Tema 2

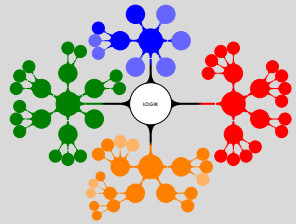
Sincronización en memoria compartida

SCD para GIIM

Asignatura *Sistemas Concurrentes y Distribuidos*

Fecha 11 Octubre, 2024

Departamento de Lenguajes y Sistemas Informáticos
Universidad de Granada



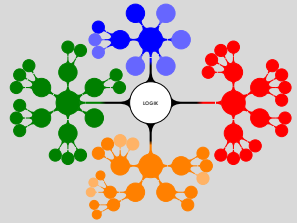
Limitaciones de los semáforos

Hay algunos inconvenientes de usar mecanismos como los semáforos:

- Basados en variables globales
- El uso y función de las variables (*protegidas* de los semáforos)
- Las operaciones se encuentran dispersas y no protegidas

Por tanto, es necesario:

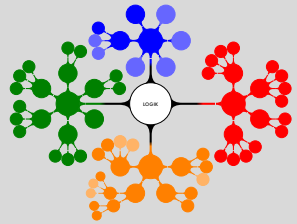
- un nuevo *mecanismo* de programación que permita la encapsulación de la información y de la sincronización entre procesos,
- programar las operaciones de sincronización (`wait`, `signal` `post`...) dentro de bloques o procedimientos que se ejecuten con instrucciones atómicas



Estructura y funcionalidad de un monitor

C.A.R. Hoare, en 1974, idea el concepto de Monitor, que es un mecanismo de programación de alto nivel que permite definir objetos abstractos compartidos entre los procesos concurrentes.

Las variables permanentes y los procedimientos de los monitores garantizan acceso en exclusión mutua y encapsulación de su sincronización.

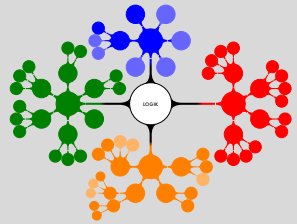


Características generales de los monitores

Concepto de monitor

"Módulo" de las aplicaciones concurrentes

- Modularidad en el desarrollo de programas y aplicaciones
- Programa= {Monitores, Procesos}
- Estructuración en el acceso a tipos de datos, variables compartidas, etc.
- Capacidad de modelado de interacciones cooperativas y competitivas entre procesos concurrentes, lo más general posible
- Ocultación a los procesos de las operaciones de sincronización sobre datos compartidos
- Reusabilidad basada en parametrización de los módulos monitor
- Verificación mediante reglas más simples que las de los semáforos



Semántica de los componentes de un monitor

Exclusión mutua en el acceso a los procedimientos/métodos del módulo monitor

Variables permanentes: son el estado interno del monitor

Procedimientos: modifican el estado interno (garantizando la exclusión mutua durante dicho cambio)

Código de inicialización: fija el estado interno inicial

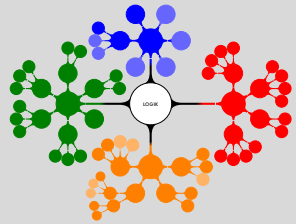


Diagrama de los componentes del monitor

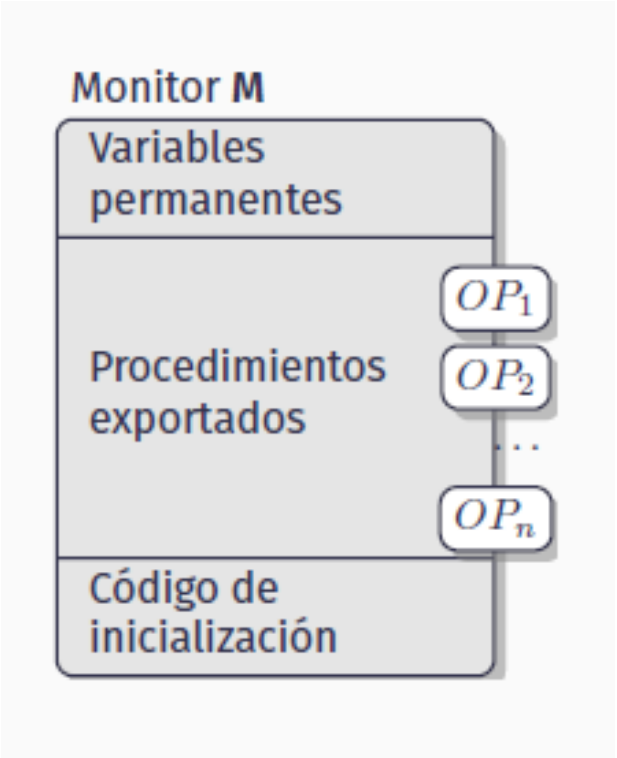
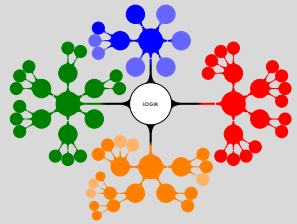


Figure: Un monitor se puede visualizar como aparece en el diagrama



Concepto de *monitor* III

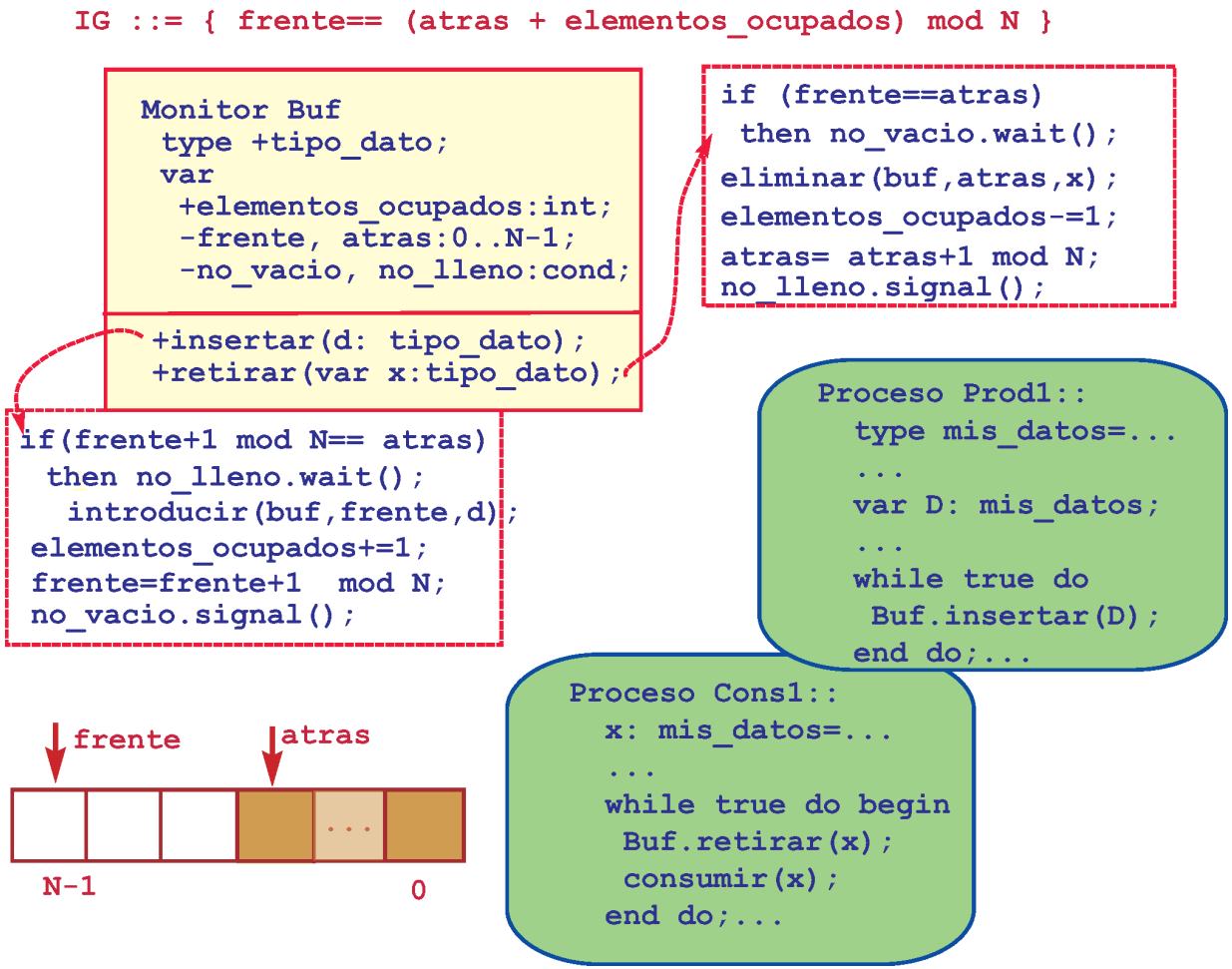
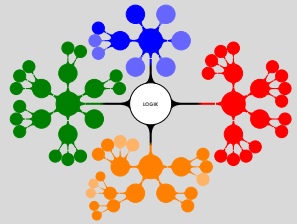
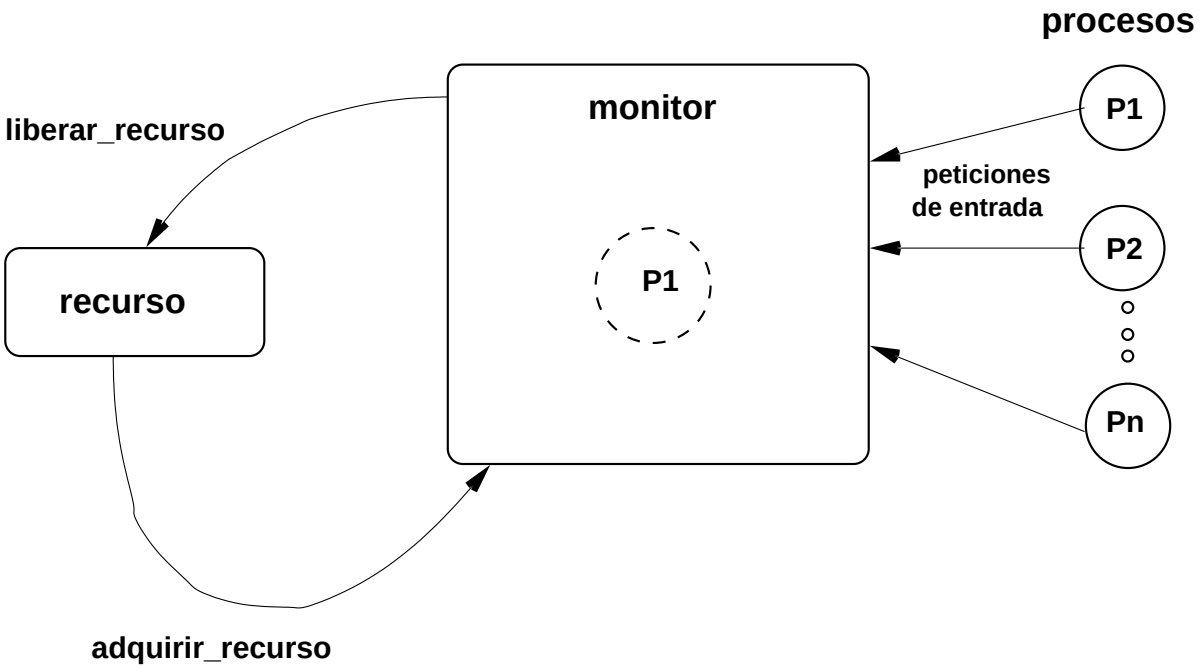


