Tema 2

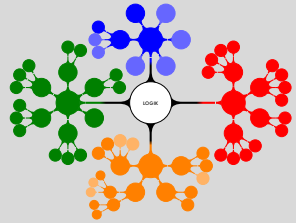
Sincronización en memoria compartida

SCD para GIIM

Asignatura *Sistemas Concurrentes y Distribuidos*

Fecha 7 octubre 2021

Departamento de Lenguajes y Sistemas Informáticos
Universidad de Granada

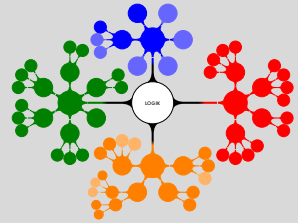


Limitaciones de los semáforos

Hay algunos inconvenientes de usar mecanismos como los semáforos:

- Basados en variables globales: 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 (*protegidas* de los semáforos) no se hace explícito en el programa, lo cual dificulta razonar sobre la corrección de los programas
- Las operaciones se encuentran dispersas y no protegidas (posibilidad de errores)

Por tanto, es necesario un mecanismo que permita el acceso estructurado a los datos del programa y la encapsulación de las estructuras de datos y que, además, proporcione herramientas para garantizar la exclusión mutua e implementar condiciones de sincronización



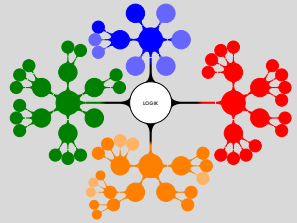
Estructura y funcionalidad de un monitor

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

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

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

- Se garantiza el acceso en exclusión mutua a las variables encapsuladas
- Se implementan la sincronización requerida por el problema mediante esperas bloqueadas (los procesos del programa suspenden sin consumir ciclos del procesador)

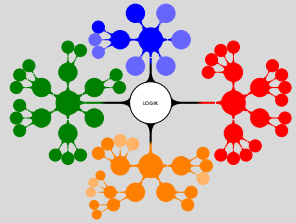


Características generales de los monitores

Concepto de monitor

Módulo de las aplicaciones concurrentes que se usa como un objeto al que se accede concurrentemente por los procesos

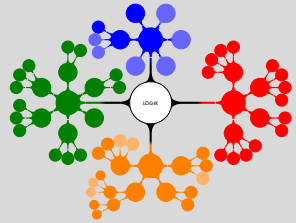
- Modularidad en el desarrollo de programas y aplicaciones
- Programa= {Monitores, Procesos}
- Estructuración en el acceso a tipos de datos, variables compartidas, etc.
- Capacidad de modelado de interacciones cooperativas y competitivas entre procesos concurrentes lo más general posible
- Ocultación 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



Concepto de *monitor* II

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

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



Componentes de un monitor

Variables permanentes: son el estado interno del monitor

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

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

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

Código de inicialización: fija el estado interno inicial

- Este bloque es opcional
- Se ejecuta una única vez, antes de cualquier llamada a procedimientos del monitor

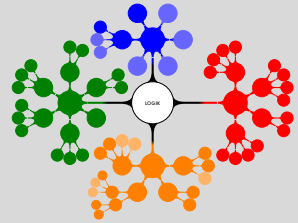


Diagrama de los componentes del monitor

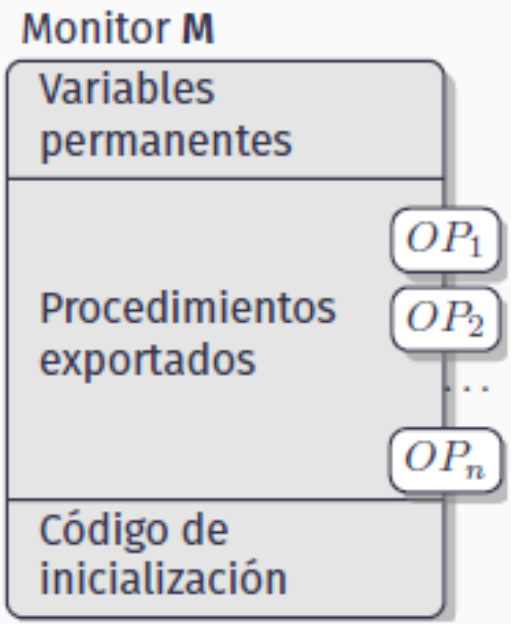
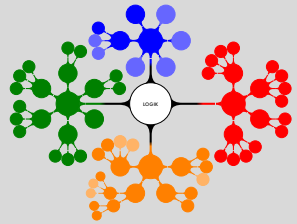


Figure: Un monitor se puede visualizar como aparece en el diagrama

- El uso que se hace del monitor se hace exclusivamente a través de los procedimientos exportados, que constituyen su 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 el código de los programas que las utilizan



Concepto de *monitor* III

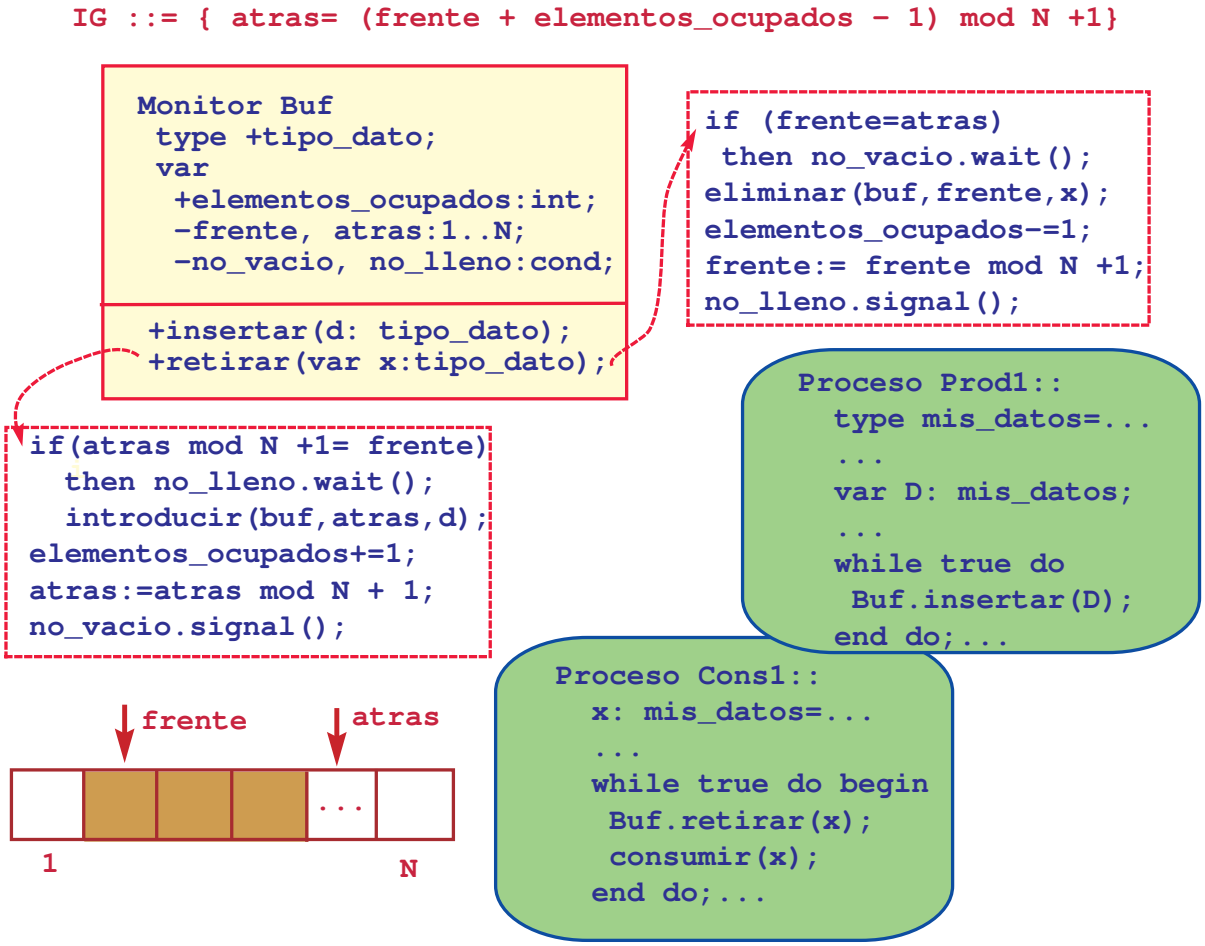
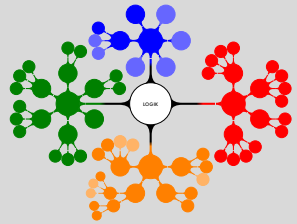
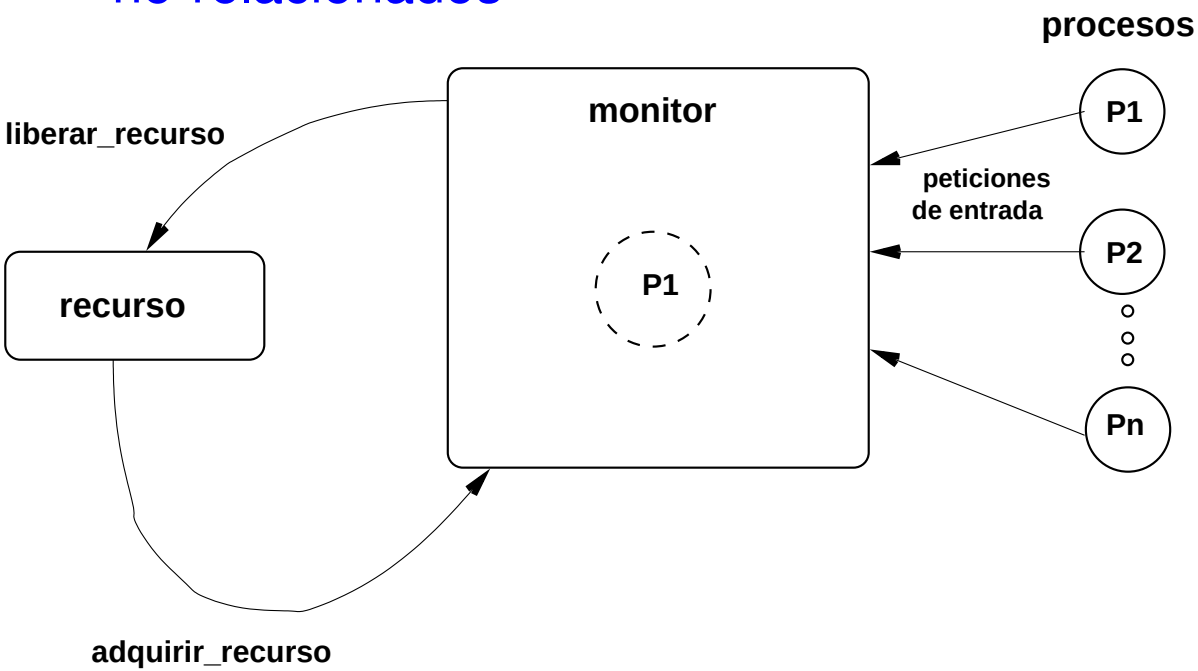


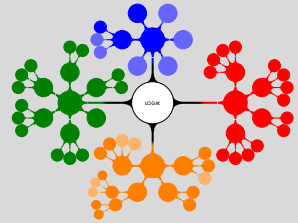
Figure: Representación de gráfica de un módulo monitor



Características de la programación con *monitores*

- Centralización de recursos críticos
- Monitor como versión descentralizada del monitor monolítico de los sistemas operativos
- Mantiene la seguridad de los datos compartidos incluso si sus procedimientos son ejecutados concurrentemente por múltiples procesos
- sólo 1 procedimiento ejecutado por un solo proceso dentro del monitor, aunque puede interrumpirse y reemprenderse posteriormente
- Posibilidad de ejecución concurrente de monitores no-relacionados





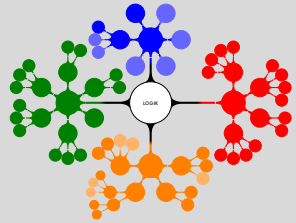
Instanciación de los monitores

Los monitores son “objetos” pasivos

Después de ejecutarse el código de inicialización, un monitor es un módulo pasivo del programa y el código de sus procedimientos sólo se ejecuta cuando estos son invocados por los procesos

Objetivo de la instanciación

- Obtener réplicas de los recursos de un monitor que se puedan acceder simultáneamente por los procesos de un programa concurrente
- Conseguir un principio similar al de *DCE Distributed File System*: procesos distintos se conectan desde diferentes máquinas para acceder a un espacio (recursos) único.¡ Los procesos remotos no distinguen a los ficheros remotos, que utilizan con *locales*



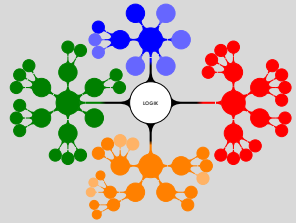
Instanciación de los monitores

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

Condición para que un compilador pueda ofrecer monitores instanciables a los programadores:

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

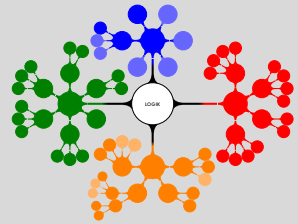
- No dependerá de datos estáticos globales
- No puede modificar su propio código
- No puede llamar a funciones *no-reentrantes*



