

Limitaciones de los semáforos

- El uso de variables globales limita la estructura modular y la escalabilidad, ya que, cada vez que se crece un nuevo proceso, hay que revisarlas.
 - El uso y función de las variables se hace de forma no explícita, lo que dificulta la corrección de los errores.
 - Las operaciones aparecen dispersas y no protegidas, lo que conduce a mayor cantidad de errores.

Por tanto, es necesario un mecanismo que permita el acceso estructurado a los datos, el encapsulamiento de las estructuras de datos y que permitan mecanismos que garantizan la exclusión mutua y la sincronización.

Monitors

Como consecuencia, aparecen los monitores. Una primera definición de monitor viene dada por Hoare y lo define como un mecanismo de alto nivel que permite la creación de objetos abstractos compartidos:

- Un conjunto de variables encapsuladas (datos), que son comunes a los múltiples procesos.
 - Un conjunto de procedimientos (funciones) que controlan manejo de dichos recursos compartidos.

Además, se encarga de garantizar la exclusión mutua de las variables compartidas y de la sincronización de los procesos, mediante la espera bloqueada. La espera bloqueada permite suspender los procesos sin que estos consuman ciclos del procesador.

Algunas ~~comunicaciones~~ interacciones entre los procesos.

- Heterogeneidad
 - Reestructuración en el acceso a los datos
 - Optimización a los procesos de las operaciones de sincronización.
 - Restabilización basada en la parametrización
 - Verificación mediante el uso de medios más simples que los monitores

La exclusión mutua estará garantizada por la propia definición del módulo monitor. Esta exclusión garantiza que nunca dos procesos estén ejecutando simultáneamente el mismo procedimiento de monitor (en vez de el mismo o distintos).

Digitized by srujanika@gmail.com

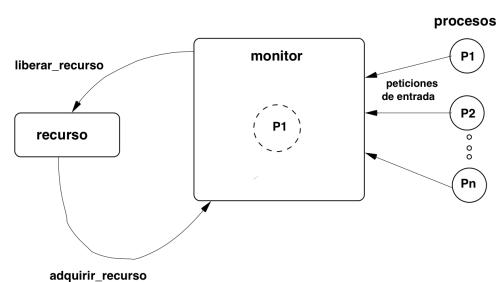
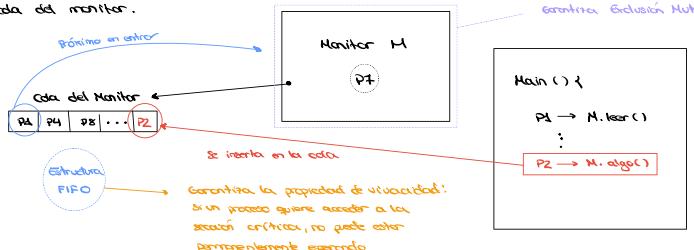
- **variables permanentes** (**conjunto de datos privados**): solo pueden ser accedidas dentro del monitor. Son el **estado interno del monitor**.
 - **procedimientos** (**funciones**): permiten modificar el **estado interno del monitor**. Algunas o todas (las funciones públicas) constituyen la **interfaz externa** del monitor y podrán ser llamadas por los procesos que comparten el recurso. → **procedimientos exportados**
 - **codificación de inicializaciones** (**constantes**): se encargan de fijar el **estado interno inicial**.

Como ventaja, la implementación de los procedimientos se puede combinar sin modificar el código de los componentes de la utilización.

los monitores son dispositivos pasivos, ya que una vez ejecutado el código de inicialización, el código de los procedimientos sólo se ejecuta cuando estos son

In addition, future research on the relationship between culture and adolescents' self-esteem and mental health

Si un proceso está dentro del monitor y otro proceso intenta ejecutar un procedimiento del mismo, entonces, este último queda bloquizado y se inserta (espera) en la lista de procesos.

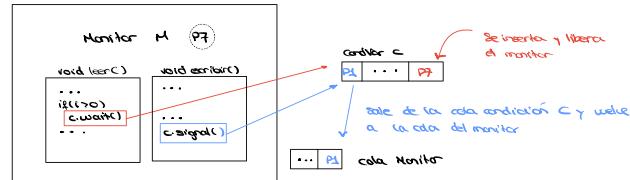


Cuando un proceso abandona el monitor (finaliza la ejecución del procedimiento), se desbloquea un proceso de la cola, y este puede acceder al monitor. Si la cola del monitor está vacía, el monitor está libre; por tanto, el primer proceso que ejecuta una llamada a uno de sus procedimientos, entra al monitor.