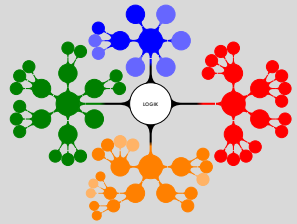
Figure: Representación de gráfica de un módulo monitor



Características de la programación con *monitores*

- Centralización de recursos críticos
- Sólo 1 procedimiento ejecutado por un solo proceso
- Los procedimientos pueden interrumpirse
- Posibilidad de ejecución concurrente de monitores no-relacionados

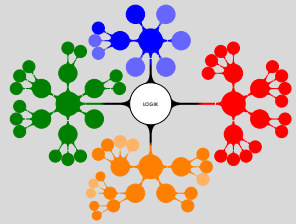




Instanciación de los monitores

Los monitores son “objetos” pasivos

Objetivos de la instanciación de los monitores

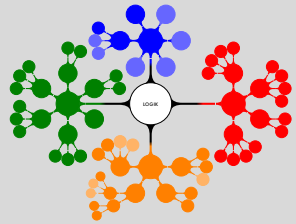


Instanciación de los monitores-II

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

Condición para que un lenguaje de programación pueda compilar monitores instanciables:

El código de los procedimientos de los monitores ha de ser *reentrante*

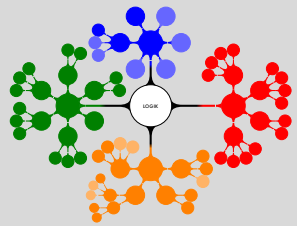


Instanciación de los monitores-III

```
class monitor VariableProtegida(entr,salid : integer);
var x, inc : integer;
// incremento(), valor(); son los procedimientos llamables
procedure incremento( );
begin
  x := x+inc ;
end;
procedure valor(var v : integer);
begin
  v := x ;
end;
begin
x:= entr ; inc := salid ;
end
```

`mv1` y `mv2` pueden ser compartidas por varios procesos concurrentes sin que se produzca *interferencia*.

```
var mv1 : VariableProtegida(0,1);
mv2 : VariableProtegida(10,4); i1,i2 : integer ;
begin
mv1.incremento() ;
mv1.valor(i1) ; { i1==1 } //permanentes (x,inc) distintas
mv2.incremento() ; //para cada instancia del monitor
mv2.valor(i2) ; { i2==14 }
end
```

Cola del monitor para exclusión mutua

El control de la exclusión mutua se basa en la existencia de una *cola FIFO de entrada al monitor* proceso que ejecute una llamada a uno de sus procedimientos, entrará en el monitor

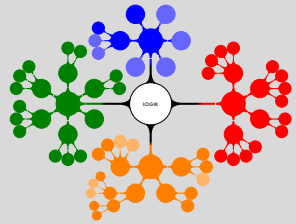


Diagrama de estados de un proceso

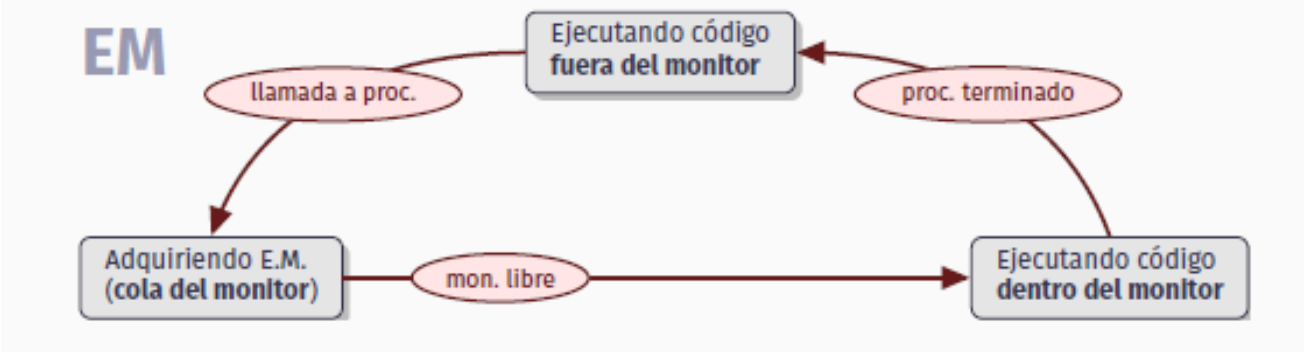
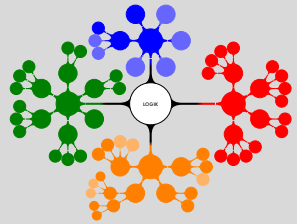
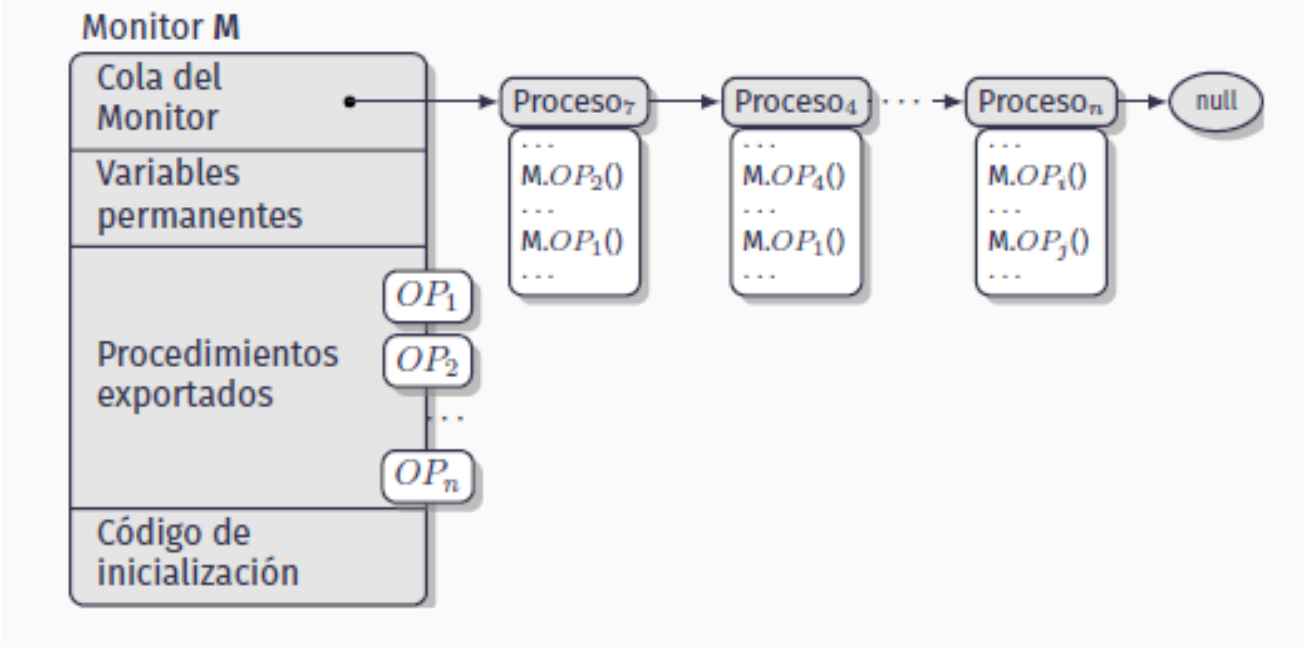


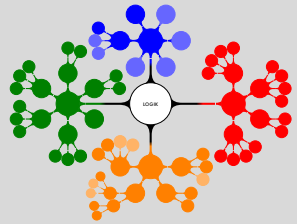
Figure: Posibles estados de los procesos y las transiciones entre dichos estados



Estado del monitor



El estado del monitor incluye la cola de procesos esperando a comenzar a ejecutar el código del mismo



Operaciones de sincronización en monitores

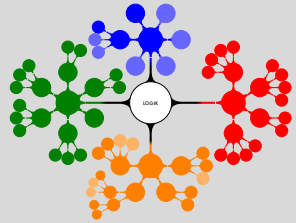
Para implementar la sincronización, se requiere de una facilidad para que los procesos hagan esperas bloqueadas, hasta que sea cierta determinada condición

En semáforos existe:

- La posibilidad de bloqueo (`sem_wait`) y activación (`sem_signal`)
- Un valor entero (el valor de la variable protegida “s” del semáforo)

En monitores, sin embargo, se tiene:

- Sólo se dispone de sentencias de bloqueo y activación
- No hay variable, ni algún valor protegido, asociada a 1 variable condición



Operaciones de sincronización en monitores-II

Encapsulación de la sincronización

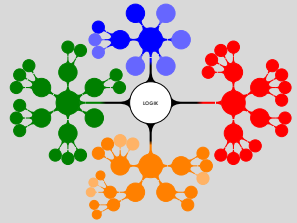
- Explícitamente programada dentro de los procedimientos
- TDA `cond`
- Variables condición (sólo declararlas, no inicializarlas)
- Se define 1 variable condición, por cada una de las condiciones para esperar, en el monitor