Instanciación de los monitores-II

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

```
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 de entrada al 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 (espera) 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 propiedad de *vivacidad* del programa con monitores, la planificación de la cola del monitor debe seguir una política FIFO

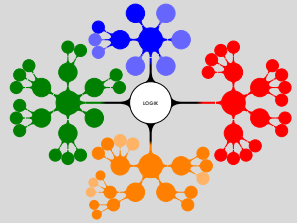


Diagrama de estados de un proceso

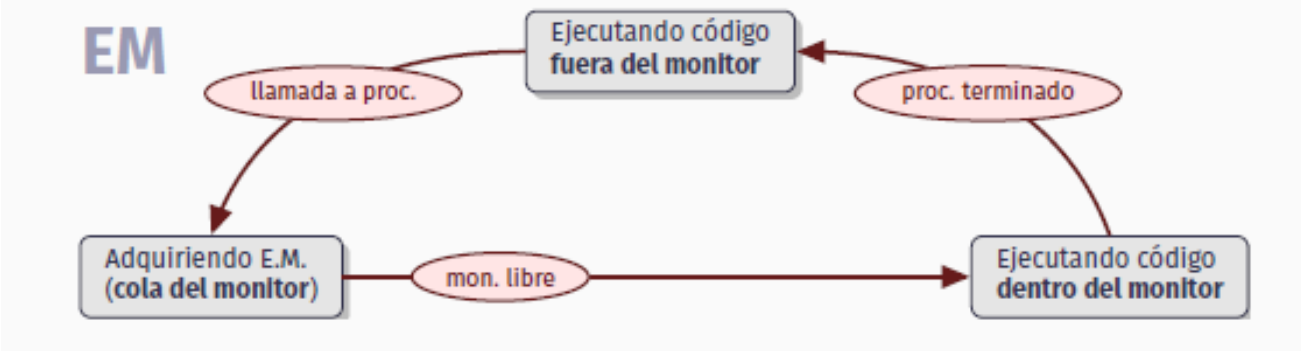
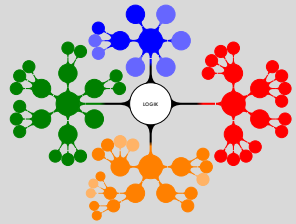
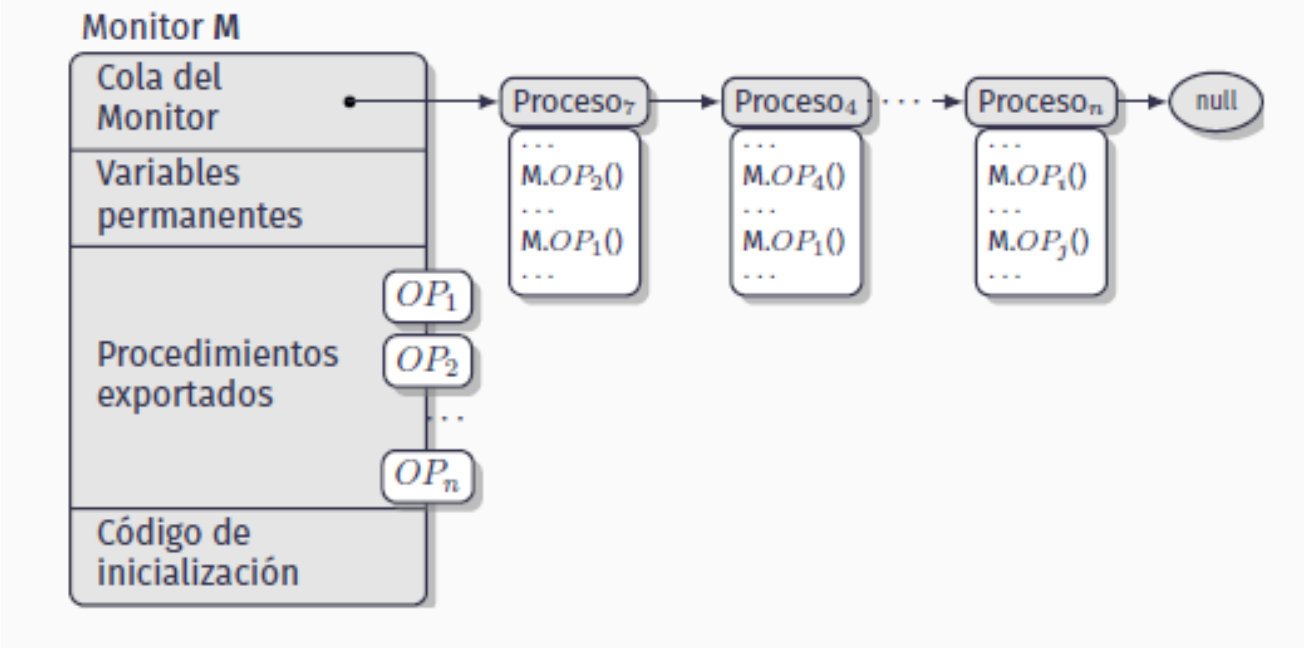


Figure: Posibles estados de los procesos y las transiciones entre dichos estados

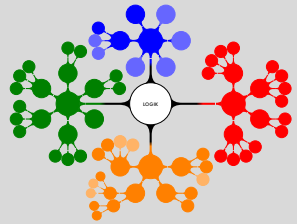
Si el monitor está libre en el momento de la llamada al procedimiento del monitor, no habrá espera en la cola del monitor



Estado del monitor



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



Operaciones de sincronización en monitores

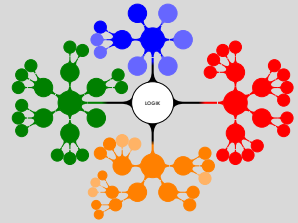
Para implementar la sincronización, se requiere de una facilidad para que los procesos hagan esperas bloqueadas, hasta que sea cierta determinada condición

En semáforos existe:

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

En monitores, sin embargo, se tiene:

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



Operaciones de sincronización en monitores

Encapsulación de la sincronización

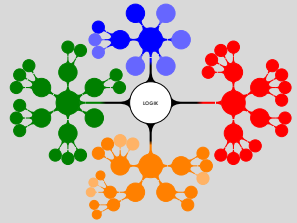
- Explícitamente programada dentro de los procedimientos
- TDA `cond`
- Variables condición: posibilitan la espera de condiciones diferentes dentro de un monitor
- Se define 1 variable por cada condición, dentro de los procedimientos

`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

- La representación de las variables condición no es accesible al programador de monitores

`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

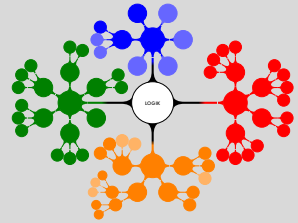
- La sincronización y la exclusión mutua –en el acceso a recursos que protege el monitor– se programa dentro del monitor con variables del TDA `cond` (soporte de señales)
- La exclusión mutua se levanta como consecuencia de ejecutar `c.wait()` $\rightarrow$ se evita el bloqueo del monitor
- Se cede el acceso al monitor al proceso señalado (`c.signal()` con semántica desplazante) $\rightarrow$ se evita el robo de señal

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

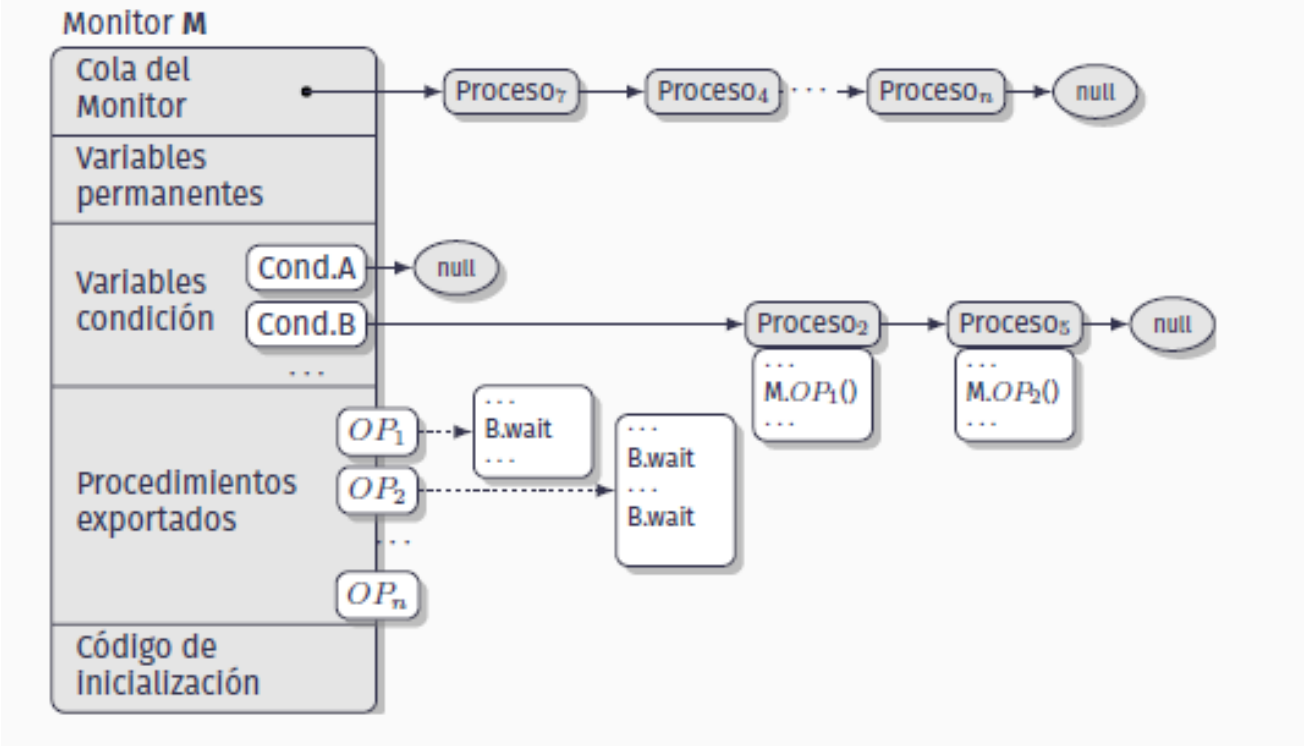
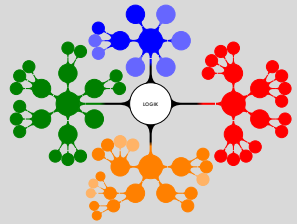


Figure: Los procesos 2 y 5 ejecutan las operaciones 1 y 2, ambas producen esperas de la condición B



Verificación de programas con monitores

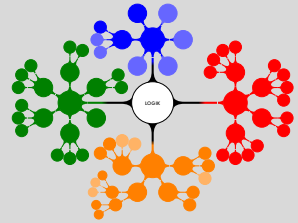
La verificación de la corrección de un programa concurrente con monitores supone:

- 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 traza concreta de llamadas a los procedimientos del monitor. El enfoque de verificación que vamos a seguir utiliza un invariante de monitor

Invariante de un monitor (IM):

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



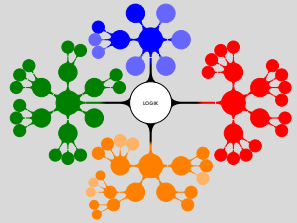
Verificación de programas con monitores-II

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
-Aplicar la regla de la concurrencia

Características de un 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 de verdad* depende de la traza del monitor y de los valores de las variables permanentes de dicho monitor
- Debe ser cierto en cualquier estado del programa concurrente, excepto cuando un proceso está ejecutando código del monitor, en E.M. (está en proceso de actualización de los valores de las variables permenentes)



Verificación de los monitores

- Axiomas iniciales

  La demostración de corrección se basa en:

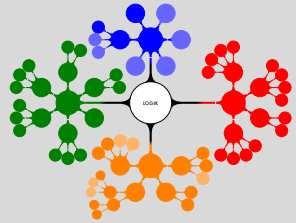
> **Invariante del monitor**(IM)
> relación constante entre los valores permitidos de las variables permanentes del monitor

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



Verificación de los monitores-II

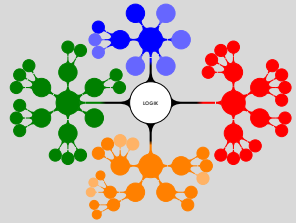
El invariante del monitor debe ser cierto:

- 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

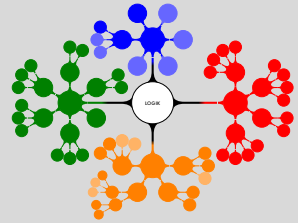
El invariante del monitor debe ser cierto:

- antes de cada operación `wait`
- después de cada operación `signal`
- Además, justo antes de una operación `signal` sobre una variable condición **c** debe ser cierta la condición lógica **C** asociada a dicha variable

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()`

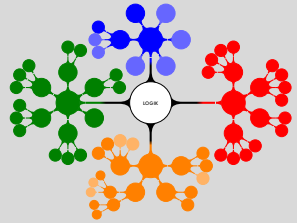


Axiomas de operaciones de sincronización con semántica desplazante-II

Axioma (operación `c.signal()`)

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

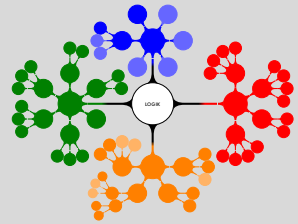
- No tiene efecto si la cola `c` está vacia
- Se supone semántica desplazante: el monitor mantiene el estado expresado por C hasta que un proceso bloqueado en `c` se reanude
- Los axiomas no tienen en cuenta el orden de desbloqueo de los procesos



Ejemplo: verificación de un monitor con señales desplazantes

```
Monitor Semaforo;
 var s: integer; {IM : 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

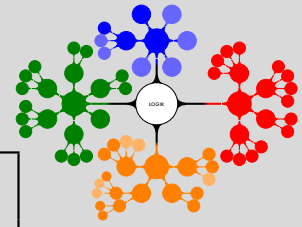
Ejemplo

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;
```

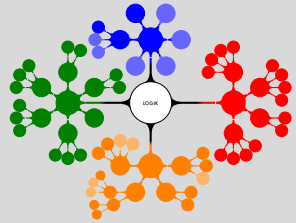



Regla de la concurrencia para la verificación de programas con monitores

$\{P_i\}\, S_i\, \{Q_i\},\ 1 \leq i \leq n$
ninguna variable libre en P_i o en Q_i es modificada por $S_j,\ i \neq j$
todas las variables en IM_k son locales al monitor m_k

$$\{IM_1 \wedge \ldots IM_m \wedge P_1 \wedge \ldots \wedge P_n\}$$
$$\text{cobegin } S_1 \parallel S_2 \parallel \ldots \parallel S_n \text{ coend}$$
$$\{IM_1 \wedge \ldots IM_m \wedge Q_1 \wedge \ldots \wedge Q_n\}$$

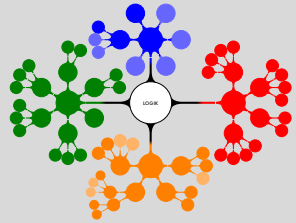
- La programación concurrente con monitores excluye cualquier interferencia entre las demostraciones de los procedimientos y las de los procesos secuenciales del programa
- Si los asertos de las demostraciones de los procesos no son críticos, entonces las demostraciones individuales $\{P_i\}S_i\{Q_i\}$ cooperan (no es necesario demostrar teoremas de no-interferencia)



Patrones de uso de monitores

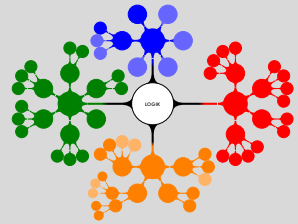
Se estudian a continuación los patrones de solución para tres problemas sencillos, típicos de la Programación Concurrente

- 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)



Monitor para Espera Única (EU)

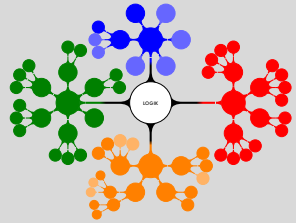
```
monitor EU;
var terminado:boolean;
    //true si se ha terminado E, si no: false
    lee: boolean //variable auxiliar
    cola:condition;
    //cola consumidor esperando terminado==true
    export esperar, notificar;
  //Invariante: terminado= false => lee= false
  procedure esperar();//para llamar antes de L
  begin
   if (not terminado) then //si no se ha terminado E
       cola.wait();//esperar hasta que termine
   //Condición de sincronización terminado= true
   lee:= true;
  end;
  procedure termina();
  begin lee:= false; terminado:= false; end;
  procedure notificar();//para llamar después de E
  begin
    terminado:=true;//Condición de sincronización
    cola.signal();//reactivar el otro proceso, si espera
  end
  begin { inicializacion: }
    terminado := false;//al inicio no ha terminado E }
    lee:= false;
  end;
```

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
```

- El proceso Consumidor espera, antes de leer, a que el Productor termine la sentencia de escritura
- De esta forma nos aseguramos que es posible un entrelazamiento **E**, **L**, pero no puede ocurrir el entrelazamiento **L**, **E**

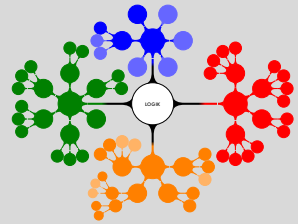


Verificación del monitor EU

Invariante del monitor : $terminado = false \Rightarrow lee = false$

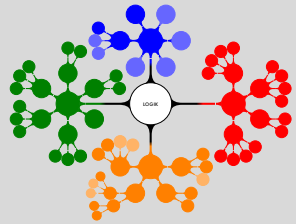
```
procedure esperar();
{Invariante, terminado= false, lee= false}
if (not terminado) then
  begin
    {terminado= false, lee= false, Invariante}
    cola.wait();
  endif;
  {Condicion de sincronizacion: terminado= true}
  lee:= true
  {Invariante}
 end;
```

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

Patrón de Exclusión Mutua (EM)

```
monitor EM ;
  var ocupada:boolean;
      //true hay un proceso en SC, si no: false
      cola:condition;
      //cola de procesos esperando ocupada==false
      export entrar,salir;
      //nombra procedimientos publicos
  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 uno
    cola.signal();
   end
  begin//inicializacion:
   ocupada:=false;//al inicio no hay procesos en SC
  end
```

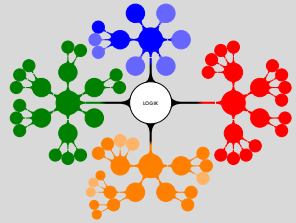



Uso y propiedades del monitor

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
    ..... //seccion critica
    EM.salir(); //registrar SC libre, señalar
    ..... //otras actividades (RS)
  end
end
```

- #E es el número de llamadas a entrar completadas
- #S es el número de llamadas a salir completadas
- El número de procesos en SC es $\#E - \#S$
- El único entrelazamiento correcto es **E, S, E, S,...**
- Se debe cumplir $0 \leq \#E - \#S \leq 1$



Invariante del monitor

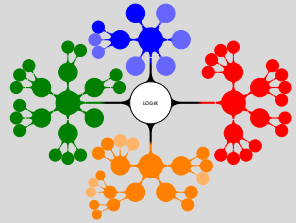
El invariante del monitor, es la conjunción de estas dos condiciones:

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

  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).



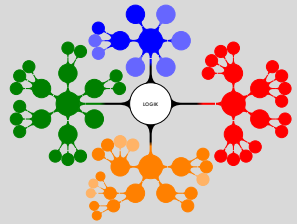
Corrección del programa

- Demostración del procedimiento *entrar()* modificado:

