la E.M. y ejecutar código tras wait.

- Tanto el señalador como otros procesos pueden hacer falsa la condición después 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() ;
```

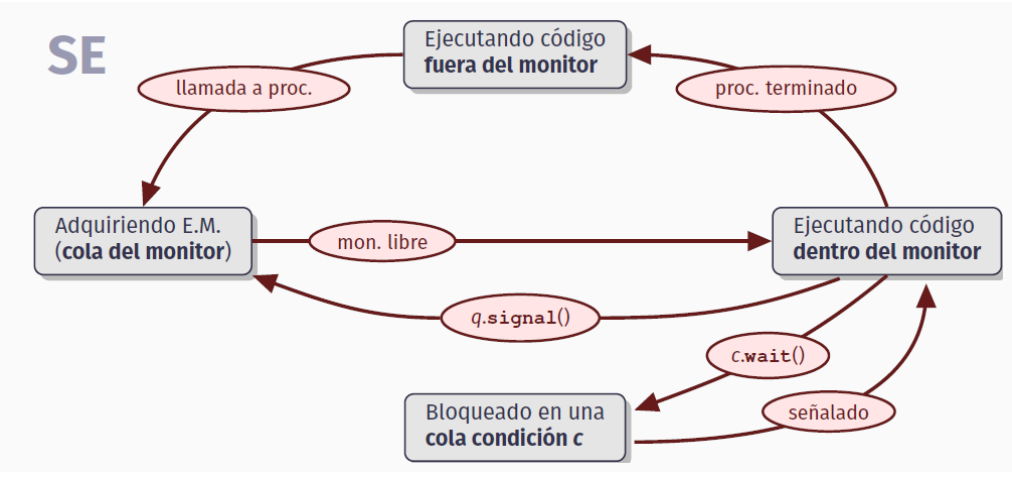
Señalar y salir (**SS**)



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.

- 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.

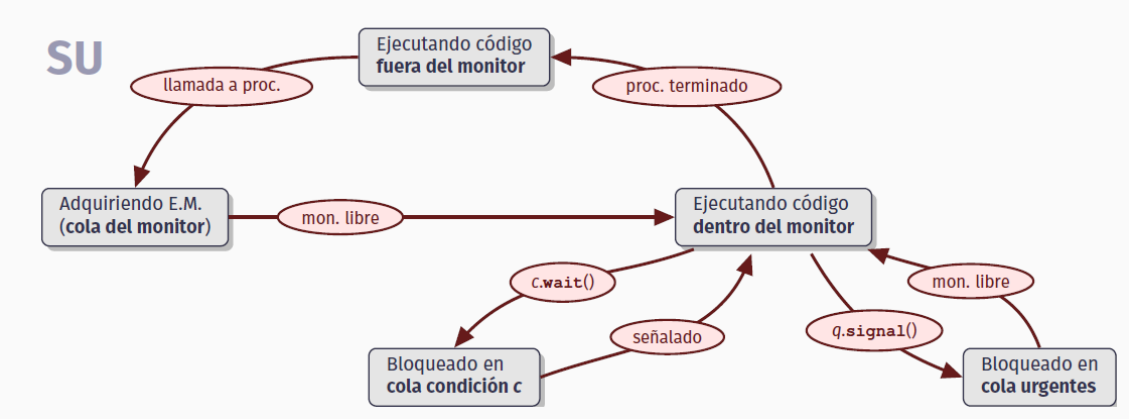
Señalar y esperar (**SE**)



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.

- 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**)



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.

- Es similar a la semántica SE, pero se intenta corregir el problema de falta de equitatividad.
- 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.

Análisis comparativo de las diferentes semánticas

1. **Potencia expresiva**: todas las semánticas son capaces de resolver los mismos problemas.
2. **Facilidad de uso**: la semántica SS condiciona el estilo de programación y puede llevar a aumentar de forma artificial el número de procedimientos
3. **Eficiencia**: las semánticas SE y SU resultan ineficientes cuando no hay código tras signal. La semántica SC también es un poco ineficiente al obligar a usar un bucle para cada instrucción wait.

5.8. Colas de 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.

Ejemplo:
En este monitor 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 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
```

5.9. Implementación de monitores

Implementación con semáforos

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

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

La exclusión mutua en el acceso a los procesos 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 ;
```

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
```