`c.wait()`: bloquea siempre. El desbloqueo de los procesos se produce en orden FIFO	`c.signal()`: si la cola de `c` no está vacía, desbloquea al primer proceso de la cola

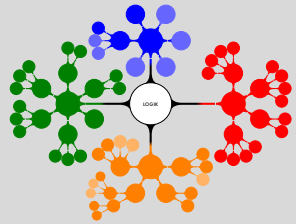
- Las variables condición pertenecen a un tipo abstracto y opaco de datos La representación interna de las variables condición no es accesible al programador de monitores
- La cola de procesos bloqueados asociada a 1 variable condición es FIFO

`c.queue()`:

función lógica que devuelve `true` si hay algún proceso esperando en la cola de cond, y `false` en caso contrario



El TDA *cond* de los monitores



- La exclusión mutua se levanta como consecuencia de ejecutar `c.wait()` cuidado: deja "abierto" el monitor antes de bloquear al proceso
- Se cede el acceso del monitor al proceso señalado (`c.signal()` con semántica desplazante)
- La semántica desplazante de las señales evita un robo de señal (que se *cuele* un tercer proceso en el monitor)

Comportamiento de los procesos después de ejecutar las operaciones de sincronización

- Después de ejecutar `c.wait()`
- Después de ejecutar `c.signal()`

  No pueden programarse operaciones `c.wait()` indebidas, ni tampoco omitirse las operaciones `c.signal` necesarias

- Operaciones `c.queue` y `c.signal_all`
- No es segura la simulación de `c.signal_all` utilizando `c.queue` y señales desplazantes

Estado de un monitor con varias colas

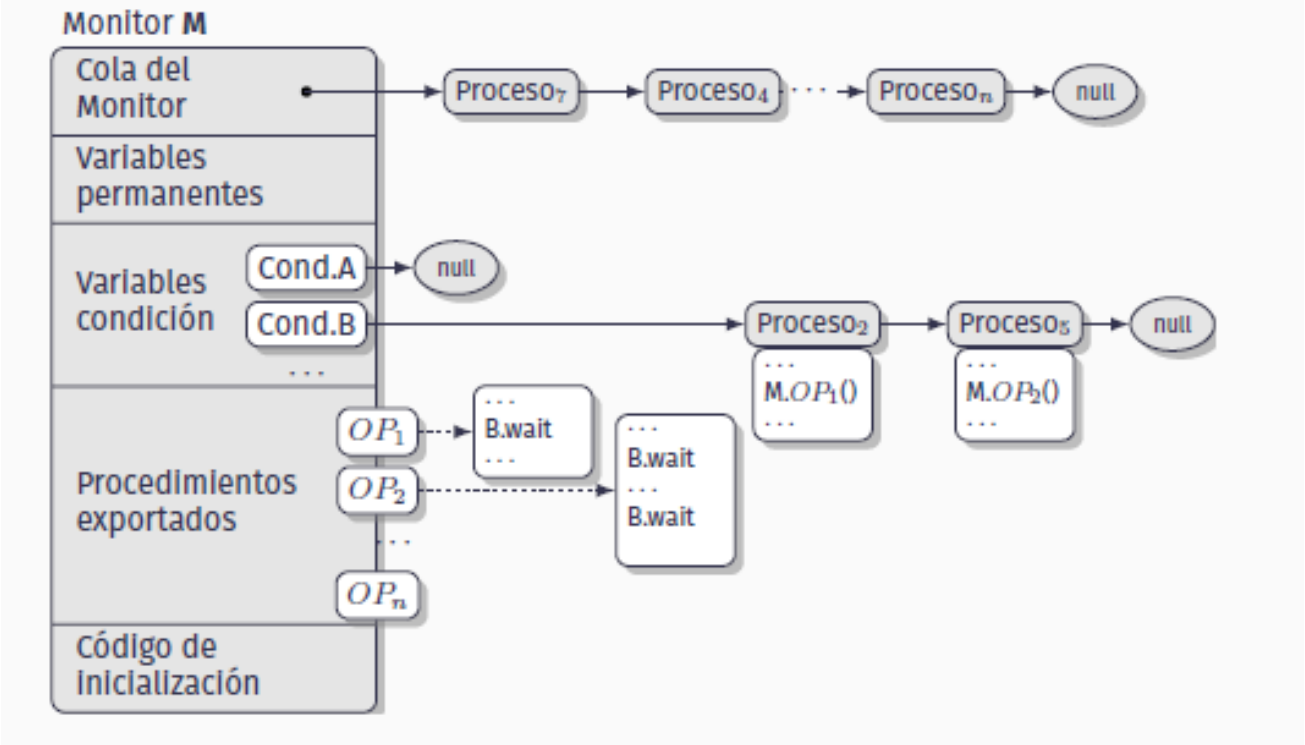
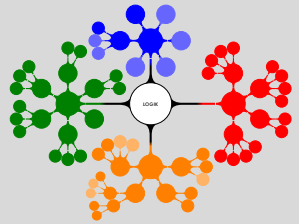


Figure: Los procesos 2 y 5 ejecutan las operaciones 1 y 2, ambas producen esperas de la condición B



Verificación de programas con monitores

El programador no puede conocer a priori la traza concreta de llamadas a los procedimientos del monitor por parte de los procesos del programa concurrente.

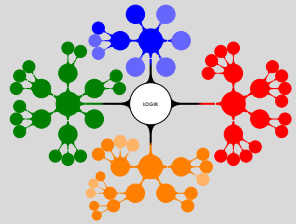
Monitor Buf;

Process P1 (consumidor) || Process P2 (consumidor) || Process P3(productor)
==>Trazas:
P3::Buf.insertar(x);P3::Buf.insertar(x');P2::Buf.retirar(x);P1::Buf.retirar(x'); ...

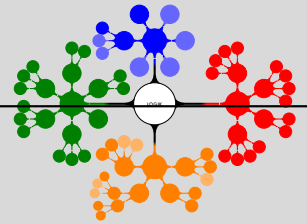
P3::Buf.insertar(x);P2::Buf.retirar(x);P1::Buf.retirar(x') [bloqueado]; P3::Buf.insertar(x');...
...

Invariante de un monitor (IM):

- Es una propiedad que el monitor cumple siempre, pero específico de cada monitor diseñado por un programador



Verificación de programas con monitores-II



Esquema general de demostración de programas con monitores:

-Probar la corrección parcial de los procesos secuenciales

-Comprobar que el invariante de cada uno de los monitores del programa se mantienen: incialmente y antes/después de ejecutar cada pr

-Aplicar la regla de la concurrencia

Características de un IM:

- Su *valor de verdad* depende de los valores de las variables permanentes
- Debe ser cierto en cualquier estado del programa concurrente, excepto cuando un proceso está ejecutando código del monitor

Verificación de los monitores

- Axiomas iniciales

  La demostración de corrección se basa en:

> **Invariante del monitor**(IM)
> relación constante entre los valores permitidos de las variables permanentes del monitor

Ejemplo (del Monitor Buf)`elementos_ocupados` $\leq$ `N`:

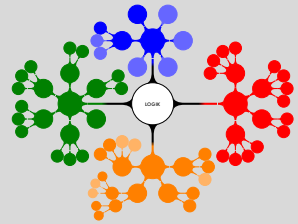
Axioma (inicialización variables)

{V} inicializacion variables permanentes {IM}

La inicialización se lleva a cabo dentro del cuerpo `begin ... end` del monitor

El invariante del monitor debe ser cierto en su estado inicial, justo después de la inicialización de las variables permanentes

Inicialmente: `elementos_ocupados = 0` $\leq$ `N` : dado que `N` $>$ `0`



Verificación de los monitores-II

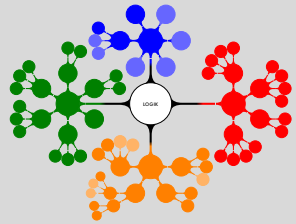
El invariante del monitor debe ser cierto:

- antes y después de cada llamada a un procedimiento del monitor

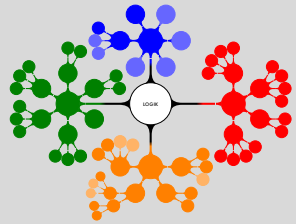
Axioma (procedimientos del monitor)

$$\{\text{IN} \wedge \text{IM}\}\ \text{procedimiento}_i\ \{\text{OUT} \wedge \text{IM}\}$$

- `IN` satisfecho por los p. `in,in/out`
- `OUT` satisfecho por los p. `out,in/out`



Axiomas de operaciones de sincronización con semántica desplazante

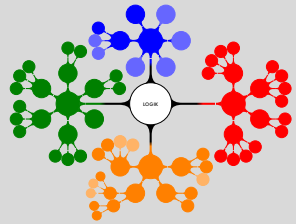


Axioma (operacion `c.wait()`)

$$\{IM \wedge L\}\ c.wait()\ \{C \wedge L\}$$

- El proceso que ocasiona la ejecución de `c.wait` se bloquea y deja libre el monitor
- Entra en la cola FIFO asociada a `c`
- Las demostraciones de corrección no tienen en cuenta la obligación de que el proceso termine de ejecutar `c.wait()`

Axiomas de operaciones de sincronización con semántica desplazante-II



Axioma (operación `c.signal()`)

$$\{\neg \text{vacio(c)} \wedge \text{L} \wedge \text{C}\}\ \text{c.signal()}\{\text{IM} \wedge \text{L}\}$$

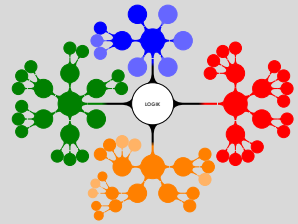
- No tiene efecto si la cola `c` está vacia
- Se supone semántica desplazante: el monitor mantiene el estado expresado por C
- Los axiomas no tienen en cuenta el orden de desbloqueo de los procesos