```
procedure entrar();
{ocupada= false, num_sc= 0 OR ocupada= true, num_sc=
    1}=>{IM}
if (ocupada) then
      cola.wait();{Condicion sincronizacion: ocupada
          == true}
    else num_sc:= num_sc + 1;
  endif;
{ocupada= true, num_sc= 1}=>{IM}
end;
```

- Demostración del procedimiento *salir()* modificado:

```
procedure salir();
begin {ocupada== true, num_sc== 1, IM}
  if cola.queue() then
    begin {Condicion de sincronizacion}
     ocupada:= true;
     cola.signal();
    end else begin
      ocupada:= false;
      num_sc:= 0;
    end;
end:
```

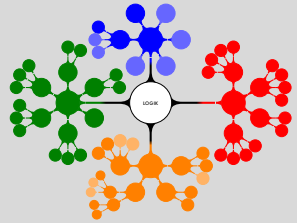



Sincronización tipo Productor/Consumidor

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

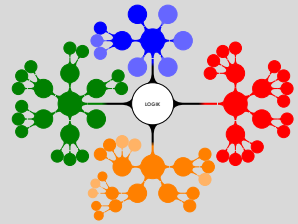
- El procedimiento `escribir` escribe el parámetro en la variable compartida
- El procedimiento `leer`, lee el valor que hay en la variable

```
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/Concumidor

```
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 );
   begin
    if pendiente then
      cola_prod.wait();
      valor_com:=v; //num_escrituras:= num_escrituras+1
      pendiente:=true;
      cola_cons.signal();
   end;
  function leer():integer;
   begin
   if (not pendiente) then
     cola_cons.wait();
     result:=valor_com;//num_lecturas:= num_lecturas+1
     pendiente:=false;
     cola_prod.signal();
   end;
   begin //inicialización }
     pendiente := false ;
   end;
```

Verificación del monitor

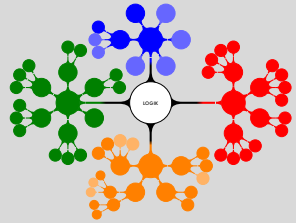
Al igual que en los otros casos, podemos verificar que el monitor funciona bien:

- #E = número de llamadas a escribir completadas
- #L = número de llamadas a leer completadas
- El monitor es correcto sólo si en cualquier estado: $0 \leq \#E - \#L \leq 1$

El invariante del monitor es:

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

- Condición sincronización del productor: pendiente== false; Condición sincronización del consumidor: pendiente== true;

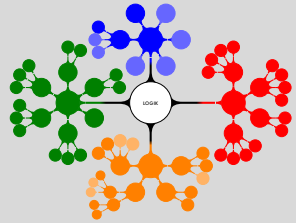


Demostración del invariante

Se cumplen las siguientes condiciones lógicas:

- Al inicio, se cumple el invariante, ya que #E y #L son 0, y `pendiente` es `false`
- Si se cumple el invariante y se ejecuta `escribir`:

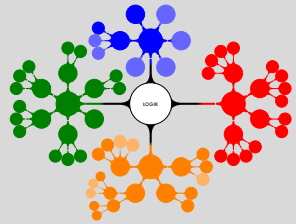
```
 procedure escribir( v:integer );
   begin
{pendiente==false, num_escrituras==num_lecturas OR
  pendiente==true, num_escrituras==num_lecturas+1}=>
      Invariante
    if pendiente then
      cola_prod.wait();
      {Condicion sincronizacion: pendiente= false}
    valor_com:=v; //num_escrituras:= num_escrituras+1
    pendiente:=true;
    {Condicion sincronizacion del consumidor}
    if cola_cons.queue() then cola_cons.signal();
{ Invariante }
  end;
```

Demostración del invariante-II

- Si se cumple el invariante y se ejecuta `leer`:

```
function leer():integer;
{pendiente==false, num_escrituras==num_lecturas OR
   pendiente==true, num_escrituras==num_lecturas+1}=>
        Invariante
  begin
   if (not pendiente) then
     cola_cons.wait();
     {Condicion sincronizacion: pendiente= true}
   result:=valor_com;//num_lecturas:= num_lecturas+1
   pendiente:=false ;
   {Condicion sincronizacion del productor}
   if cola_prod.queue() then cola_prod.signal();
{Invariante}
  end;
```

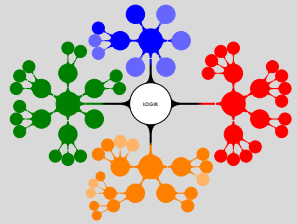
Señales con prioridad

- 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

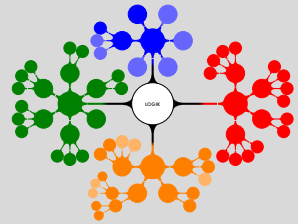
Despertador que recuerda los tiempos indicados por sus usuarios. Varios de ellos pueden indicar la misma hora

```
procedure tick();
  begin
    ahora:= ahora +1;
    despertar.signal();
  end;
begin
  ahora:= 0;
end;
```

Señales con prioridad-II

```
monitor despertador;
  var
    ahora: integer;
    despertar: {\bf cond}; --prioritaria

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

Señales con prioridad-III

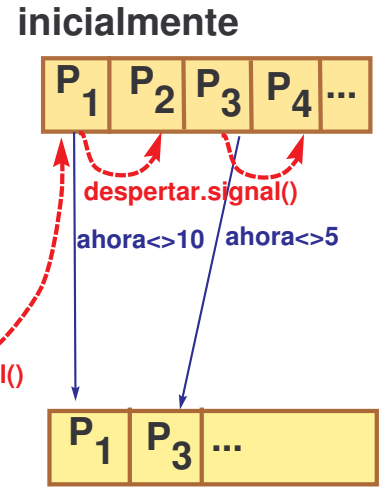
- Simulación mediante señales FIFO con semántica desplazante

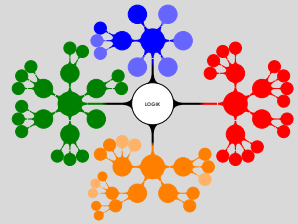
```
Monitor despertador[d:tipo_enumerado]
 var
  ahora:int;
  despertar:cond;
 procedure despiertame(n:int)
  begin
   alarma:= ahora + n;
   while(ahora< alarma) do
    begin
     despertar.signal();
     despertar.wait();
    end;
   despertar.signal;
  end;
 procedure tick
   begin
    ahora:= ahora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

Escenario de ejecucion

id	instruccion	tiempo
P1	despiertame(10)	0
P2	despiertame(3)	1
P3	despiertame(5)	2
P4	despiertame(3)	3 ahora=3

despertar.signal()

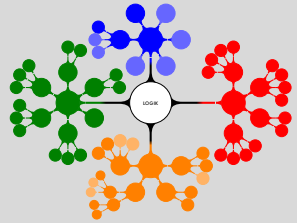




El problema de Lectores/Escritores (LE)

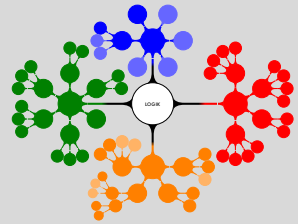
Dos tipos de procesos acceden concurrentemente a datos compartidos:

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



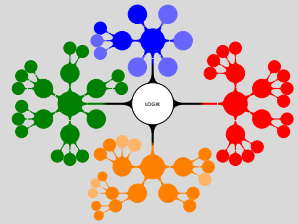
Variables permanentes y procedimientos para lectores

```
monitor Lec_Esc ;
var n_lec:integer;//numero de lectores leyendo
    escrib:boolean;
    //true si hay algun escritor escribiendo
    lectura:condition;
    //no hay escrit. escribiendo, lectura posible
    escritura:condition;
    //no hay lect. ni escrit., escritura posible
    export ini_lectura, fin_lectura,
    //invocados por lectores
    ini_escritura, fin_escritura;
    //invocados por escritores
procedure ini_lectura()
begin
  if escrib then//si hay escritor:
    lectura.wait();//esperar
//registrar un lector más
  n_lec := n_lec + 1 ;
//desbloqueo en cadena de
//posibles lectores bloqueados
  lectura.signal()
end
```

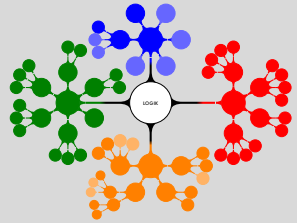
Variables permanentes y procedimientos para lectores-II

```
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 escribiendo then
      escritura.wait()
     //registrar que hay un escritor
    escribiendo:= true;
  end;
procedure fin_escritura()
  begin//registrar que ya no hay escritor
    escribiendo := false;
//si hay lectores, despertar uno; si no hay, a un escritor
   if lectura.queue() then
     lectura.signal();
    else
      escritura.signal() ;
  end;
begin//inicializacion
 n_lec := 0; escribiendo:= false ;
end
```

Uso del monitor-II

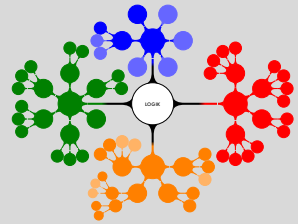
- En esta implementación se ha dado prioridad a los lectores (en el momento que un escritor termina, si hay escritores y lectores esperando, pasan los lectores)
- Hay otras opciones:
  - prioridad a escritores,
  - prioridad al que más tiempo lleva esperando



Uso del monitor

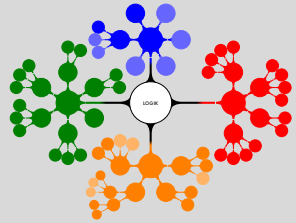
Los procesos lectores y escritores usan el monitor de esta forma:

```
process Lector[ i:1..n ] ;
begin
while true do begin
..........
Lec_Esc.ini_lectura() ;
//codigo de lectura: ....
Lec_Esc.fin_lectura() ;
..........
end
end
process Escritor[ i:1..m ] ;
begin
while true do begin
..........
Lec_Esc.ini_escritura() ;
//codigo de escritura: ....
Lec_Esc.fin_escritura() ;
..........
end
end
```

Todos los mecanismos de señalación alternativos de los monitores

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



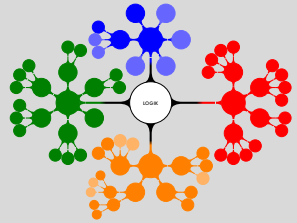
Posibles semánticas de las señales de los monitores

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

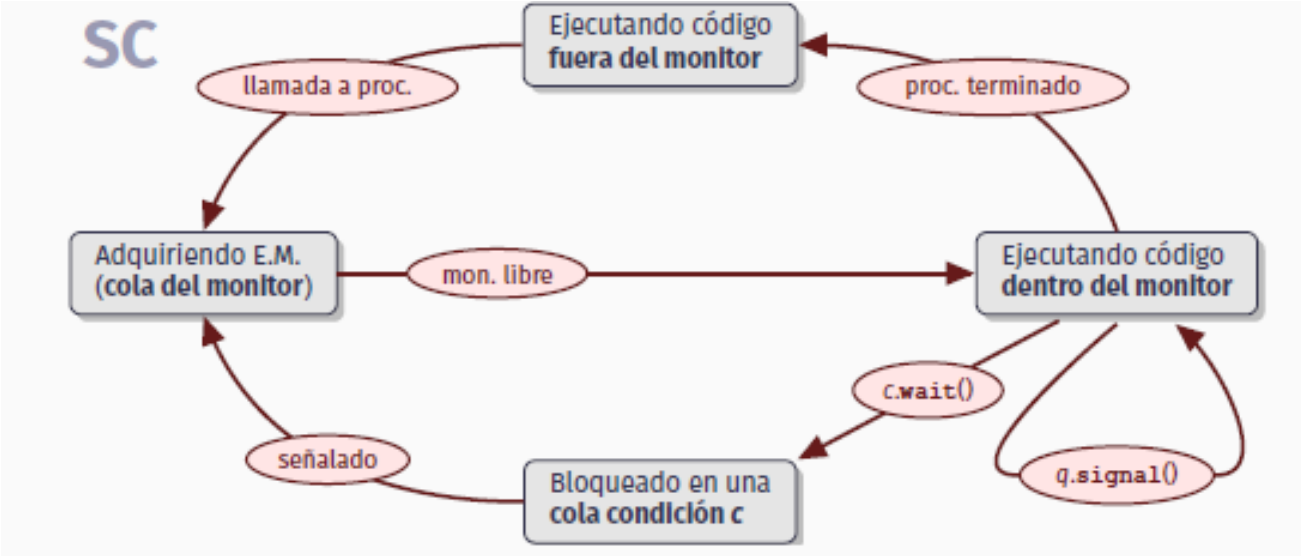
El proceso señalado espera bloqueado hasta que puede adquirir la E.M. de nuevo, semántica de señal:*SC: señalar y continuar*

El proceso señalado se reactiva inmediatamente

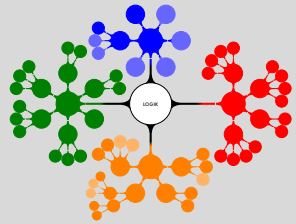
2. El proceso señalador abandona el monitor tras hacer `signal`, sin ejecutar el código que haya después de dicho `signal`, semática de señal:*SS: señalar y salir*
   - El señalador queda bloqueado a la espera en la cola del monitor, junto con otros posibles procesos que quieren comenzar a ejecutar código del monitor, semántica de señal: *SE: señalar y esperar*
   - El señalador se queda bloqueado en una cola específica, dicha cola mantiene a estos procesos con mayor prioridad para volver a entrar al monitor:*SU: señalar y espera urgente*



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`

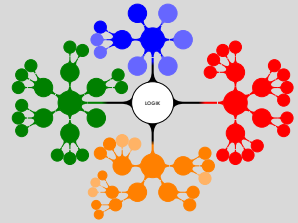


Señalar y continuar (SC): características

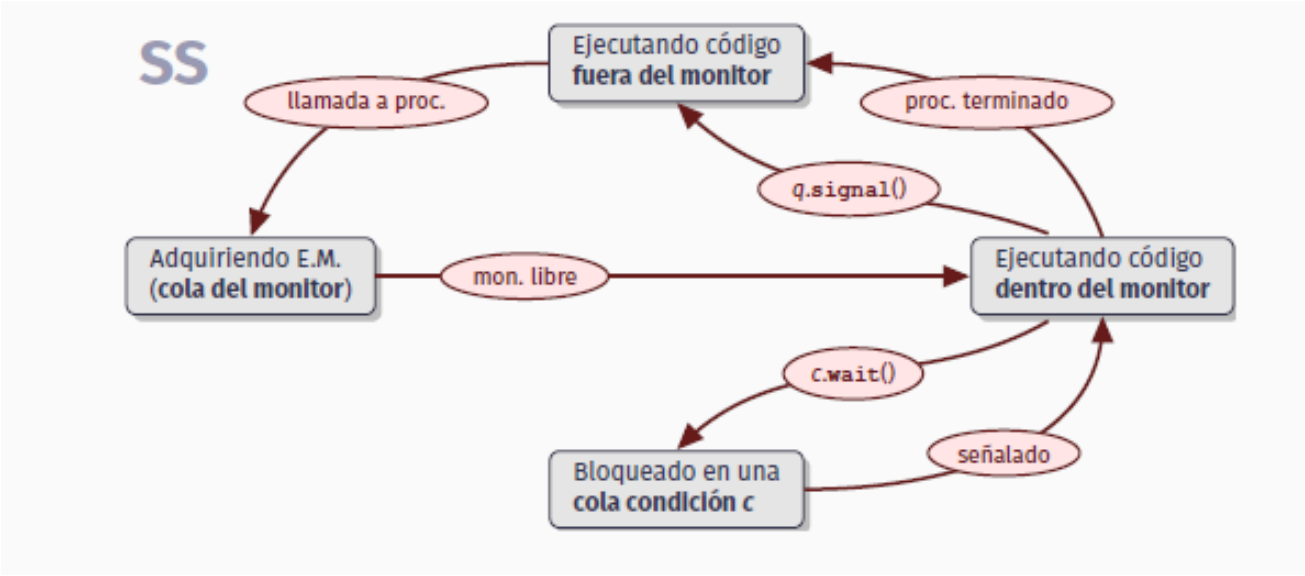
El proceso señalador continúa su ejecución dentro del monitor después del signal

- 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, 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:

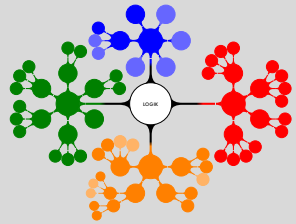
```
while not {\bf cond}icion_lógica_desbloqueo do
  {\bf cond}.wait() ;
```

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



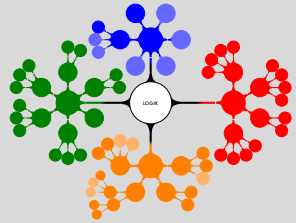
- Señalador: abandona el monitor (ejecuta código tras la llamada al procedimiento del monitor). Si hay código en este proceso, tras ejecutar `signal`, no se ejecuta inmediatamente
- Señalado: reanuda inmediatamente la ejecución del código del procedimiento del monitor, programado tras la operación `wait`



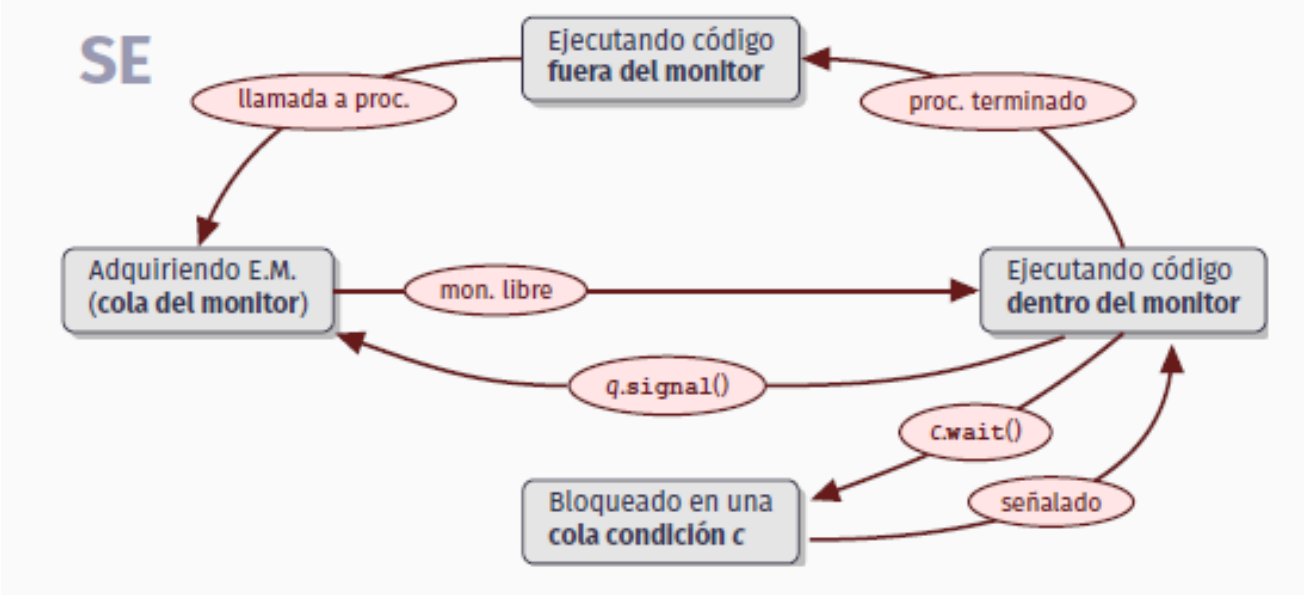
Señalar y salir (SS): características

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

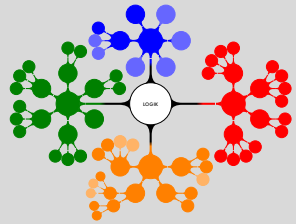
- Si hay código tras signal, no se ejecuta. El proceso señalado reanuda inmediatamente la ejecución de código del monitor.
- En ese caso, la operación signal conlleva:
  1. Liberar al proceso señalado
  2. Terminación del procedimiento del monitor que estaba ejecutando el proceso señalador
  3. Está 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)
  4. Esta semántica condiciona el estilo de programación ya que obliga a colocar siempre la operación signal como última instrucción de los procedimientos de monitor que la usen



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



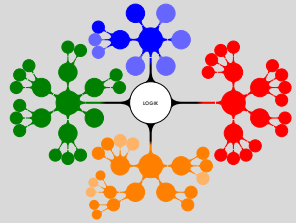
- Señalador: se bloquea en la cola del monitor hasta readquirir E.M. y ejecutar el código del monitor tras signal
- Señalado: reanuda inmediatamente la ejecución de código del monitor tras wait



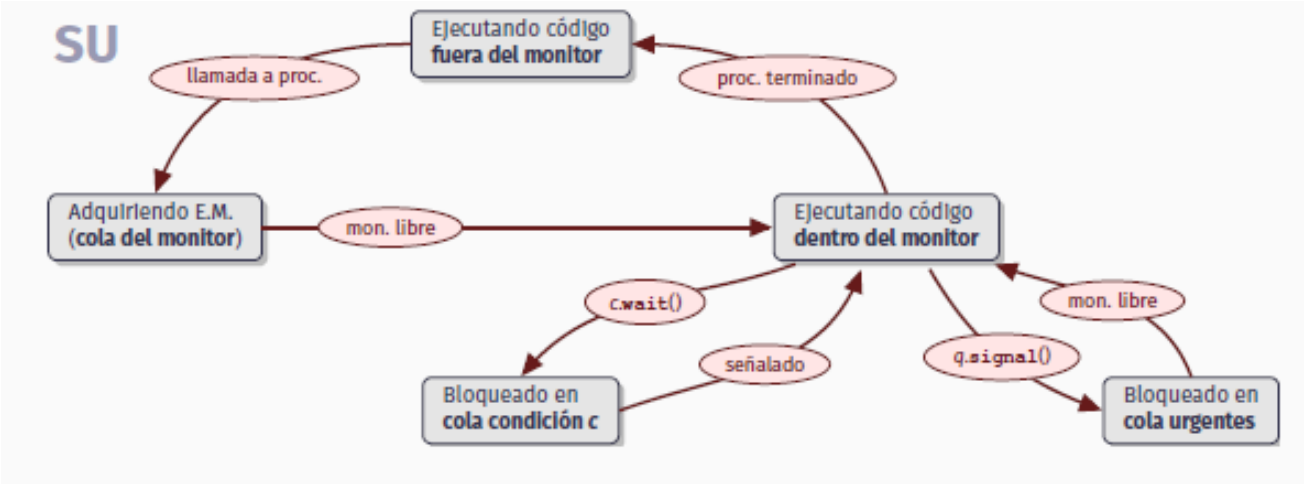
Señalar y esperar (SE): características

El proceso señalador se bloquea en la cola del monitor justo después de ejecutar signal

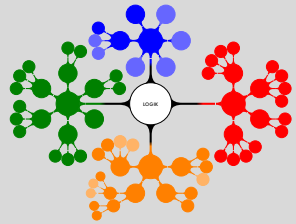
- El proceso señalado entra de forma inmediata en el monitor
- Está asegurado el estado que permite al proceso señalado continuar la ejecución del procedimiento del monitor en el que se bloqueó
  1. El proceso señalador entra en la cola de procesos del monitor, por lo que está al mismo nivel que el resto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



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



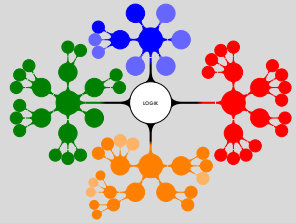
- Señalador: se bloquea en la cola de urgentes hasta readquirir la E.M. y ejecutar código del monitor tras signal. Para readquirir E.M. tiene más prioridad que los procesos en la cola del monitor
- Señalado: reanuda inmediatamente la ejecución de código del monitor tras la operación `wait`



Señalar y espera urgente (SU): características

Es similar la semántica SE, pero se intenta corregir el problema de falta de equitatividad de las señales SS

- El proceso señalador se bloquea justo después de ejecutar la operación signal
  1. El proceso señalado entra de forma inmediata en el monitor
  2. Está asegurado el estado que permite al proceso señalado continuar la ejecución del procedimiento del monitor en el que se bloqueó
  3. El proceso señalador entra en una nueva cola de procesos que esperan para acceder al monitor, que podemos llamar cola de procesos urgentes
  4. 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



Procesos en la cola de urgentes

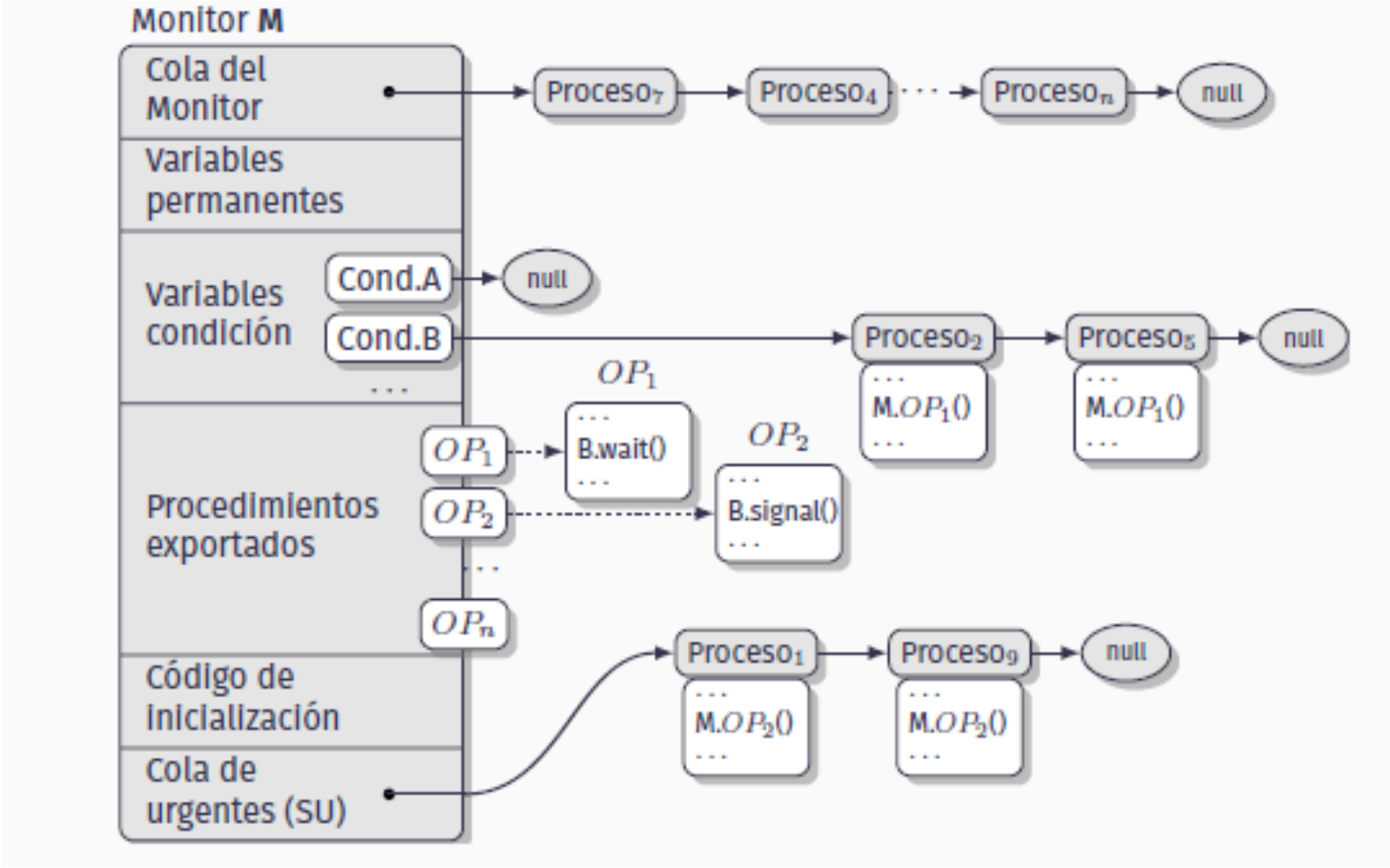
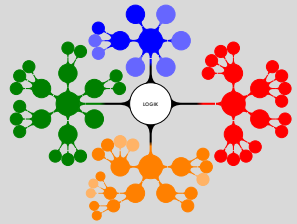


Figure: El proceso 1 y el 9 han ejecutado la op.2, que hace signal de la **cond**. B.



Análisis comparativo de las diferentes semánticas de señales

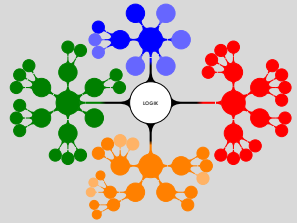
Potencia expresiva: todas las semánticas son capaces de resolver los mismos problemas

Facilidad de uso:

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

Eficiencia:

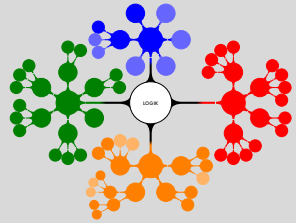
- Las semánticas SE y SU resultan ineficientes cuando no hay código tras `signal`, ya que en ese caso implican que el señalador emplea tiempo en bloquearse y después reactivarse, pero justo a continuación abandona el monitor sin hacer nada
- La semántica SC también es un poco ineficiente al obligar a usar un bucle para cada instrucción `signal`



Comparativa entre las distintas semánticas de señales mediante un ejemplo

Barrera parcial

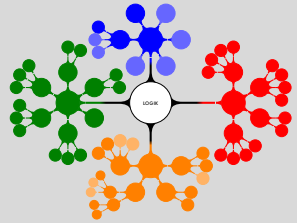
- El monitor tiene un único procedimiento público llamado *cita*
- Hay p procesos ejecutando un bucle infinito, en cada iteración realizan una actividad de duración arbtraria y después llaman a *cita*
- Ningún proceso termina *cita* antes de que haya al menos n de ellos que la hayan inciado (donde $1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



Diseño del monitor *barrera parcial*

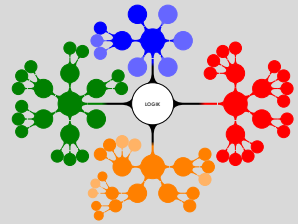
Cuando un proceso comienza a ejecutar cita, debe esperar hasta que haya otros `n - 1` que tambien lo hayan hecho:

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



Una posible implementación del monitor

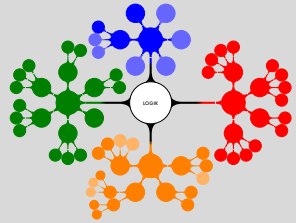
```
Monitor BP //monitor Barrera Parcial }
  var cola : {\bf cond}ition;//procesos esperando contador
      ==n
      contador : integer;//numero de procesos ejecutando
          cita
procedure cita() ;
  begin
   contador := contador+1;//registrar un proceso mas
       ejecutando cita
   if (contador<n) then
       cola.wait();//esperar a que haya n procesos
           ejecutando
      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:

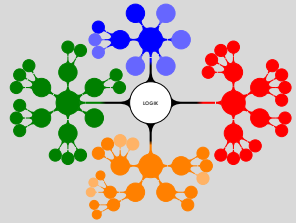
- *Señalar y Continuar:* el último proceso ejecuta todos los `signal` seguidos sin esperar entre ellos, y después pone el `contador` a 0. Los n - 1 procesos señalados abandonan el `wait`, pero pasan a la cola del monitor. Por tanto, antes de que esos señalados puedan terminar de ejecutar cita, todos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



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

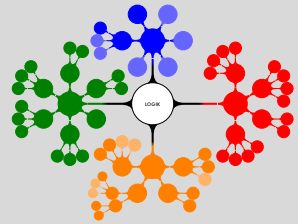
...

- *Señalar y Esperar:* el último proceso, después de ejecutar el primer `signal`, va a la cola del monitor. Por tanto, podría entrar a la cita el primer proceso del siguiente grupo y observar `contador==n`, con lo cual este primer proceso no haría la espera requerida
- *Señalar y Espera Urgente:* el último proceso ejecuta todos los `signals`. Entre cada dos de ellos, espera en la *cola de urgentes* a que el señalado abandone el monitor. Los procesos del siguiente grupo esperan en la cola del monitor hasta que todos los señalados lo abandonen y el último ejecute la asignación `contador:=0`. 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;
```