Ejemplo: verificación con señales desplazantes

Regla verificación del IF: ambas ramas con misma postcondición $\{s > 0\}$

```
Monitor Semaforo;
 var s: integer; {IM: s ≥ 0}
condición de sincronización: {s > 0}
     c: cond;
   procedure P;
     begin
      {IM}
      if s=0 then
        {s = 0 ∧ IM}
        c.wait;
      {s>0}
      else
      {s>0}
        null;
      {s>0}
      endif;
      {s > 0 }
        s:= s-1
      {s ≥ 0 → IM}
     end;
```

```
procedure V;
 begin
  {IM}
     s:= s+1;
    {s>0}
         c.signal;
  {s≥ 0 → IM}
  end;
  begin
   {TRUE}
   s:= 0;
{s ≥ 0} → {IM}
   end;
```

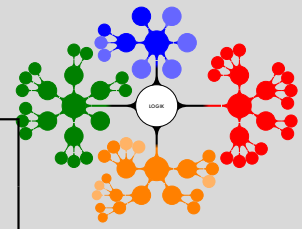
Verificación de un monitor con señales desplazantes-II

Ejercicio: Semáforo de Habermann

$n_p \leq n_a$, ya que primero ha de incrementarse n_a
$n_p \leq n_v$, ya que n_a no puede superar a n_v para incrementar n_p
$n_p \geq \min(n_a, n_v)$, condición no necesaria, pero deseable

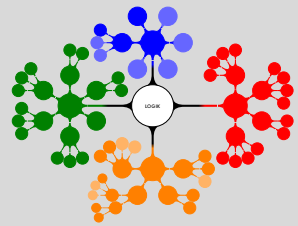
```
Monitor Semaforo;
  var na, np, nv:int
      c: cond;

   procedure P;
   begin
     na:= na+1;
     if(na > nv)
       then c.wait();
     np:= np+1;
    end;

                  na:=0;
                 nv:=0;
                 np:=0;
```

```
procedure V;
 begin
   nv:= nv+1;
   if(na > np)
     then c.signal;
 end;
```

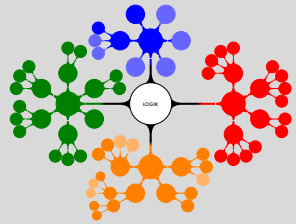
Verificación del Monitor “Semáforo de Habermann”

```
Monitor Semaforo;
  var na, np, nv:int
      c: cond;
  procedure P();
  begin{np==min(na,nv)}
    na:= na+1;
   {np==min(na-1,nv)}
    if(na > nv)
       then
{(na>nv) and np===min(na-1,nv)}=>
np== min(na,nv)
          c.wait();
        {na>np and np===nv-1}
      else null
        { np<nv and np===na-1}
      endif
    {np+1===min(na,nv)}
    np:= np+1;
    {np===min(na,nv)}
  end;

  procedure V();
begin {np==min(na,nv)}
     nv:= nv+1;
{np==min(na,nv-1)}
     if(na > np)
        then
{(na>np) and np===min(na,nv-1)}=>
na>np and np== nv-1
      c.signal;
     {np==min(na,nv)}
   end;

{V}  na:=0;   nv:=0;   np:=0;   {np===min(na,nv)}
```

Regla de la concurrencia para la verificación de programas con monitores

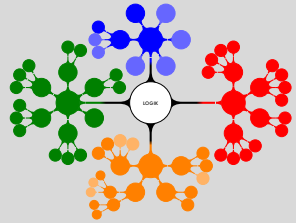
$$\frac{\begin{array}{l}\{P_i\}\, S_i\, \{Q_i\},\ 1 \leq i \leq n \\ \text{ninguna variable libre en } P_i \text{ o en } Q_i \text{ es modificada por } S_j,\ i \neq j \\ \text{todas las variables en } IM_k \text{ son locales al monitor } m_k\end{array}}{\begin{array}{l}\{IM_1 \wedge \ldots\ IM_m \wedge P_1 \wedge \ldots \wedge P_n\} \\ \text{cobegin } S_1 \parallel S_2 \parallel \ \ldots\ \parallel S_n \text{ coend} \\ \{IM_1 \wedge \ldots\ IM_m \wedge Q_1 \wedge \ldots \wedge Q_n\}\end{array}}$$



Patrones de uso de monitores

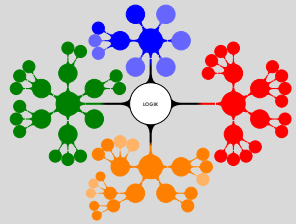
Se estudian a continuación los patrones de solución para tres problemas sencillos, típicos de la Programación Concurrente

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



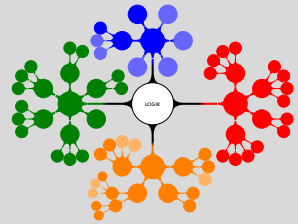
Uso del monitor de EU

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

Monitor para Espera Única (EU)

```
monitor EU;
var terminado:boolean;
//terminado==true si terminado, si no: terminado==false
    leer_autorizado: boolean //variable auxiliar
    cola:condition;
    //cola consumidor esperando a que terminado==true
    export esperar, notificar;
  //Invariante: terminado= false => lee= false
  procedure esperar();//para llamar antes de sentencia  L
  begin
   if (not terminado) then //si no se ha terminado W
       cola.wait();//esperar hasta que termine
   //Condición de sincronización terminado= true
   leer_autorizado:= true;//funciona solo suponiendo
       semantica desplazante de la senial!!!
  end;
  procedure termina();
  begin
    leer_autorizado:= false; terminado:= false;
  end;
  procedure notificar();//para llamar después de E
  begin
    terminado:=true;//Condición de sincronización
    cola.signal();//reactivar al otro proceso, si esa
        esperando
  end
```

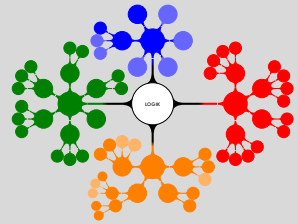



Verificación del monitor EU

$$\textit{Invariante}: \ \textit{terminado} = \textit{false} \ \Rightarrow \ \textit{leer\_autorizado} = \textit{false}$$

```
procedure esperar();
{Invariante, terminado= false, leer_autorizado= false}
if (not terminado) then
  begin
    {terminado= false, leer_autorizado= false, Invariante}
    cola.wait();
  endif;
  {Condicion de sincronizacion: terminado= true}
  leer_autorizado:= true
  {Invariante}
 end;
```

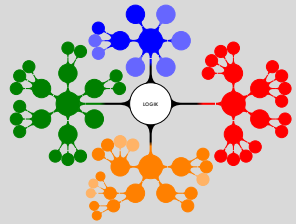
```
procedure notificar();
begin
  {terminado= false, leer_autorizado=false, Invariante}
  terminado:= true;
  {Condicion de sincronizacion: terminado= true}
  cola.signal();
  {Invariante}
end;
```

Uso y propiedades del monitor EM

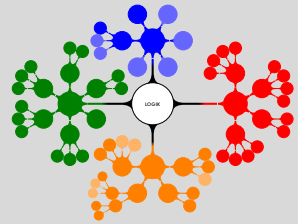
El monitor puede ser utilizado por n procesos concurrentes:

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

Patrón de Exclusión Mutua (EM)

```
monitor EM ;
  var ocupada:boolean;
      //ocupada==true hay un proceso en SC, si no: ocupada
          ==false
      cola:condition;
      //cola de procesos esperando a que 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
    //si al menos un proceso espera, reactivar al primero
    cola.signal();
   end
  begin//inicializacion:
   ocupada:=false;//al inicio no hay procesos en SC
  end
```

Invariante del monitor

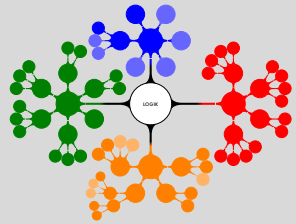
El invariante del monitor, es la conjunción de estas dos condiciones:

- $$IM :: \ ocupada == false \Leftrightarrow num\_sc == 0 \wedge$$
$$0 \leq num\_sc \leq 1$$

  es decir, no puede ejecutar más de 1 proceso la sección crítica.

Demostración del IM:

- Al inicio, IM es cierto (la sección crítica está vacía, luego $num\_sc == 0$ y ocupada == false).
- Demostración de los procedimientos (`entrar()`,`salir()`)del monitor:
  - Declarar una variable permanente ficticia `num_sc` (inicialmente== 0)
  - Modificar el IM como: $libre + num\_sc == 1$ (suponemos la aritmetización de las constantes lógicas)

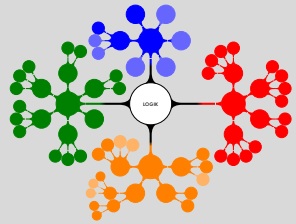


Sincronización tipo Productor/Consumidor

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

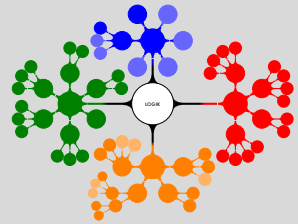
- El procedimiento `escribir` escribe el parámetro en la variable compartida
- El procedimiento `leer`, lee el valor que hay en la variable

```
process Productor;//calcula x
  var a:integer;
  begin
   while true do begin
    a:=ProducirValor();
    PC.escribir(a);//copia a en valor
   end
end
process Consumidor //lee x
  var b : integer ;
  begin
   while true do begin
    PC.leer(b); //copia valor en b
    UsarValor(b);
   end
end
```

Monitor Productor/Consumidor

```
Monitor PC ;
 var valor_com:integer;//valor compartido
     pendiente:boolean;//true: valor escrito y no leido
     cola_prod:condition;
     //espera productor hasta que pendiente == false
     cola_cons:condition;
     //espera consumidor hasta que pendiente == true
  procedure escribir( v:integer );//sentencia E
   begin
    if pendiente then
      cola_prod.wait();
      valor_com:=v;
      pendiente:=true;
      cola_cons.signal();
   end;
  function leer():integer;//sentencia L
   begin
   if (not pendiente) then
     cola_cons.wait();
     result:=valor_com;
     pendiente:=false;
     cola_prod.signal();
   end;
   begin // inicializacion }
     pendiente := false ;
   end;
```

Verificación del monitor

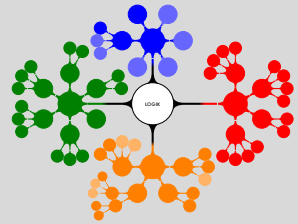
Al igual que en los otros casos, podemos verificar que el monitor funciona bien:

- #E = número de llamadas a escribir completadas
- #L = número de llamadas a leer completadas
- El invariante del monitor (IM) es:

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

Demostración del IM:

- Se demuestra igual que en el caso del Patrón EM, sustituyendo:
  - $\#E - \#L == num\_sec$ y
  - *pendiente* == *NOT libre*



Señales con prioridad

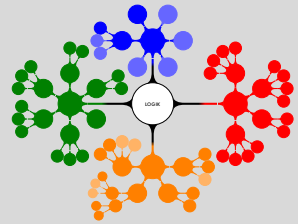
- La semántica de las señales no contempla el desbloqueo de los procesos según un orden prioritario

`c.wait(prioridad)` bloquea a los procesos en la cola `c`, pero ordenándolos con respecto al valor del argumento prioridad

Despertador que recuerda los tiempos indicados por sus usuarios. Varios de ellos pueden indicar la misma hora

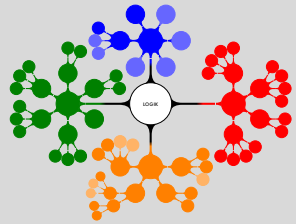
```
procedure tick();//cableada a INT CLK
  begin
    ahora:= ahora +1;
    despertar.signal();
  end;
begin
  ahora:= 0;
end;
```

`ahora` es una variable del sistema que se obtiene procesando el resultado de ejecutar la llamada `clock()` del sistema operativo



Señales con prioridad-II

```
monitor despertador;
  var
    ahora: Long_integer;
    despertar: cond; --prioritaria

  procedure despiertame(n: integer);
    var alarma: Long_integer;
    begin
      alarma:= ahora + n;
      while ahora< alarma do
        despertar.wait(n);
      end do;
      despertar.signal();
    end;
```

Señales con prioridad-III

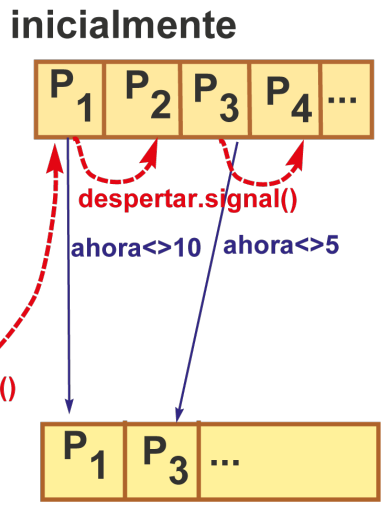
- Simulación mediante señales FIFO con semántica desplazante

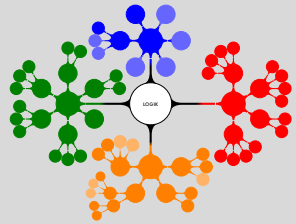
```
Monitor despertador[d:tipo_enumerado]
 var
  ahora:int;
  despertar:cond;
 procedure despiertame(n:int)
  begin
   alarma:= ahora + n;
   while(ahora< alarma) do
    begin
     despertar.signal();
     despertar.wait();
    end;
   despertar.signal;
  end;
 procedure tick
   begin
    ahora:= ahora + 1;
    despertar.signal();
   end;
 begin
  ahora:= 0;
 end;
```

Escenario de ejecucion

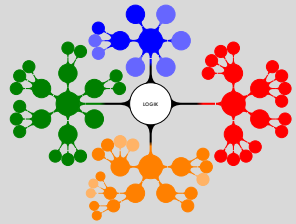
id	instruccion	tiempo
P1	despiertame(10)	0
P2	despiertame(2)	1
P3	despiertame(3)	2
P4	despiertame(0)	3 ahora=3

despertar.signal()





Todos los mecanismos de señalación alternativos de los monitores

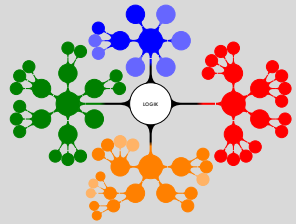


SA	señales automáticas	señal implícita
1 SC	señalar y continuar	señal explícita, no desplazante
2 SS	señalar y salir	señal explícita, desplazante, el proceso sale del monitor
3 SE	señalar y esperar	señal explícita, desplazante, el proceso señalador espera en la cola de entrada al monitor
4 SU	señales urgentes	señal explícita, desplazante, el proceso señalador espera en la cola de *procesos urgentes*

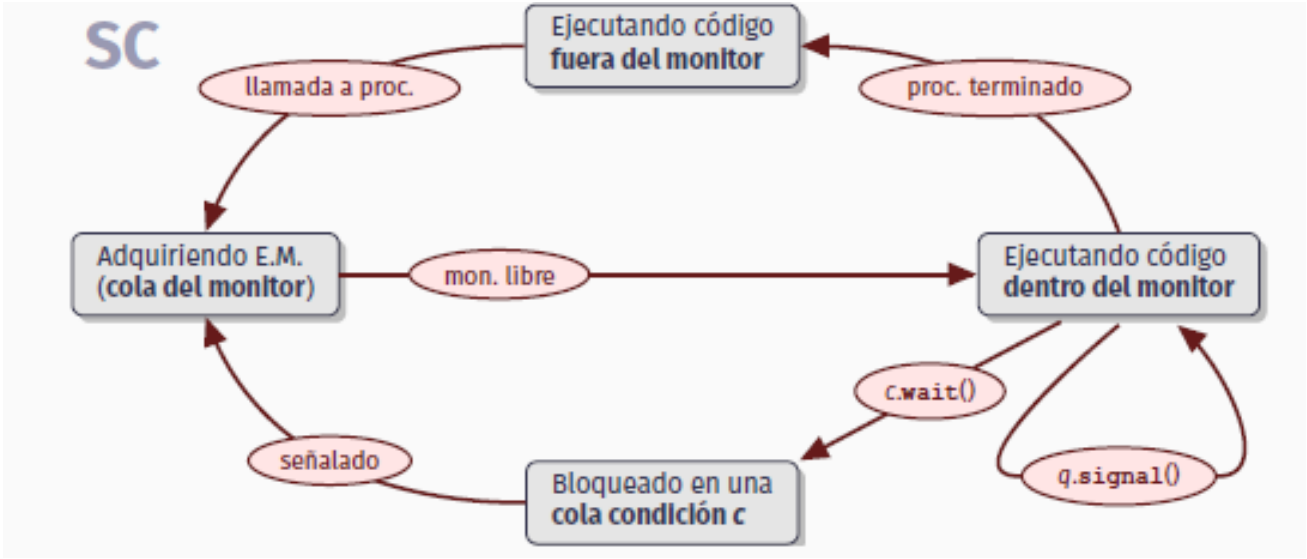
Posibles semánticas de las señales de los monitores

1. [SC] El proceso señalador continua su ejecución tras la operación `signal`

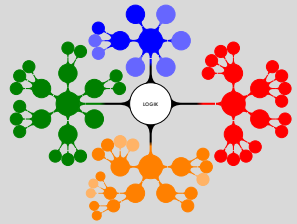
El proceso señalado espera bloqueado hasta que puede adquirir la E.M. de nuevo, semántica de señal:*SC: señalar y continuar*



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



- Señalador: continúa inmeditamente la ejecución de código del procedimiento 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 la operación `wait`

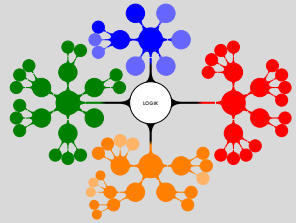


Señalar y continuar (SC): características

El proceso señalador continúa su ejecución dentro del monitor después del signal

- El proceso señalado abandonará después la cola condición y espera en la cola del monitor para readquirir la E.M.
- Tanto el señalador como otros procesos pueden hacer falsa la condición despues de que el señalado abandone la cola condición
- Por tanto, no se puede garantizar que la condición de sincronización asociada a **cond** sea cierta al volver **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:

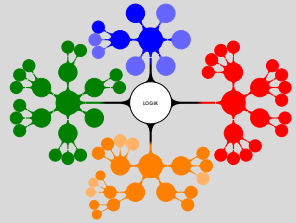
```
while not condicion_logica_sincronizacion do
        cond.wait() ;
```

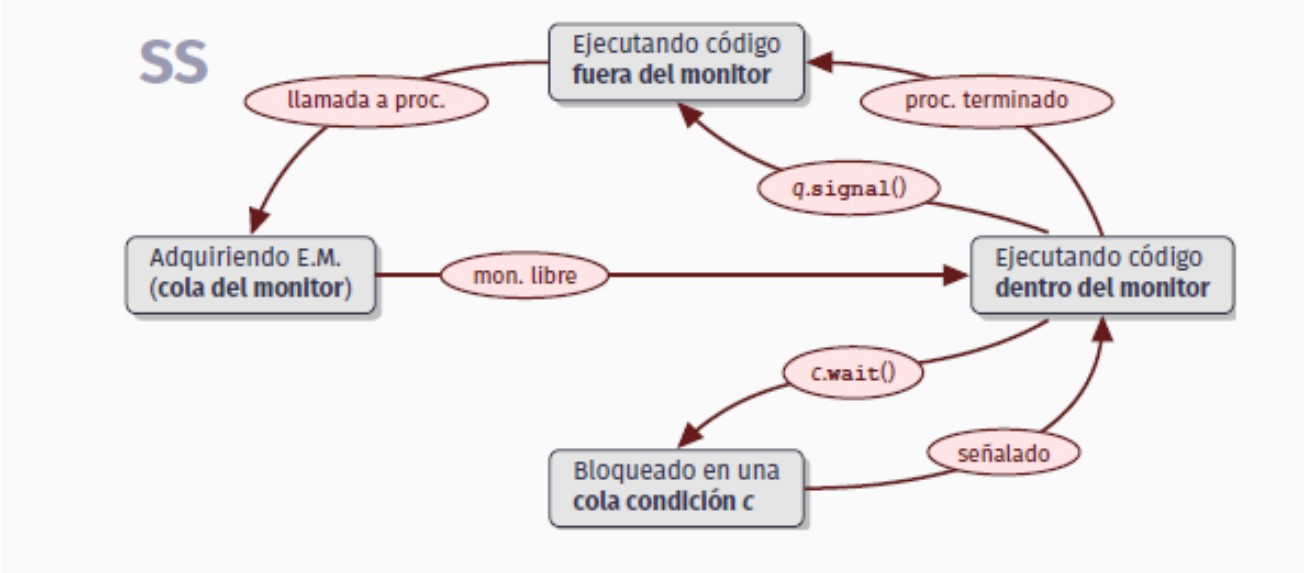
Señalar y Salir

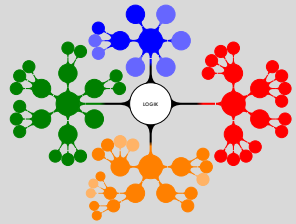
[SS] El proceso señalado se reactiva inmediatamente

2. El proceso señalador abandona el monitor tras hacer `signal`, sin ejecutar el código que haya después de dicho `signal`, la semática de señal:*SS: señalar y salir* admite varias implementaciones.