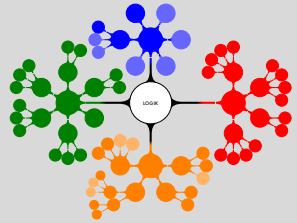



Comportamiento de la versión alternativa

Analizamos el comportamiento en todas las semánticas:

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

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



Reglas de verificación de las señales no-desplazantes SC

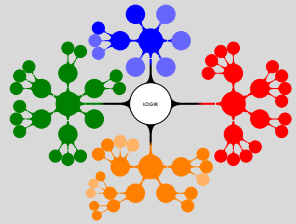
Axioma de la operación `c.wait()`

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

- 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()`

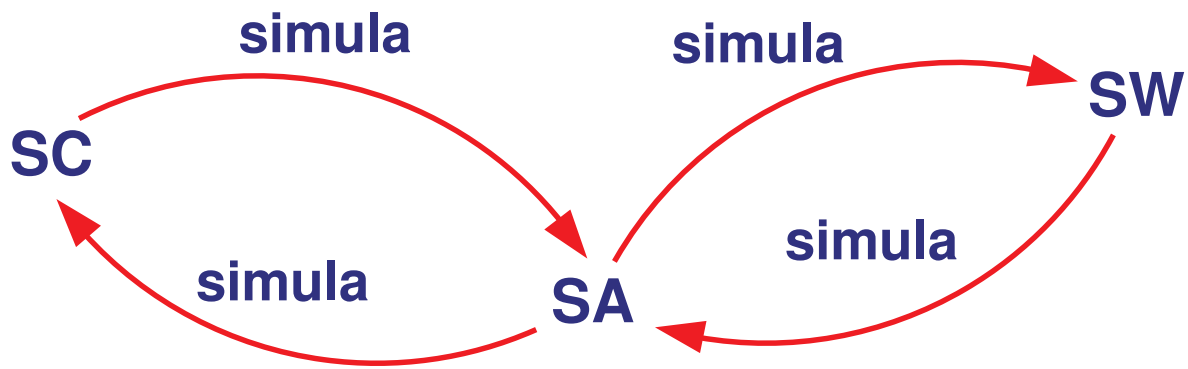
$$\{\mathtt{P}\}\ \mathtt{c.signal()}\ \{\mathtt{P}\}$$

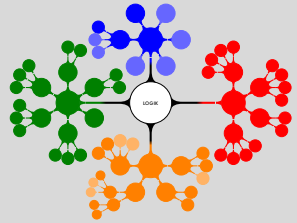


Intercambio de señales en programas que usan monitores

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

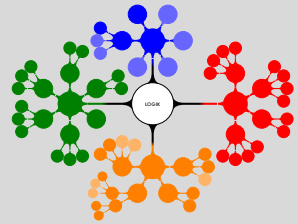




Implementaciones alternativas de un semáforo con monitores

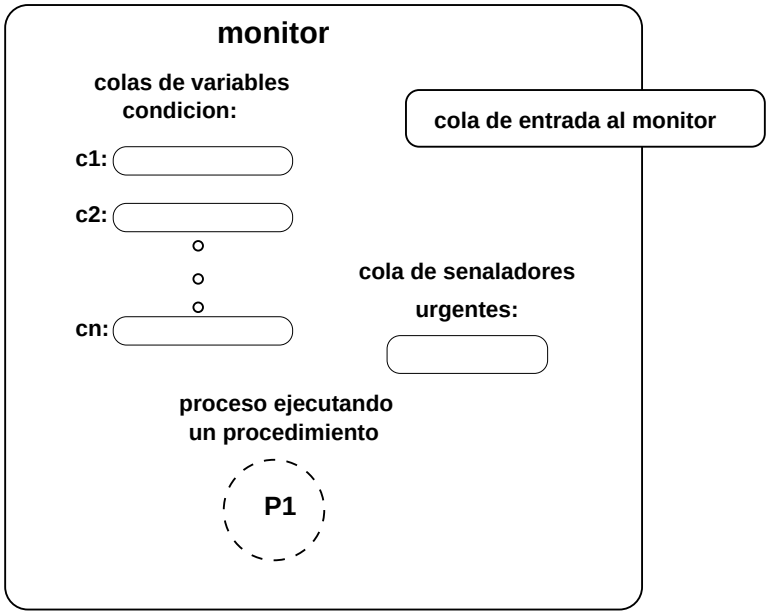
```
Monitor semaforo_FIFO1;
 {IM: s>=0}
var c: cond;
    s: int;
procedure P;
 begin
  if(s=0) then
      c.wait();
      {s > 0}
   s:=s-1;
 end;

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

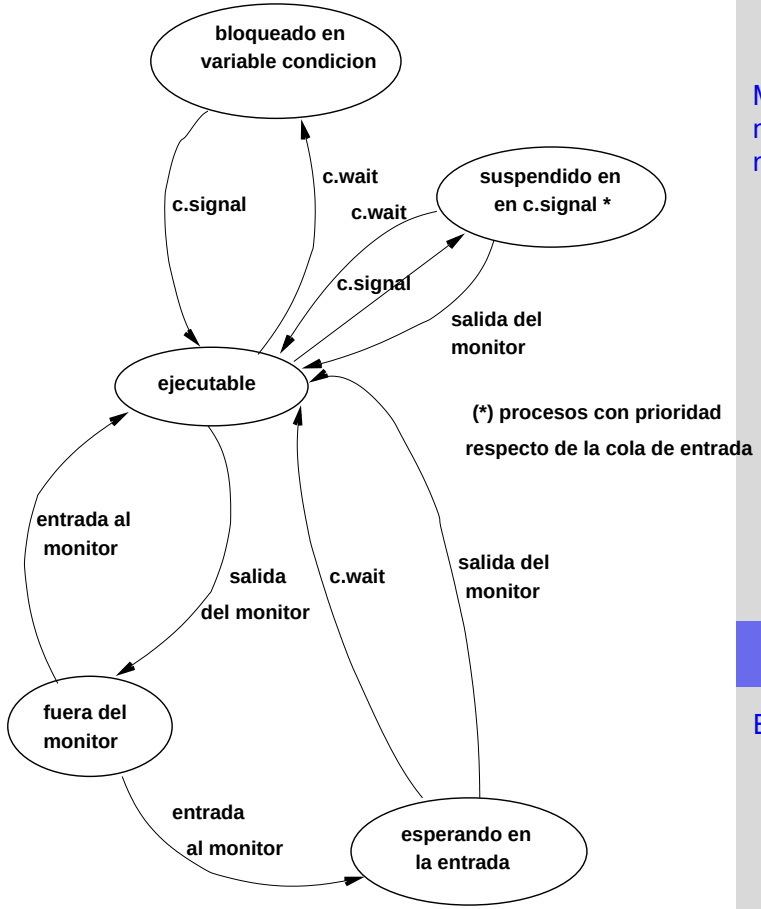
```
Monitor semaforo_FIFO2;
{IM: s>=0}
var c: cond;
    s: int;
procedure P;
 begin
  while(s=0) do
      c.wait();
      {s>= 0}
   end do;
   {s > 0}
   s:=s-1;
 end;

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

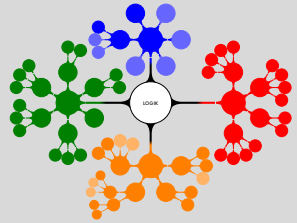



Implementación de los monitores

Esquema de implementación de un monitor con señales SU



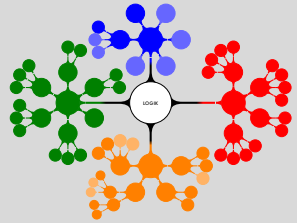




Implementación de los monitores con semáforos

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)

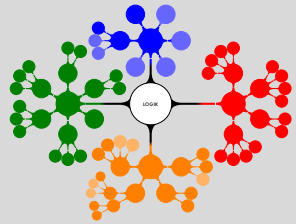


Implementación de los monitores con semáforos-II

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 ;
```

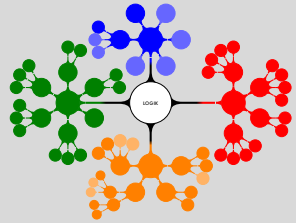



Implementación de los monitores con semáforos-III

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 ;
```

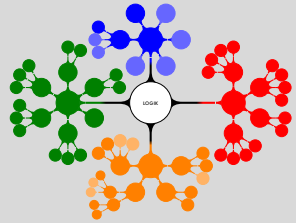



Implementación de los monitores con semáforos-IV

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
}
```

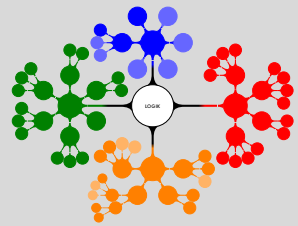
```
void x_wait(Semaphore x_sem, unsigned x_sem_count){
//implementacion de x.wait()
x_sem_count := x_sem_count + 1 ;//cuenta 1 proceso mas
    bloqueado en la cola de condicion "x"
if (next_count <> 0) then//hay procesos señaladores
    esperando
    sem_signal(next); //desbloquea un proceso señalador
   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

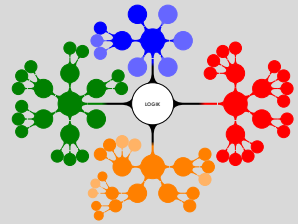
```
void x_signal(Semaphore x_sem, unsigned x_sem_count){
//implementacion de x.signal()
if (x_sem_count <> 0) then
  begin //hay procesos bloqueados esperando la condicion "
      x"
    next_count := next_count + 1;//cuenta 1 proceso mas en
        la cola de señaladores
    sem_signal(x_sem); //desbloquea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

Introducción al problema del “exclusión mutua”

Introducción histórica al problema de la exclusión mutua

1962 Dekker propone el problema de la exclusion mutua para multiprocesadores:

"disenar un protocolo que garantice el acceso mutuamente excluyente, sin que exista interbloqueo, a una seccion critica por parte de un determinado numero de procesos que compiten por entrar a dicha seccion..."

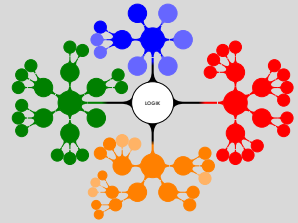
1965 Dijkstra propone una solucion segura, libre de interbloqueo, pero que puede producir inanicion

1966 Knuth propone una solucion sin inanicion; garantiza retraso limitado de los procesos, pero no FIFO

1974 Lamport: permite a los procesos "detenerse" en la ejecucion del protocolo de adquisicion, solapar las operaciones de lectura con la escritura y retraso FIFO de los procesos que ya esperan entrar

1981 Peterson propone una solucion equitativa para 2 y "n" procesos; garantiza el retraso cuadratico de los procesos, es la solucion mas simple hasta fecha para multiprocesadores

1983 Algoritmos totalmente distribuidos que resuelven el problema para multicomputadores; Ricart-Aggrawala, Suzuki-Kasami

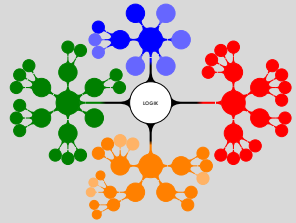


Solución al problema con bucles de espera activa

- Los procesos iteran en un bucle vacío hasta que la entrada en Sección Crítica (SC) sea segura
- Aceptable si el sistema/aplicación no tuviera muchos procesos

Condiciones de Dijkstra para obtener una solución *parcialmente correcta* al problema de exclusión mutua:

1. No hacer ninguna suposición acerca de las instrucciones o número de procesos soportados por el multiprocesador
2. Ni tampoco acerca de la velocidad de ejecución de los procesos, excepto que no es cero (*Progreso Finito*)
3. Cuando un proceso se encuentra ejecutando código fuera de la sección crítica no puede impedir a los otros procesos entrar en ésta
4. La sección crítica siempre será alcanzada por alguno de los procesos que esperan entrar



Método de refinamiento sucesivo

Esquema de "corrutinas": no se cumple la tercera propiedad de Dijkstra!

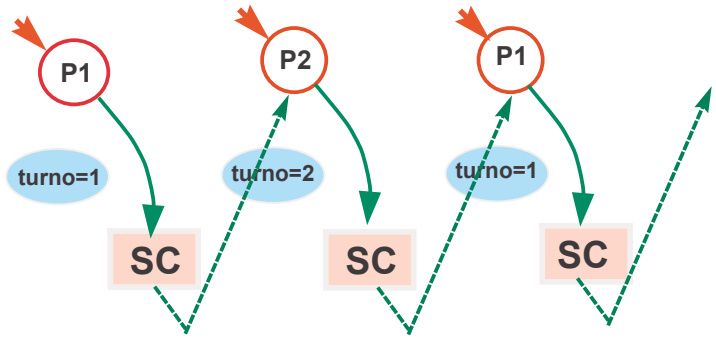
1A Etapa

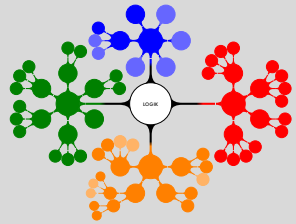
Proceso P1

```
while true do
 begin
  <<resto instrucciones>>

  while turno <> 1 do
    nothing;
  enddo;

  <<seccion critica>>
  turno:= 2;
 end
enddo;
```

Proceso P2

```
while true do
 begin
  <<resto instrucciones>>

  while turno <> 2 do
    nothing;
  enddo;

  <<seccion critica>>
  turno:= 1;
 end
enddo;
```





Método de refinamiento sucesivo –II

2A Etapa

Proceso P1

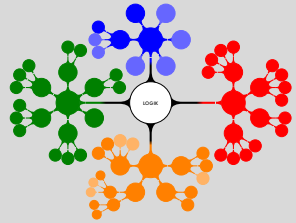
```
while true do
 begin
  <<resto instrucciones>>
  while c2=0 do
    nothing;
  enddo;
  c1:= 0;
  <<seccion critica>>
  c1:= 1;
 end
enddo;
```

Proceso P2

```
while true do
 begin
  <<resto instrucciones>>
  while c1= 0 do
    nothing;
  enddo;
  c2:= 0;
  <<seccion critica>>
  c2:= 1;
 end
enddo;
```

La salida de la espera activa y el cambio de la clave no se realizan atomicamente

Problema!: no se cumple la propiedad de seguridad del algoritmo.



Método de refinamiento sucesivo –III

Adelantando la asignacion de la clave la solucion es segura

3A Etapa

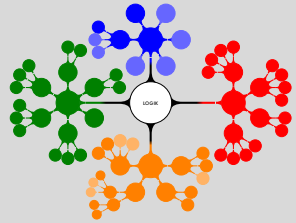
Proceso P1

```
while true do
 begin
  <<resto instrucciones>>
  c1:= 0;

  while c2=0 do
    nothing;
  enddo;

  <<seccion critica>>
  c1:= 1;

 end
enddo;
```

Proceso P2

```
while true do
 begin
  <<resto instrucciones>>
  c2:= 0;

  while c1= 0 do
    nothing;
  enddo;