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

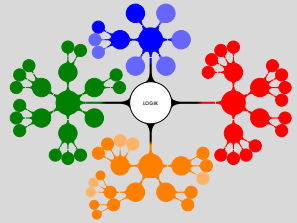




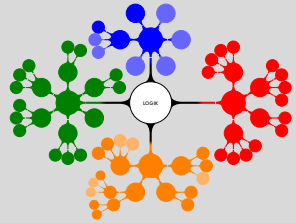
Señalar y salir (SS): características

El proceso señalador sale del monitor después de ejecutar cond.`signal()`

- Si hay código tras signal, no se ejecuta. El proceso señalado reanuda inmediatamente la ejecución de código del monitor.
- En ese caso, la operación signal conlleva:
  - Liberar al proceso señalado
  - Terminación del procedimiento del monitor que estaba ejecutando el proceso señalador porque se le obliga a salir del monitor (señales SS)
  - La condición de sincronización se mantiene hasta que el señalado continua su ejecución.
  - Esta semántica condiciona el estilo de programación:
    - operación signal como última instrucción de los procedimientos
    - o antes de programar una operación c.wait()



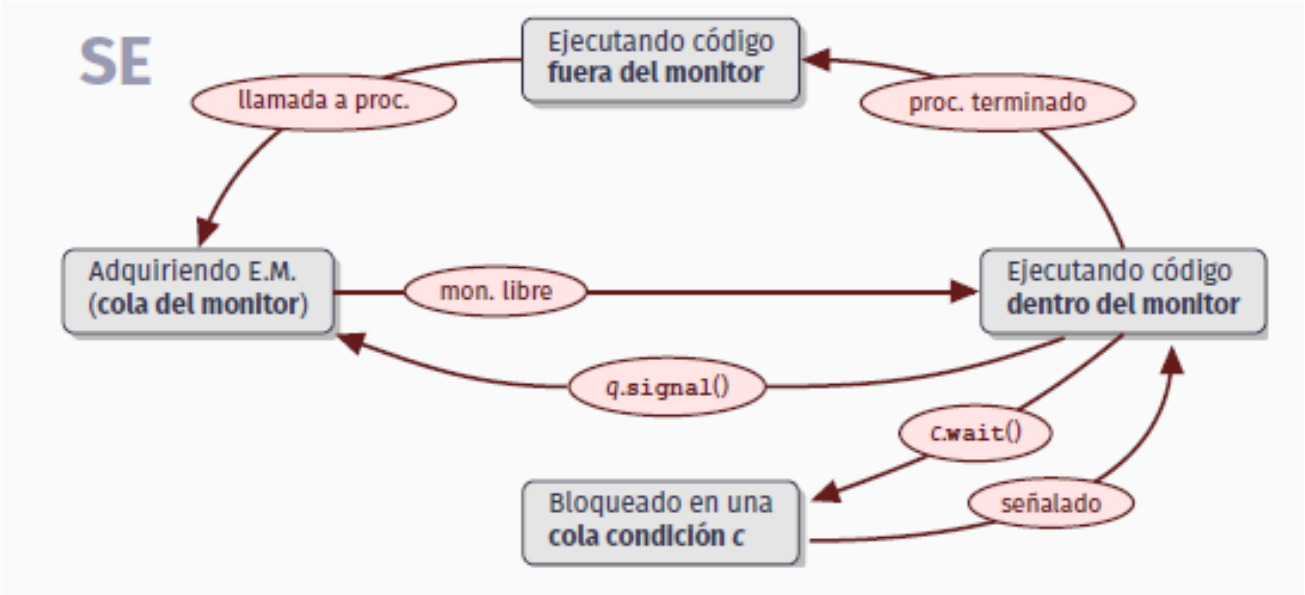
Señalar y esperar (SE)

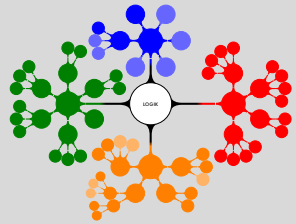


3. **[SE] El proceso señalador se bloquea en la cola del monitor justo después de ejecutar** `signal()`

   - El proceso señalado entra de forma inmediata en el monitor
   - Está asegurada el cumplimiento de la condición de sincronización
     - El proceso señalador entra en la cola de procesos del monitor
     - Puede considerarse una semántica injusta respecto del progreso del proceso señalador

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

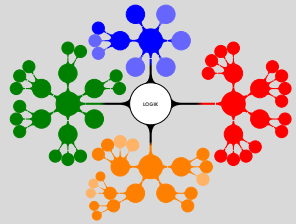




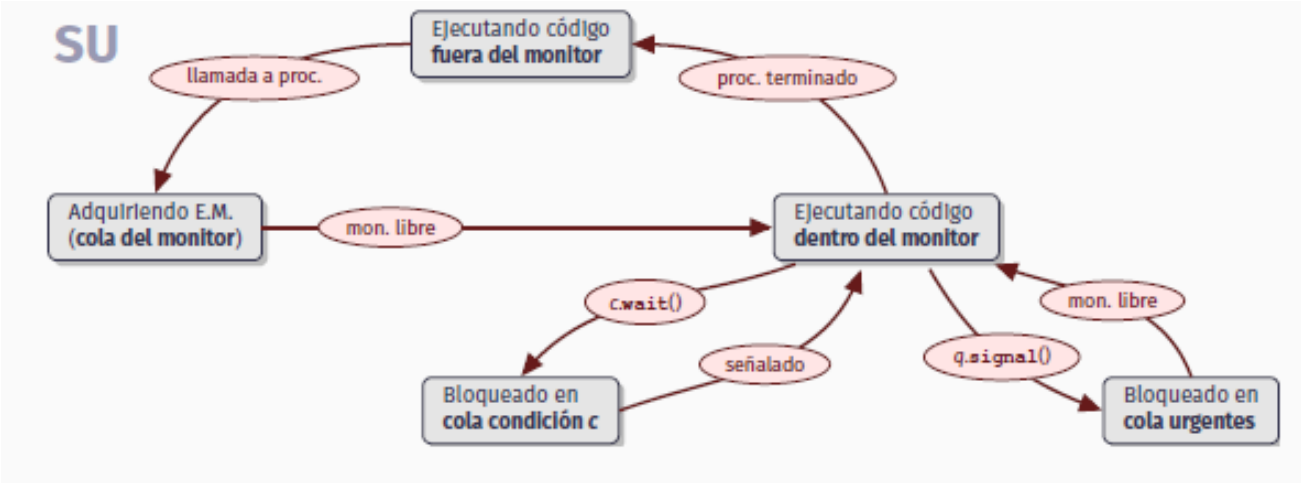
Señalar y espera urgente (SU): características

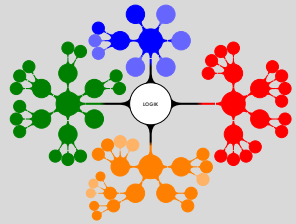
4. **[SU] Es similar la semántica SE, pero se intenta corregir el problema de falta de equidad de los señaladores**

   - El proceso señalador se bloquea justo después de ejecutar la operación `signal()`
     1. El proceso señalado entra de forma inmediata en el monitor
     2. El proceso señalador entra en una nueva cola de procesos del monitor prioritarios o cola de *procesos urgentes*
     3. Los procesos de la cola de procesos urgentes tienen preferencia cuando se queda libre el monitor