  <<seccion critica>>
  c2:= 1;
 end
enddo;
```

Nuevo problema!: el proceso que modifica la clave no sabe si el otro hace lo mismo concurrentemente con el



Método de refinamiento sucesivo –IV

4A Etapa:

Proceso P1

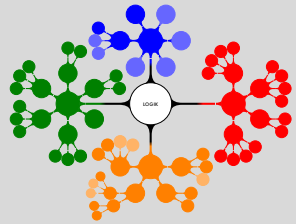
```
while true do
 begin
  <<resto instrucciones>>
  c1:= 0;

  while c2=0 do
     begin
      c1:= 1;
      while c2= 2 do
        nothing;
       enddo;
      c1:= 0;
     end
  enddo;
  <<seccion critica>>
  c1:= 1;
 end
enddo;
```

Proceso P2

```
while true do
 begin
  <<resto instrucciones>>
  c2:= 0;

  while c1= 0 do
     begin
      c2:= 1;
      while c1= 1 do
        nothing;
      enddo;
      c2:= 0;
     end
  enddo;
  <<seccion critica>>
  c2:= 1;
 end
enddo;
```

Para indicar que intenta entrar en S.C., cambia su clave

Comprueba la clave del otro; la vuelve a cambiar si el otro tambien intenta entrar,



Método de refinamiento sucesivo –V

5A Etapa: Algoritmo de Dekker

Proceso P1

```
while true do
 begin
  <<resto instrucciones>>
  c1:= 0;
 while c2= 0 do
   if turno= 2 then
    begin
     c1:= 1;
     while turno= 2 do
       nothing;
     enddo;
     c1:= 0;
    end
   endif
  enddo;
  <<seccion critica>>
  turno:= 2;
  c1:= 1;
 end
enddo;
```

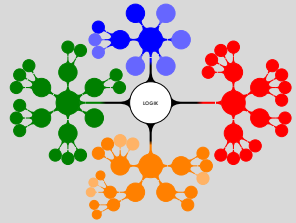
El proceso intenta entrar en S.C.

comprueba la clave del otro

si no tiene el turno hace espera activa, despues de cambiar su clave

Proceso P2

```
while true do
 begin
  <<resto instrucciones>>
  c2:= 0;
 while c1= 0 do
   if turno= 1 then
    begin
     c2:= 1;
     while turno= 1 do
       nothing;
     enddo;
     c2:= 0;
    end
   endif
  enddo;
  <<seccion critica>>
  turno:= 1;
  c2:= 1;
 end
enddo;
```

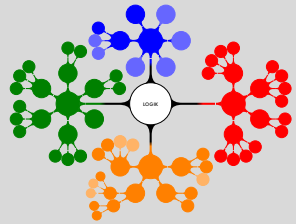
Método de refinamiento sucesivo: verificación de propiedades

Verificación de propiedades de seguridad

Exclusión mutua:

1. P_i entra en sección crítica sólo si $c[j] == 1$
2. P_i comprueba la clave del otro, $c[j]$, sólo después de asignar su propia clave

   Luego, cuando P_i entra se cumple $c[j] == 1 \wedge c[i] == 0$



Método de refinamiento sucesivo: verificación de propiedades –II

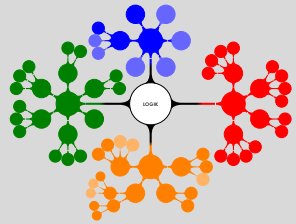
Verificación de propiedades de seguridad

Alcanzabilidad de la sección crítica:

Si P_i y P_j intentan entrar en sección crítica y `turno == i`:

1. si P_i encuentra la clave del otro c[j] == 1, entonces P_i entra;
2. si no, dependerá de quien tenga el turno:
   1. si `turno == i` espera que P_j cambie su clave y, después, entra
   2. si `turno == j` cambia su clave a 1 y se queda en espera activa

Discusión sobre la *equidad* de la solución dada por el Algoritmo de Dekker



Generalización a N procesos: Algoritmo de Dijkstra

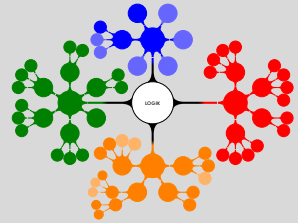
```
c: array[0..n-1] of (pasivo,solicitando,en_SC)
turno: 0.. n-1;

repeat
  repeat
    E2:c[i]:= solicitando;

    while turno <> i do
    E3:if c[turno]= pasivo
         then turno:= i
       endif;
    enddo;

    E4:c[i]:= en_SC;
    j:= 0;

    while(j<n) and (j=i or c[j] <> en_SC) do
      j:= j+1;
    enddo;
  until j>= n;

    <<Seccion Critica>>
E1:c[i]:= pasivo;
    <<Resto de instrucciones>>
until false
```

1A barrera detiene a los procesos si el que posee el turno no esta pasivo

La comprobacion del estado del que tiene el turno y el cambio de este no se hace atomicamente

Si un grupo de procesos se ve obligado a ciclar nuevamente, el que posee el turno no puede estar pasivo

2A barrera asegura que se cumple la propiedad de seguridad



Verificación de las propiedades del A. Dijkstra

Verificación de las propiedades de seguridad

Exclusión mutua:

demostración similar a la del A. Dekker

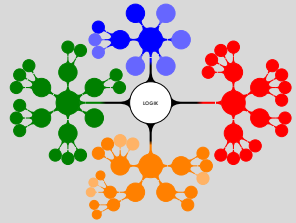
Alcanzabilidad de la sección crítica:

1. `turno` es una variable compartida, será asignada por el último P_i que cambie su clave, `c[j] == en_SC`

2. Sean $\{P_l \ldots P_i \ldots P_m\}$ tales que `c[i] = en_SC` y `turno ==k` con $l \leq k \leq m$, entonces

   P_k entrará en su sección en tiempo finito y el resto

   $P_i : l \leq i \leq m \wedge i \neq k$ se quedará ciclando



Verificación de las propiedades de vivacidad del A. Dijkstra

El A. Dijkstra satisface seguridad pero no evita el peligro de inanición de los procesos del programa

posición/acción	c[1]	c[2]	c[3]	turno
Inic.:P_1, P_2, P_3 en E1	pasivo	pasivo	pasivo	3
$P_1 : E_1 \rightarrow E_2$	solicitando	pasivo	pasivo	3
$P_2 : E_1 \rightarrow E_2$	solicitando	solicitando	pasivo	3
$P_1 : E_2 \rightarrow E_3$	solicitando	solicitando	pasivo	3
$P_2 : E_2 \rightarrow E_3$	solicitando	solicitando	pasivo	3
$P_1 : E_3 \rightarrow E_4$	en_SC	solicitando	pasivo	3
$P_2 : E_3 \rightarrow E_4$	en_SC	en_SC	pasivo	3
...	...	...	...	...

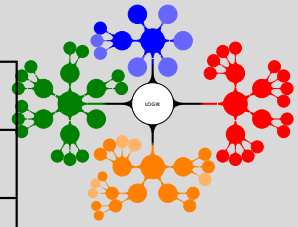
```
c: array[0..n-1] of (pasivo,solicitando,en_SC)
turno: 0.. n-1;
repeat
  repeat
    E2:c[i]:= solicitando;

    while turno <> i do
    E3:if c[turno]= pasivo
         then turno:= i
       endif;
    enddo;

    E4:c[i]:= en_SC;
    j:= 0;

    while(j<n) and (j=i or c[j] <> en_SC) do
      j:= j+1;
    enddo;
  until j>= n;

    <<Seccion Critica>>
E1:c[i]:= pasivo;
    <<Resto de instrucciones>>
until false
```

1A barrera detiene a los procesos si el que posee el turno no esta pasivo

La comprobacion del estado del que tiene el turno y el cambio de este no se hace atomicamente

Si un grupo de procesos se ve obligado a ciclar nuevamente, el que posee el turno no puede estar pasivo

2A barrera asegura que se cumple la propiedad de seguridad



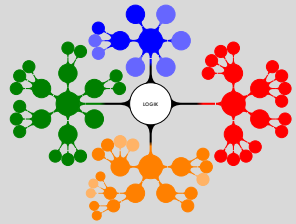
Algoritmo de Knuth para N procesos

```
                c: array[0..n-1] of (pasivo,solicitando,en_SC)
                turno: 0.. n-1;
repeat
  repeat
   E0: c[i]:= solicitando;
       j:= turno; --variable local
   E1: while j <> i do
        if c[j] <> pasivo then j:= turno
          else j:= (j-1) MOD n
        endif;
       enddo;
   E2: c[i]:= en_SC;
       k:= 0
   while (k<n) and (k=i pr c[k]<>en_SC) do
     k:= k+1;
   enddo;
  until k>= n;
   E3: turno:= i;
   << Seccion Critica >>
       turno:= (i-1) MOD n;
   E4: c[i]:= pasivo;
   E5: <<resto de instrucciones>>
until false;
```

1A barrera detiene a los procesos si el que posee el turno no esta pasivo

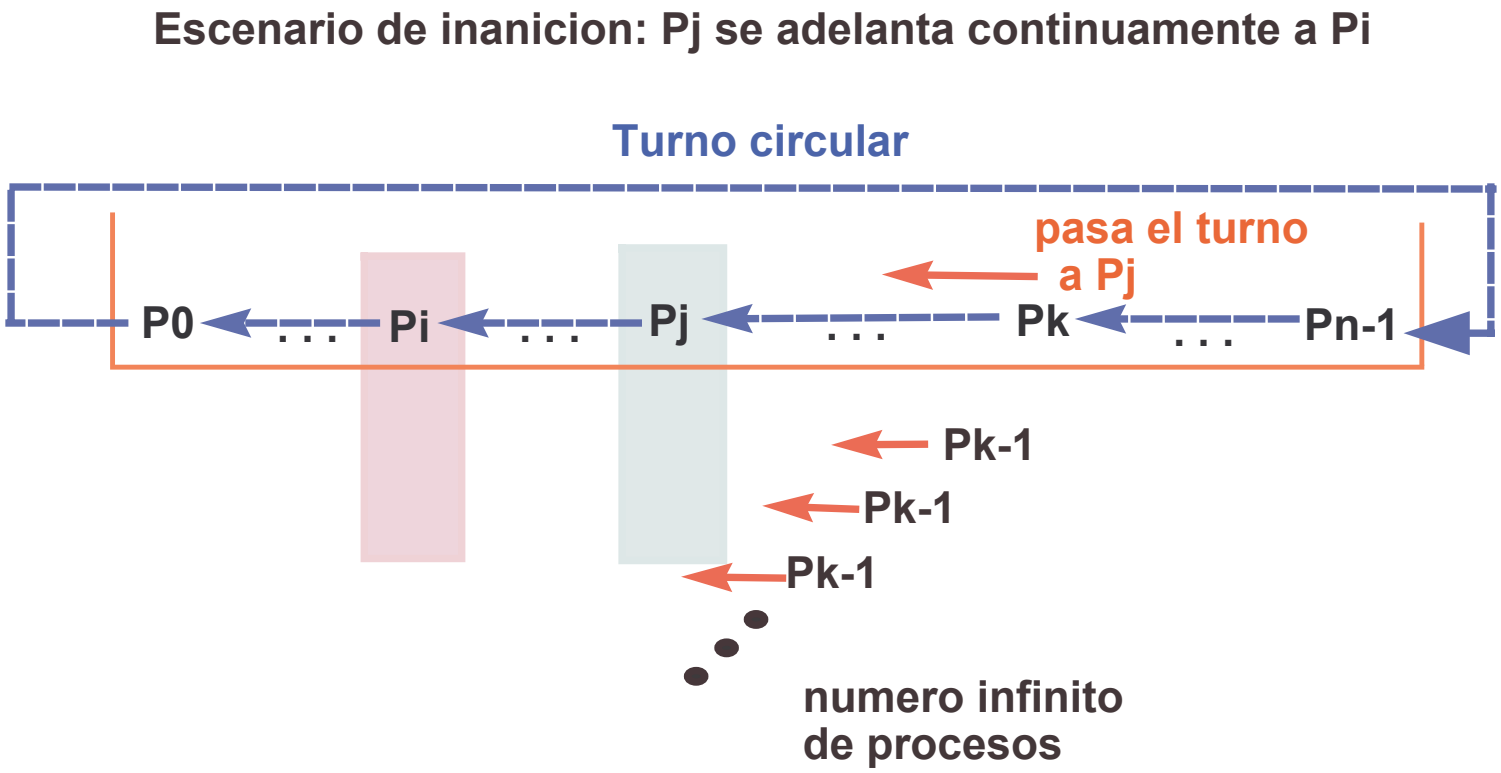
2A barrera asegura que se cumple la propiedad de seguridad

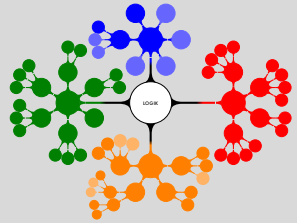
Si un grupo de procesos se ve obligado a ciclar nuevamente, el que posee el turno no puede estar pasivo



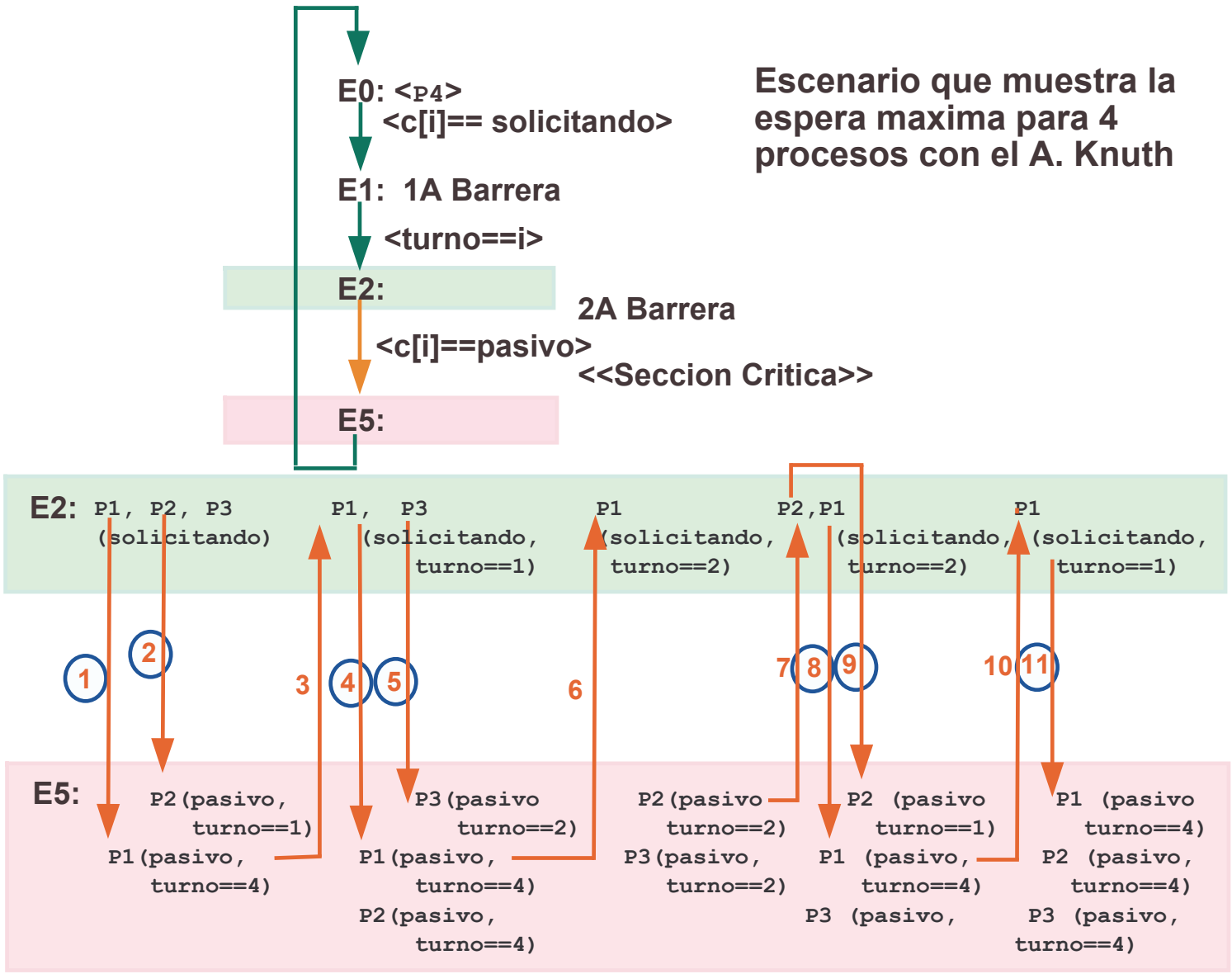
Algoritmo de Knuth para N procesos –II

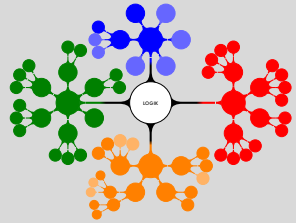
Imposibilidad de la inanición de los procesos si se supone que existe un número finito de ellos en el algoritmo





Algoritmo de Knuth para N procesos –III

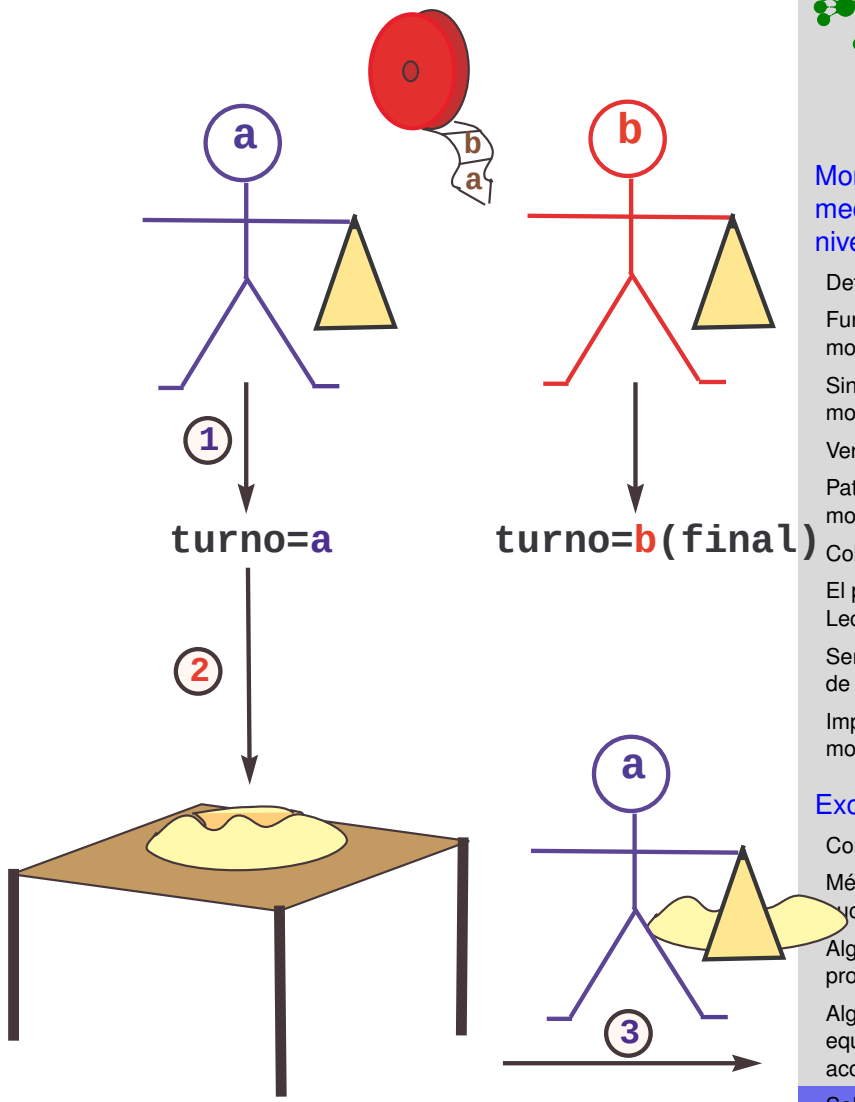


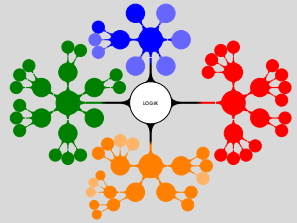


Algoritmo de Peterson

Solución para *2* procesos

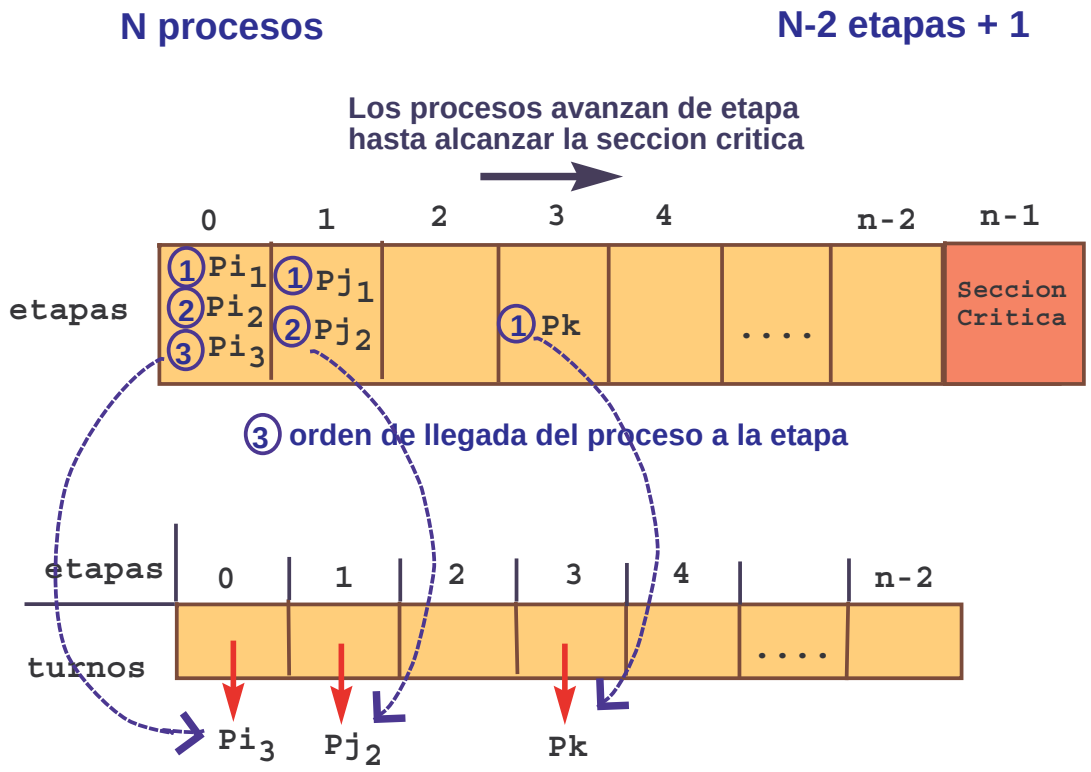
```
var
  solicitado: array[0..1] of boolean;
  turno: 0..1;

Pi::=
 ...
 solicitado[i]:= true; --j=2, i=1
 turno:= i;
 while (solicitado[j] and turno=i) do
  nothing;
 enddo;
 <<seccion critica>>
 solicitado[i]:= false;
 ...
```





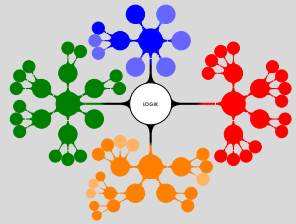
Algoritmo de Peterson-II

Solución para *N* procesos - Idea general:



Variables compartidas entre los procesos:

```
type etapas= -1..n-2;   var c: array[0..n-1] of etapas;
     procesos= 0..n-1;      turno: array[0..n-2] of procesos;
```

Algoritmo de Peterson-III

Solución para *N* procesos

```
Pi
. . .
while true do
 begin
 <<resto de instrucciones>>
 for j=0 to n-2 do
  begin
   c[i]:= j;
   turno[j]:= i;
   while ( (Exists k <> i: c[k] >= j) && turno[j] = i) do
      nothing;
  end;
 enddo;

 c[i]:= n-1; -- metainstruccion
  <<seccion critica>>
 c[i}:= -1;
```

Variables compartidas entre los procesos:

```
var c: array[0..n-1] of -1..n-2;
    turno: array[0..n-2] of 0..n-1;
```

El proceso en etapa j

Bucle de asignacion de etapas a procesos

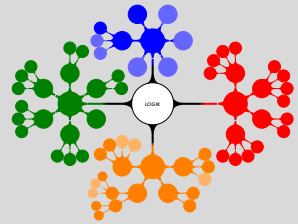
El ultimo en llegar se queda con el turno

Indica que se ha llegado a la S.C.; es una etapa ficticia

Etapa inicial de los procesos

(Exists k <> i: c[k] >= j) && turno[j] = i

No se cumple la condicion si
(a) el proceso esta en la etapa mas avanzada
o (b) ha llegado otro proceso despues a la etapa



Verificación de las propiedades del Algoritmo de Peterson

Se dice que `P_i` precede a un proceso `P_j` sii `c[i] > c[j]`

L1 Un proceso que precede a todos los demás puede avanzar al menos una etapa

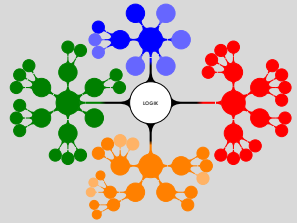
(Exists k <> i: c[k] >= j) && turno[j] = i

Ya que no se cumple la condicion del segundo bucle si:

(a) el proceso esta en la etapa mas avanzada

o (b) ha llegado otro proceso despues a la etapa

- El proceso P_i puede ser adelantado en la etapa siguiente
- Podrían llegar más de un proceso a la etapa `j`
- Pero siempre se cumplirá que P_i avanzará

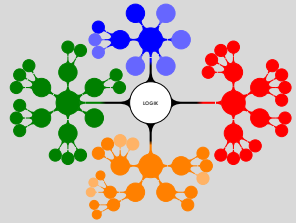


Verificación de las propiedades del Algoritmo de Peterson-II

L2 Un proceso que pasa de la etapa `j` a la `j+1` ha de verificar alguna de estas condiciones:

1. *Precede a todos los demás*
2. *No estaba solo en la etapa j*

- La condición (1) nos sitúa en las condiciones de aplicar el **Lema 1**
- Si (2) $\Rightarrow$ `turno[j] <> i`, luego al proceso se le unió otro
  - Podría suceder que, justo cuando el proceso vaya a avanzar de etapa
  - porque se cumple la condición (1), se le una otro proceso a su etapa.
  - Pero también se cumplirá



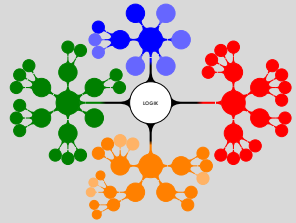
Verificación de las propiedades del Algoritmo de Peterson-III

L3 Si existe al menos dos procesos en la etapa `j`, entonces existe al menos un procesoen cada una de las etapas anteriores

- *La demostración se hace por inducción sobre la variable que representa la etapa j*

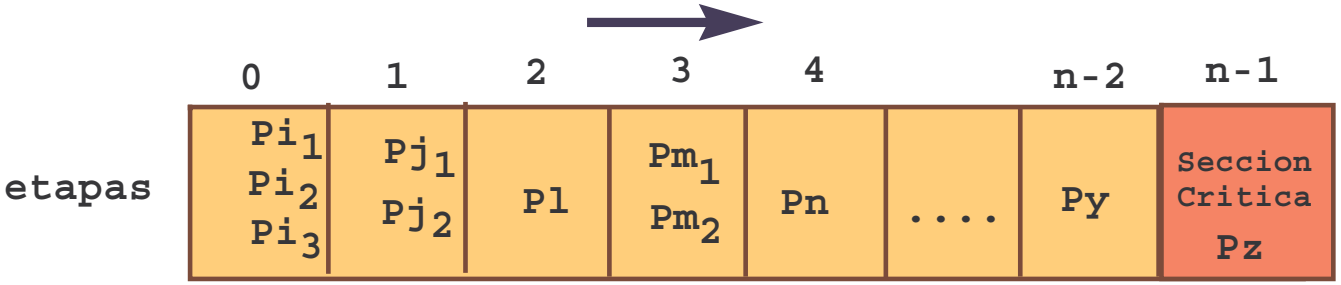
L4 El número máximo de procesos que puede haber en la etapa `j` es `n-j`, con $0 \leq j \leq n-2$

- *La demostración se hace aplicando el* **Lema 3**
- *Por tanto, a la etapa* `n-2` *llegarán como máximo 2 procesos*

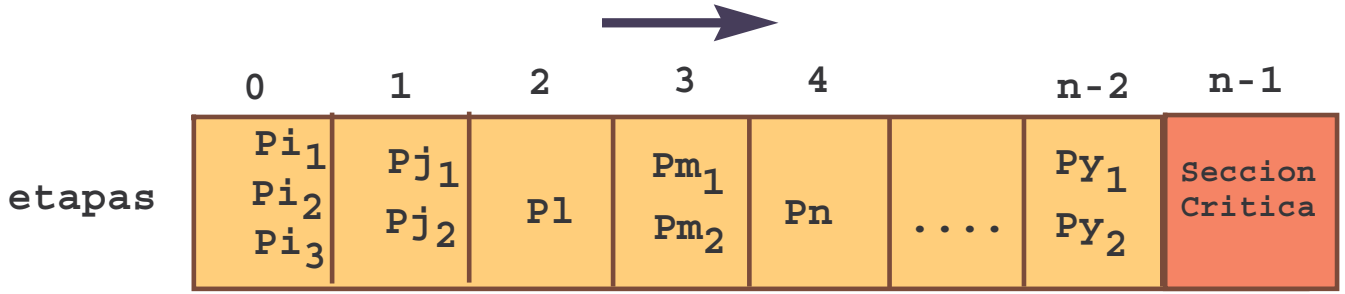


Verificación de las propiedades de seguridad del A. Peterson-IV

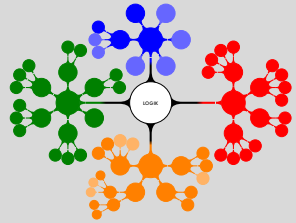
El algoritmo de Peterson cumple con la exclusion mutua en el acceso a la seccion critica



Py no puede avanzar a la siguiente etapa mientras la seccion critica este ocupada (segun las condiciones del Lema 2)

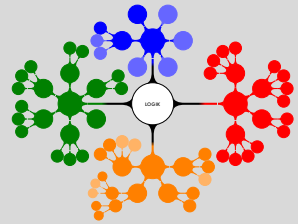


Segun el Lema 2, solo 1 de los 2 procesos en la etapa (n-2) podra avanzar a la seccion critica



Algoritmo de Peterson-IV