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





Procesos en la cola de urgentes

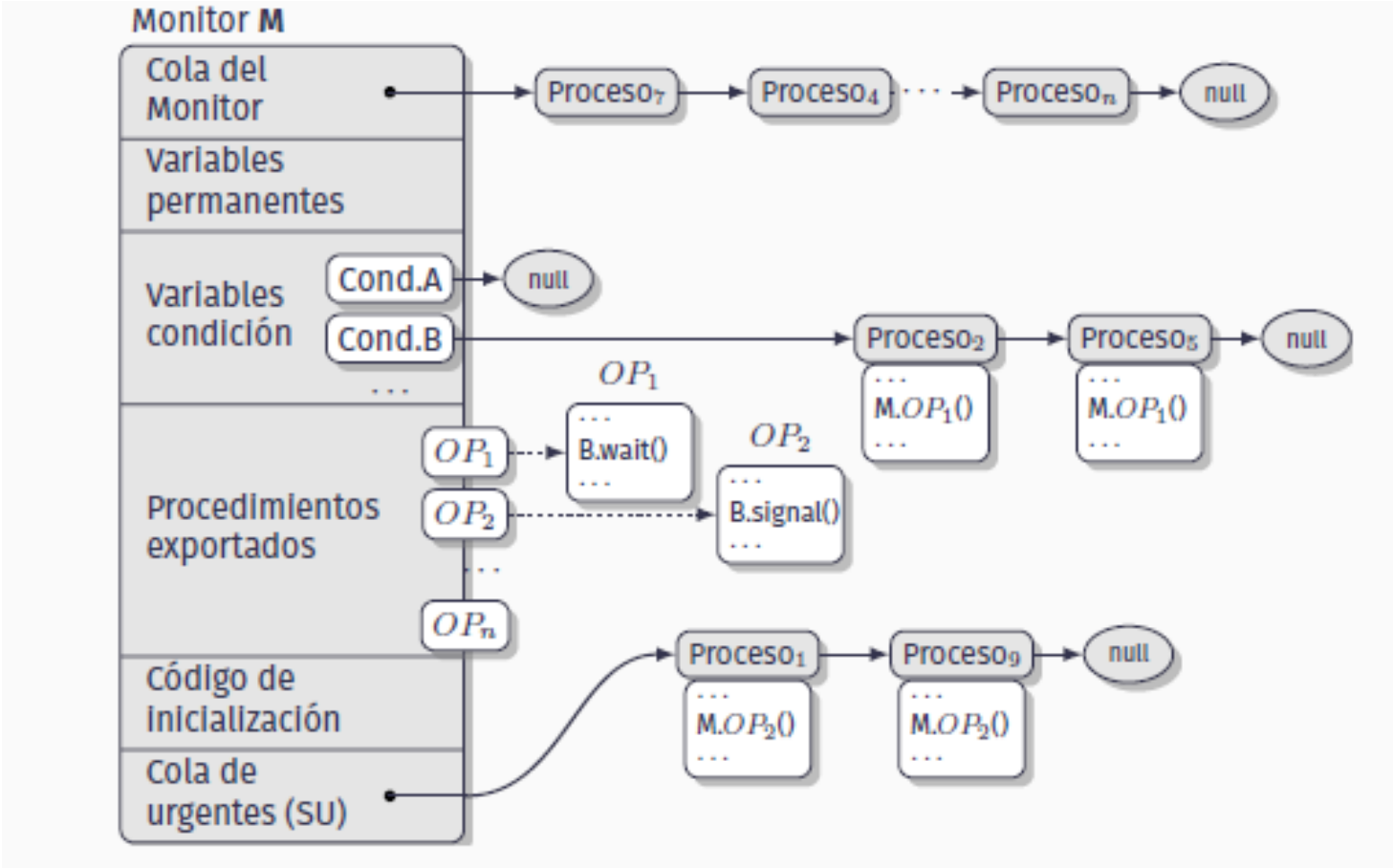
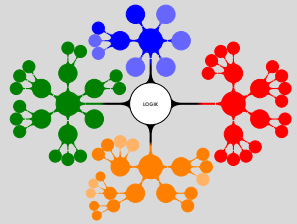


Figure: El proceso 1 y el 9 han ejecutado la op.2, que hace signal de la **cond**. B.



Análisis comparativo de las diferentes semánticas de señales

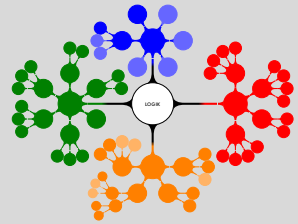
Potencia expresiva: todas las semánticas son capaces de resolver los mismos problemas

Facilidad de uso:

La semántica SS condiciona el estilo de programación y puede llevar a aumentar de forma artificial el número de procedimientos

Eficiencia:

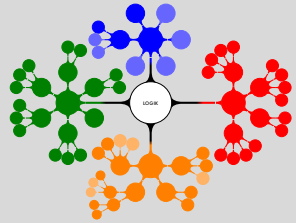
- Las semánticas SE y SU resultan ineficientes cuando no hay código tras `signal`
- La semántica SC también es un poco ineficiente al obligar a usar un bucle de comprobación para evitar el *robo de señal*



Comparativa entre las distintas semánticas de señales mediante un ejemplo

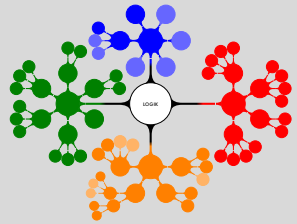
Barrera parcial

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



Una posible implementación del monitor

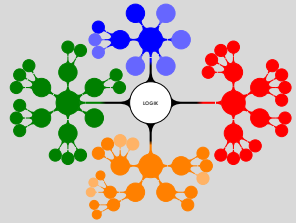
```
Monitor BP //monitor Barrera Parcial }
  var cola : condition;//procesos esperando contador==n
      contador : integer;//numero de procesos ejecutando
          cita
procedure cita() ;
  begin
   contador := contador+1;//registrar un proceso mas en el
        estado "ejecutando cita"
   if (contador<n) then
       cola.wait();//esperar a que haya n procesos en el
           estado "ejecutando cita"
      else begin //si ya hay n procesos ejecutando la cita
         for i := 1 to n-1 do //para cada uno de estos
           cola.signal();//despertalo
         contador := 0;//volver a poner el contador a 0
      end
      print("salgo de cita");//mensaje de salida
  end
begin//inicializacion del monitor
contador := 0 ; { inicialmente, no hay procesos en cita }
```

Comportamiento del monitor *barrera parcial* para las distintas semánticas de señales

Si llamamos último al último proceso en llegar a la cita de cada grupo de n (el que observa contador==n), ocurrirá lo siguiente:

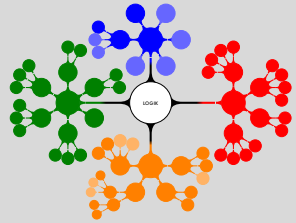
- *Señalar y Continuar:* Los n - 1 procesos señalados abandonan el `wait`, pero pasan a la cola del monitor. Los procesos del grupo de la siguiente cita se podrían adelantar.
- *Señalar y Salir:* en este caso, el último proceso abandona el monitor , no pone `contador` a 0.



Comportamiento del monitor *barrera parcial* para las distintas semánticas de señales-II

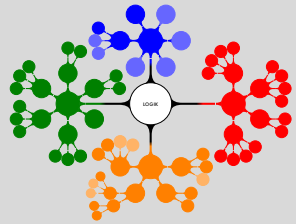
...

- *Señalar y Esperar:* Similar a la semántica SS, no es correcta tampoco esta solución.
- *Señalar y Espera Urgente:* el último proceso ejecuta los `signals` a los `n-1` procesos restantes. Entre cada dos de ellos, espera en la *cola de urgentes* a que el proceso recién señalado abandone el monitor.
- Cuando no quedan procesos bloqueados que señalar, el último proceso vuelve a entrar en el monitor y ejecuta la instrucción `contador:=0`, antes que entre otro proceso al monitor.
- La semántica SU de señales hace que esta solución sea correcta



Implementación alternativa del monitor *barrera parcial*

```
Monitor BP
var cola:{\bf cond}ition;//procesos esperando contador==n
    contador:integer;//numero de procesos esperando en la
        cola
  procedure cita() ;
   begin
    contador:=contador+1;//registrar un proceso mas
        esperando
    if (contador<n) then//todavia no hay n procesos:
      cola.wait();//esperar a que los haya
    contador:=contador-1;//registrar un proceso menos
        esperando
    print("salgo de la cita");//mensaje de salida
    if (contador>0) then//si hay otros procesos en la cola
      cola.signal();//despertar al siguiente
   end
begin//inicialización:
  contador:=0;//inicialmente, no hay procesos en la cola
end;
```

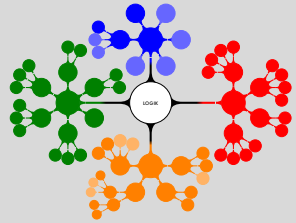



Comportamiento de la versión alternativa

Analizamos el comportamiento en todas las semánticas:

- No funciona con la semántica SC.
- Sí funciona con el resto de semánticas (SE,SS,SU).

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



Reglas de verificación de las señales no-desplazantes SC

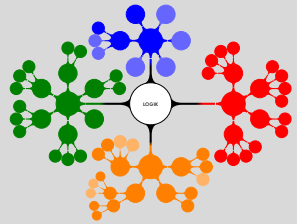
Axioma de la operación `c.wait()`

$$\{\mathtt{IM} \wedge \mathtt{L}\}\ \mathtt{c.wait()}\ \{\mathtt{IM} \wedge \mathtt{L}\}$$

- La regla permite demostrar la corrección parcial de los programas independientemente de la planificación de los procesos de la cola `c`
- No demuestra, por tanto, por tanto ni la propiedad de vivacidad, ni detecta bloqueos

Axioma de las operaciones `c.signal(), c.signal_all()`

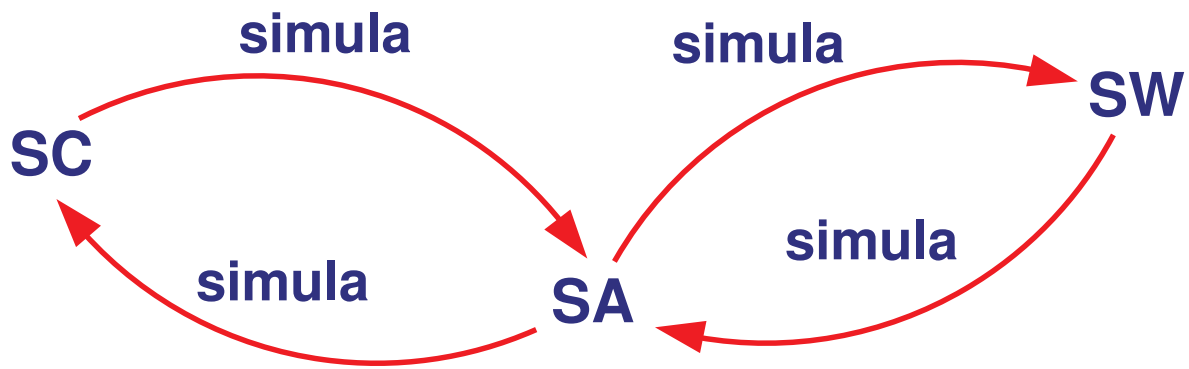
$$\{\mathtt{P}\}\ \mathtt{c.signal()}\ \{\mathtt{P}\}$$

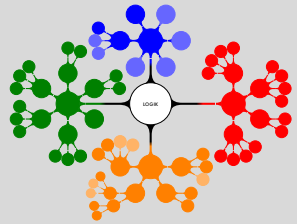


Intercambio de señales en programas que usan monitores

condiciones que se han de cumplir para poder intercambiar SW y SC sin modificar adicionalmente el código del monitor:

1. Sólo se ha exigir como post**cond**ición de `c.wait()` el Invariante del Monitor
2. Después de una llamada a la operación `c.signal()` se ha de *salir* del monitor
3. No se puede utilizar `c.signal_all()`: difusión de una señal a un grupo de procesos