```
var c: array[0..N-1] of -1..N-2;
   turno: array[0..N-2] of 0..N-1;
while (true) do
  begin
   Resto de las instrucciones;
   (1) for j=0 TO N-2 do
   (2)   begin
   (3)     c[i]:= j;
   (4)     turno[j]:= i;
   (5)     for k=0 TO N-1 do
   (6)       begin
   (7)         if (k=i) then continue;
   (8)         while(c[k]>=j and turno[j]=i) do
   (9)           nothing;
   (10)          enddo;
   (11)      end;
   (12)   end;
   (13) c[i]:= n-1; /*meta-instruccion*/
   (14) <seccion critica>
   (15) c[i]:= -1
  end
```

Verificación de propiedades concurrentes adicionales

I. Alcanzabilidad de la SC:

demostración por *reducción al absurdo*

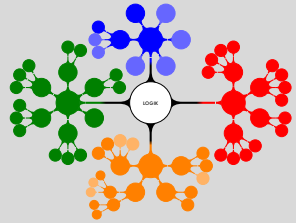
- Todos los procesos se quedan bloqueados al llegar a una etapa y no avanzan más (hipótesis de incorrección)
  1. El proceso precede a los demás $\Rightarrow$ contradice el **Lema 1**
  2. Si no, el proceso llega a una etapa ocupada al menos por otro proceso $\Rightarrow$ se contradice el **Lema 2**

II. Propiedad de equidad

demostración:

- El número máximo de turnos que un proceso cualquiera tendría que esperar con el algoritmo de Peterson es de

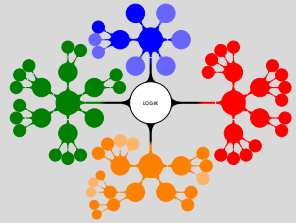
$$r(n) = n - 1 + r(n-1) = \frac{n \times (n-1)}{2} \text{ turnos}$$



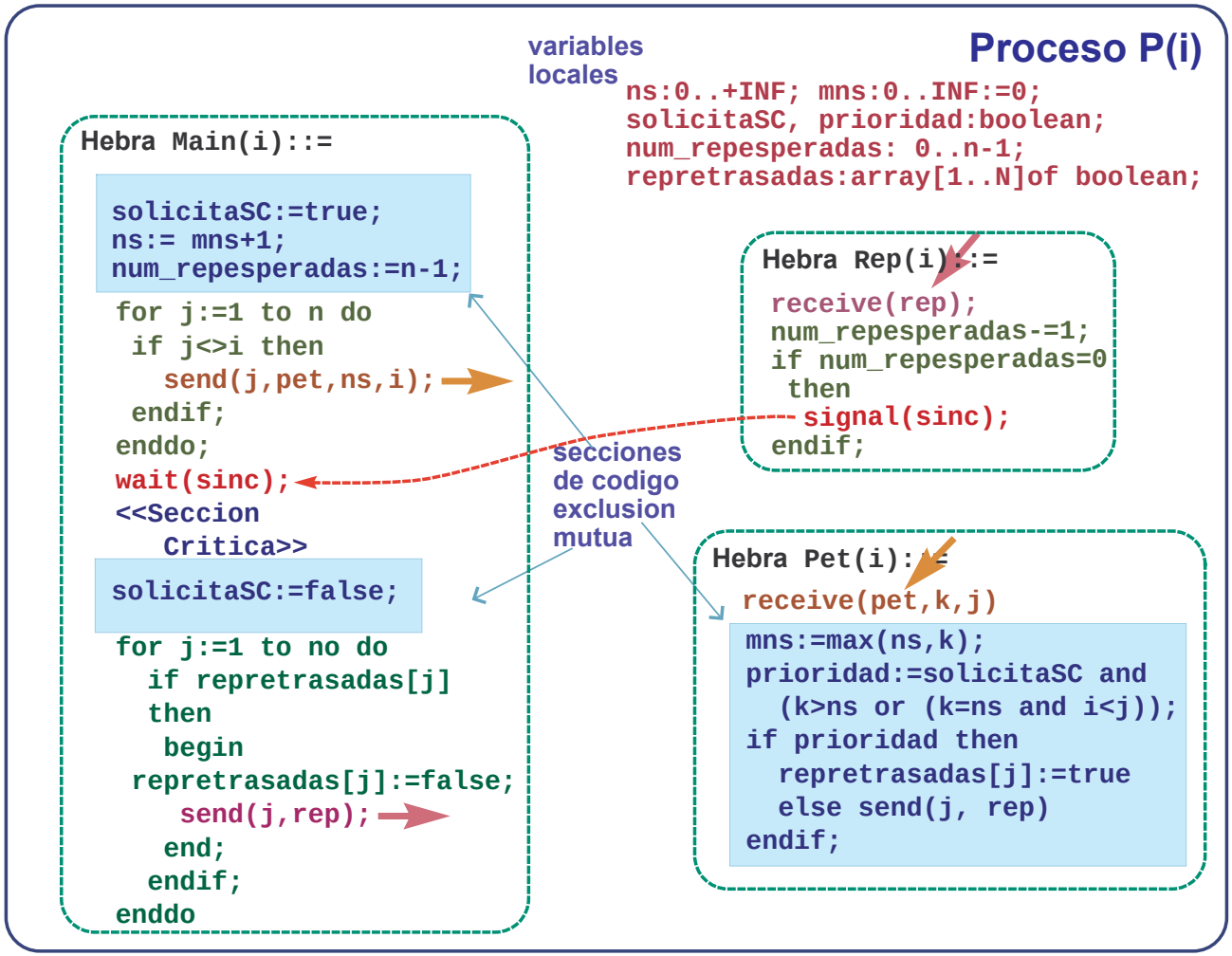
Problemática para la implementación de lso algoritmos en sistemas distribuidos

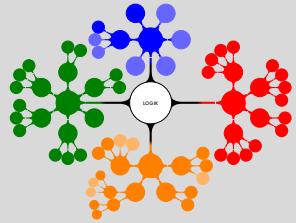
Miscelánea

- Los algoritmos no pueden utilizar sincronización global entre los procesos sólo utilizando variables en memoria compartida
- Se utilizan operaciones atómicas de paso de mensajes para sincronizarlos
- La red de comunicaciones ha de cumplir las siguientes condiciones:
  1. Red de comunicaciones completamente conectada
  2. Transmisión de mensajes sin errores
  3. Retraso variable en la entrega de mensajes con tiempo acotado
  4. Posibles *desencuenciamientos* en la entrega de mensajes en transmisión



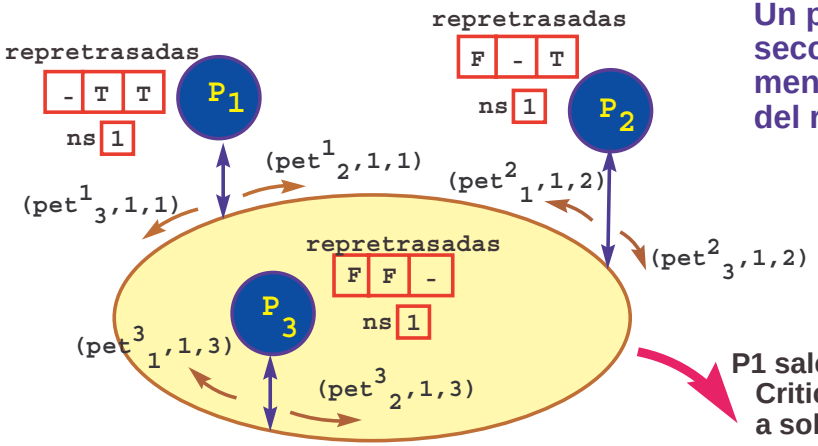
Algoritmo de Ricart-Agrawala





Algoritmo de Ricart-Agrawala-II

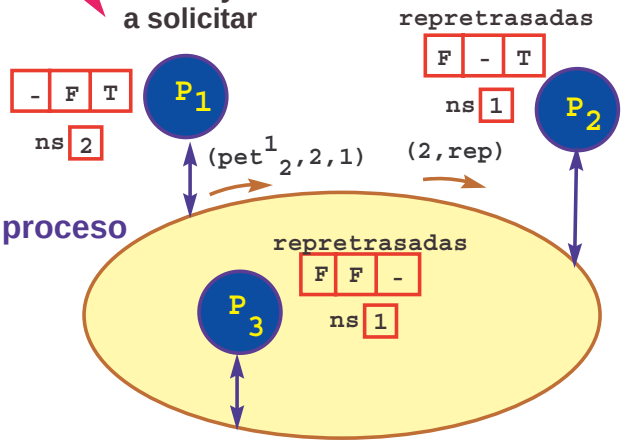
Escenario en el que 3 procesos intentan entrar en Seccion C. inicialmente



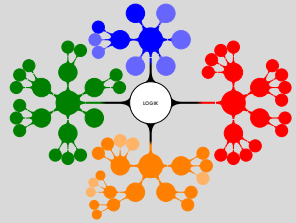
Un proceso solo puede entrar en seccion critica cuando recibe un mensaje de respuesta favorable del resto de los procesos

Se necesitan 2x(n-1) mensajes para implementar el protocolo

P1 sale de Seccion Critica y vuelve a solicitar



Al salir de la seccion critica, un proceso enviara mensajes de respuesta favorable a los procesos que estuvieran esperando



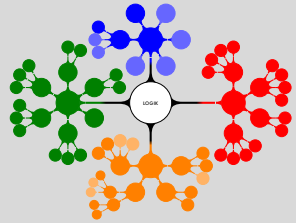
Verificación de las propiedades del algoritmo de Ricart-Agrawala

Exclusión mutua entre procesos al acceder a la sección critica

- P_i y P_j consiguen entrar a la vez en S.C. (hipótesis de incorrección)
- $\Leftrightarrow$ $P_{i,j}$ ha transmitido su petición a $P_{j,i}$, recibiendo contestación favorable
  1. P_i ha enviado contestación favorable antes de generar su número de secuencia $\Rightarrow$ P_j pospone la respuesta
  2. P_j ha enviado contestación favorable antes de generar su número de secuencia $\Rightarrow$ P_i pospone la respuesta
  3. Después de generar su número de secuencia es imposible que cada proceso envie contestación favorable al otro

Alcanzabilidad de la sección crítica

Debido a la ordenación total de las peticiones, es imposible que se retrase indefinidamente la contestación favorable a algún proceso



Algoritmo de Suzuki-Kasami-Ricart-Agrawala

Proceso P(i)

Variables locales a P(i)

```
token_presente: boolean;
en_SC: boolean;
token: array[1..N] of 0..+INF;
peticion: array[1..N] of 0..+INF;
```

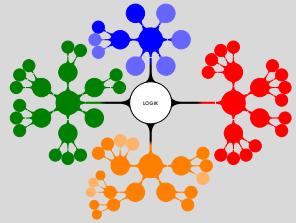
Hebra `pet(i)::=`

```
receive(pet,k,j);
peticion[j]:= max(peticion[j],k)
if token_presente and
   not en_SC then
   for j:=i+1 to n, 1 to i-1 do
     if peticion[j]>token[j] and
        token_presente then
        begin
         token_presente:= false;
         send(j,acceso,token);
        end;
     endif;
   enddo
```

secciones de codigo exclusion mutua

Hebra `main(i)::=`

```
if NOT token_presente then
  begin
   ns:= ns+1;
   broascast(pet,ns,i);
   receive(acceso,token);
   token_presente:= true;
  end;
endif;
en_SC;=true;
<<Seccion
   Critica>>
token[i]:= ns;
en_SC:= false;
for j:=i+1 to n, 1 to i-1 do
  if peticion[j]>token[j] and
     token_presente then
     begin
      token_presente:= false;
      send(j,acceso,token);
     end;
  endif;
enddo
```

Verificación del algoritmo de Suzuki-Kasami-Ricart-Agrawala

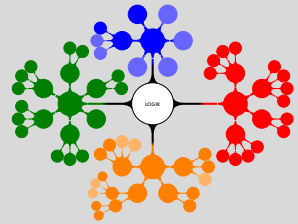
- El que todo proceso acceda en exclusión mutua es equivalente a demostrar el siguiente aserto: "globalmente, el número de variables *token_presente= true* es identicamente igual a la unidad"
  1. El aserto anterior se satisface inicialmente
  2. El aserto anterior se cumple cada vez que e transmite el token. El protocolo necesita **n** mensajes.

Alcanzabilidad de la sección crítica

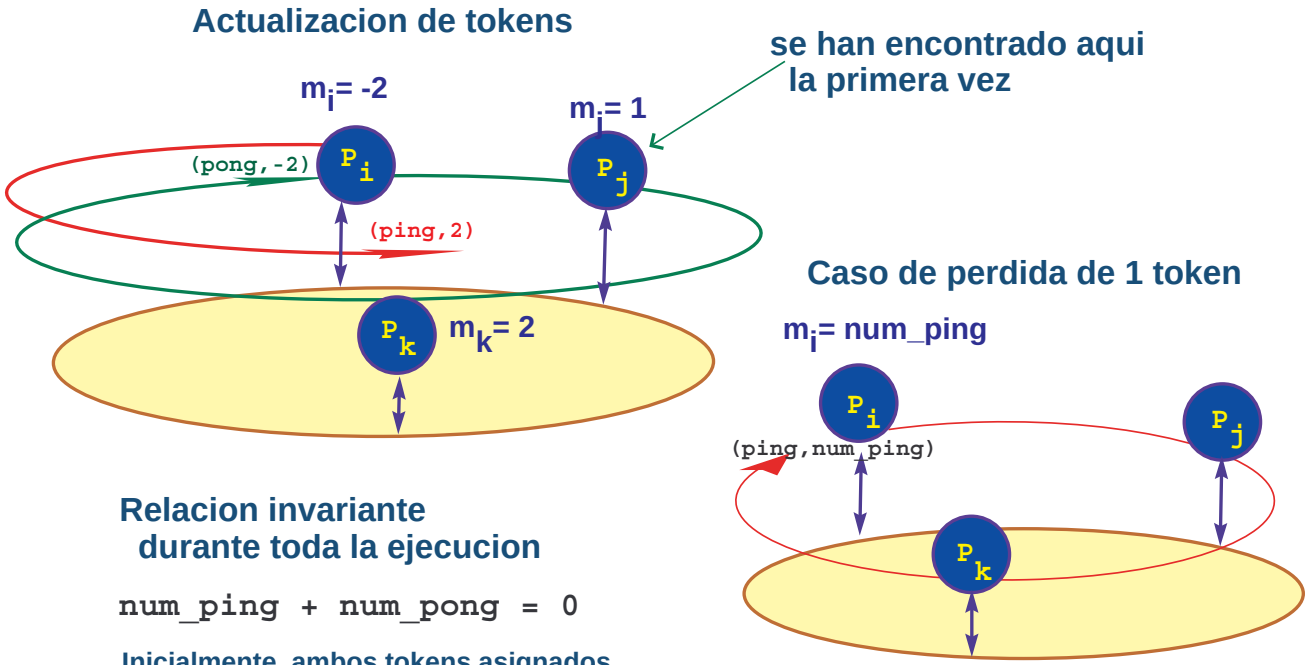
- Si ningún proceso posee el token, en un momento de la ejecución del algoritmo, este ha de estar necesariamene en transmisión

Propiedad de equidad

- Se obliga a que P_j transmita el token al primero que lo solicitó en el orden $\{j+1, j+2, \ldots, n, n-1, \ldots\}$
- ¿ Qué pasa si se pierden mensajes?



Algoritmo de regeneración de token de Misra

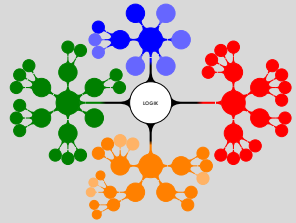


Relacion invariante durante toda la ejecucion

```
num_ping + num_pong = 0
```

Inicialmente, ambos tokens asignados a un solo nodo con:

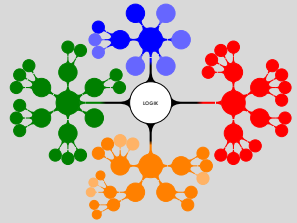
```
num_ping = 1 , num_pong= -1
```

Algoritmo de regeneración de token de Misra-II

Proceso (i)

```
while true do    variable local m_i: int:=0;
  Seleccionar
    Cuando recibido(ping,num_ping) hacer
     if m_i= num_ping then
      begin -- se ha perdido pong, hay que recuperarlo
        num_ping := (num_ping+1)mod n+1;
        num_pong := -num_ping;
      end
     else mi:= num_ping;
    endif;
    endhacer;
    Cuando recibido(pong,num_pong) hacer
     if m_i= num_pong then
      begin -- se ha perdido ping, hay que recuperarlo
        num_pong := -((-num_pong+1)mod n+1;)
        num_ping := -num_pong;
      end
     else mi:= num_pong;
    endif;
    endhacer;
    Cuando se encuentran(ping,pong) hacer
      begin -- |num_ping|=|num_pong|,llevan el "num.colisiones"
        num_ping := (num_ping+1)mod n+1;
        num_pong := -num_ping;
      end
    endhacer;
  endseleccionar;
enddo;
```

Bibliografía

Para más información, ejercicios, bibliografía adicional, o "simplemente inspiración" sobre la temática, se puede consultar:

2.1. Capel-Rodríguez Valenzuela (2012) capítulo 2

Michel Raynal (1986). Algorithms for mutual exclusion. Cambridge: MIT Press.

Andrews (2000) capítulo 5,

Ben Ari (2006) capítulo 7

Michel Raynal (2013). Distributed algorithms for message-passing systems. Springer.

“Concurrency”: `https://web.archive.org/web/20060128114620/http://vl.fmnet.info/concurrent/`

“Concurrency Talk”: `http://shairosenfeld.com/concurrency.html`