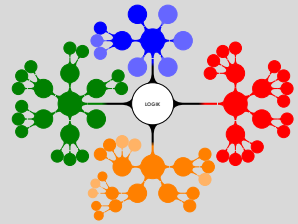




Implementaciones alternativas de un semáforo con monitores

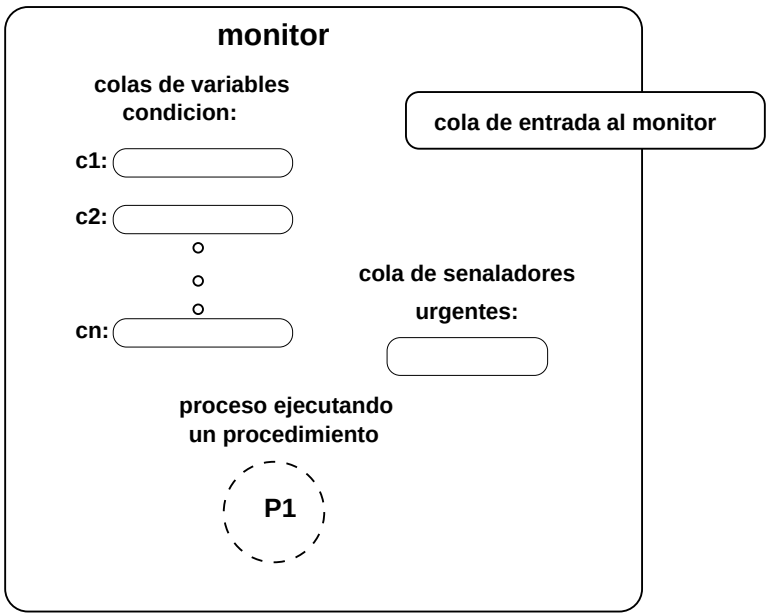
```
Monitor semaforo_FIFO1;
 {IM: s>=0}
var c: cond;
    s: int;
procedure P;
 begin
  if(s=0) then
      c.wait();
      {s > 0}
   s:=s-1;
 end;

procedure V;
 begin
  s:= s+1;
  c.signal;
 end;
begin
  s:=0;
end;
```

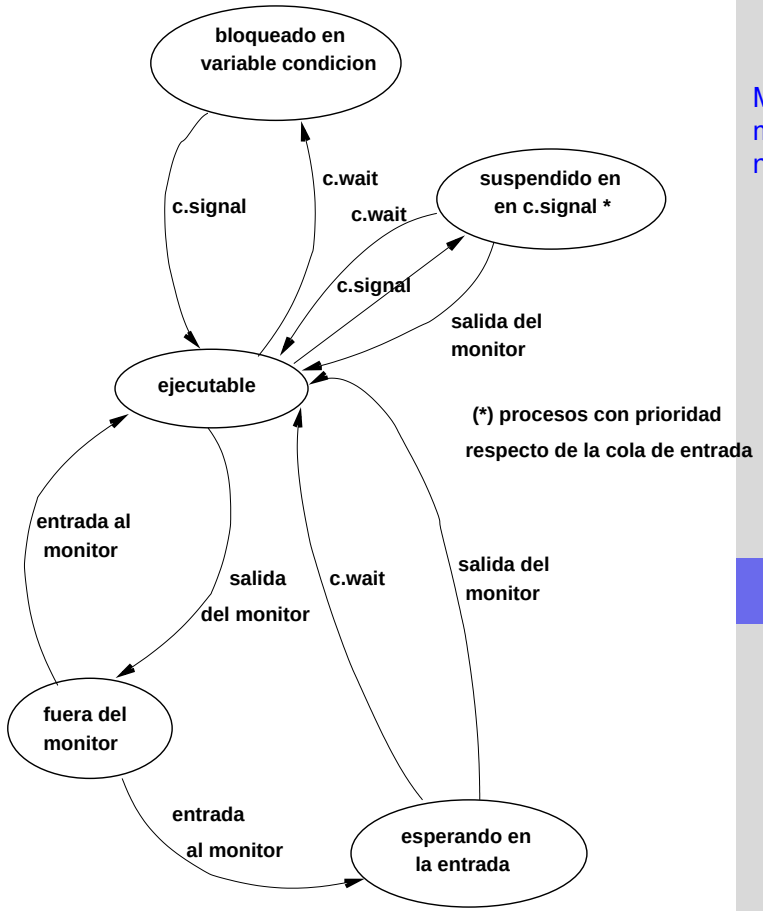
```
Monitor semaforo_FIFO2;
{IM: s>=0}
var c: cond;
    s: int;
procedure P;
 begin
  while(s=0) do
      c.wait();
      {s>= 0}
   end do;
   {s > 0}
   s:=s-1;
 end;

procedure V;
 begin
  c.signal;
  s:= s+1;
 end;
begin
  s:=0;
end;
```

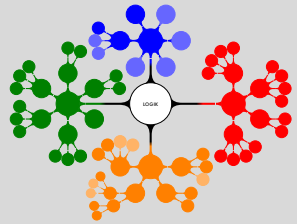



Implementación de los monitores

Esquema de implementación de un monitor con señales SU



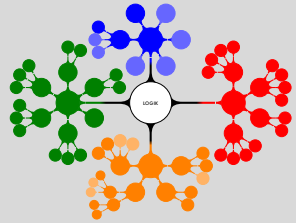




Implementación de los monitores con semáforos

Concepto fundamental:

- Cola de entrada al monitor: controlada por el semáforo mutex
- Cola de procesos urgentes: controlada por el semáforo next
- Número de procesos *urgentes* en cola: se contabiliza en la variable next_count
- 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)

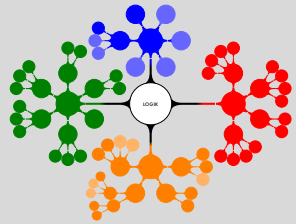


Implementación de los monitores con semáforos-II

Semáforo mutex para implementar la exclusión mutua del monitor

```
procedure P1(...)
begin
sem_wait(mutex);
{ cuerpo del procedimiento }
sem_signal(mutex);
end
```

```
//inicializacion
mutex := 1 ;
```

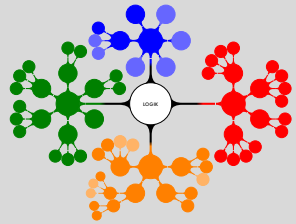



Implementación de los monitores con semáforos-III

Semáforo next para implementar la cola de *urgentes* y next_count para contar los procesos 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 ;
```

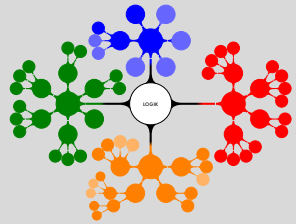



Implementación de los monitores con semáforos-IV

Para implementar las variables condición: semáforo x_sem para cada una y una variable para contar los procesos bloqueados en su cola asociada: x_sem_count

```
void entrada(){
//implementacion de entrada al monitor
sem_wait(mutex); //entre al monitor o se queda bloqueado
    en la cola de entrada
}
```

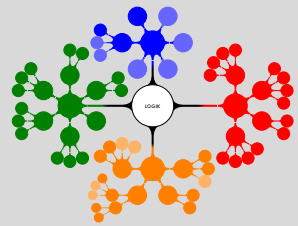
```
void x_wait(Semaphore x_sem, unsigned x_sem_count){
//implementacion de x.wait()
x_sem_count := x_sem_count + 1 ;//cuenta 1 proceso mas
    bloqueado en la cola de condicion "x"
if (next_count <> 0) then//hay procesos señaladores
    esperando
    sem_signal(next); //desbloquea un proceso señalador
   else
    sem_signal(mutex);  //deja libre el monitor para que
        entre otro proceso
sem_wait(x_sem);//se bloquea esperando la certeza de
    condicion "x"
x_sem_count := x_sem_count - 1;//cuenta 1 proceso menos
    bloqueado en la condicion "x"
}
```

Implementación de los monitores con semáforos-V

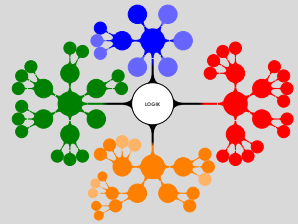
```
void x_signal(Semaphore x_sem, unsigned x_sem_count){
//implementacion de x.signal()
if (x_sem_count <> 0) then
  begin //hay procesos bloqueados esperando la condicion "
      x"
    next_count := next_count + 1;//cuenta 1 proceso mas en
        la cola de señaladores
    sem_signal(x_sem); //desbloquea 1 proceso esperando:
        la condicion "x" es cierta ahora
    sem_wait(next); //entra en la cola de señaladores
    next_count := next_count - 1;//cuenta 1 proceso
        señalador menos en cola de señaladores
  end
}
```

```
void salida(){
//implementacion de la salida del monitor
if (next_count <> 0) then//hay procesos senialadores
    esperando
     sem_signal(next); //desbloquea un proceso senialador
  else
     sem_signal(mutex);  //libera la exclusion mutua del
         monitor
}
```

Ejemplo de uso de la implementación del Monitor

```
Semaphore puede_escribir= 0, puede_leer= 0;
//Monitor PC ;
int valor_com,puede_escribir_count,puede_leer_count;
boolean pendiente;
  procedure escribir(puede_escribir,puede_escribir_count);
   begin entrada();
    if pendiente then
      x_wait(puede_escribir, puede_escribir_count);
    valor_com:=v; pendiente:=true;
    x_signal(puede_leer, puede_leer_count);
    salida();
   end;
  function leer(puede_leer,puede_leer_count):integer;
   begin entrada();
   if (not pendiente) then
     x_wait(puede_leer, puede_leer_count);
   result:=valor_com; pendiente:=false ;
   x_signal(puede_escribir, puede_escribir_count);
   salida();
  end;
  begin entrada();
    pendiente := false ;
    valor_com=0;puede_escribir_count=0;puede_leer_count=0;
    salida();
  end;
```

Bibliografía

Para más información, ejercicios, además de la bibliografía fundamental, se puede consultar:

- Concurrent Programming, Andrews (1991), capítulo 6.
- Programación Concurrente y en Tiempo Real, Capel (2022), capítulo 2.