Para implementar la sincronización, se requiere de una facilidad para que los procesos hagan esperas bloqueadas, hasta que se verifique cierta condición. Al contrario que los semáforos, solo se dispone de sentencias de bloqueo y activación, y con los valores de las variables permanentes los que determinan si se cumple la condición o no.

La sincronización en monitores se realiza mediante variables condición (cadas), que posibilitan la espera de diferentes condiciones dentro de un monitor.

- c.wait() → Bloquea **SIEMPRE**
- c.signal() → Si la cola C no está vacía, desbloquea el primer proceso de la cola (FIFO). En caso contrario, rebalse nodo.
- c.gueve() → Devuelve true si hay algún proceso esperando en la cola, falso en caso contrario. Se usa para evitar los cambios de contexto innecesarios que realiza c.signal()



Mecanismos de señalización de los monitores

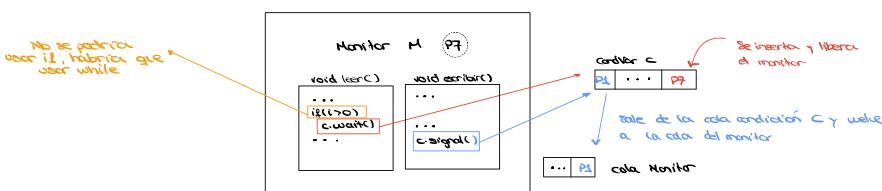
Cuando se ejecuta signal pueden ocurrir dos cosas:

- 1) El proceso señalador (el que ejecuta el signal) continua su ejecución, mientras que el proceso señalado se bloquea hasta que pueda adquirir nuevamente la exclusión mutua. → **señal NO desplazante**
- 2) El proceso señalador abandona el monitor tras el signal, sin esperar el código que haya después, y el proceso señalado se reactiva inmediatamente. → **señal desplazante**

Teniendo esto en cuenta, existen distintos tipos de mecanismo:

- señalar y continuar (SC): señal explícita, no desplazante. El señalador ejecuta inmediatamente la ejecución del código del procedimiento del monitor tras signal, mientras que el señalado abandona la cola condición y espera en la del monitor hasta readquirir la E.M. y reanudar su código tras el wait.

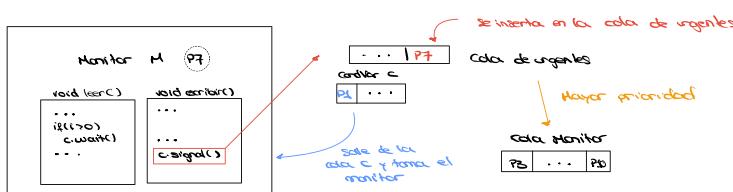
Este tiene un inconveniente. Como el señalado espera en la cola del monitor, la cual sigue una estructura FIFO, puede ser que la condición debe de ser válida cuando este responde la ejecución de su código. Luego, es necesario que se recompruebe la condición tras el wait.



- señalar y salir (SS): señal explícita, desplazante. El **señalador abandona el monitor inmediatamente** tras ejecutar signal, mientras que el señalado reanuda inmediatamente la ejecución del código del procedimiento del monitor programado tras la operación wait. Esta semantics condiciona el estilo de programación (ya que la ejecución del signal implica la terminación del procedimiento que estaba ejecutando el proceso señalador), por lo que obliga a colocar la operación signal como última instrucción de los procedimientos que la usen.

- señalar y esperar (SE): señal explícita, desplazante, el proceso señalador espera en la cola de entrada al monitor. El proceso señalador se bloquea en la cola del monitor hasta readquirir E.M. y ejecutar el código tras el signal, mientras que el señalado reanuda inmediatamente la ejecución del código del tras wait.

- señalar y esperar urgente (SU): señal explícita, desplazante, el proceso señalador espera en la cola de procesos urgentes. Funciona igual que SE, pero el señalado se bloquea en la cola de urgentes, que tiene más prioridad que la cola del monitor.



Todas las semánticas son capaces de resolver los mismos problemas. En cuanto a eficiencia, la semántica SE condiciona el estilo de programación y puede llevar a aumentar de forma artificial el número de procedimientos. Por otro lado, las semánticas SC y SU resultan inefficientes cuando no hay código tras signal, ya que en ese caso, se emplea tiempo en bloquear al ejecutador, para posteriormente reactivarse y abandonar el monitor sin hacer nada. Por último, la semántica SC también es un poco inefficiente al obligar a usar un bucle por cada instrucción signal.

Verificación de programas con monitores

La verificación de la corrección de un programa concurrente con monitores impone probar la corrección de cada monitor, la de cada proceso de forma aislada y la de ejecución concurrente de procesos implicados. El programador no puede conocer, a priori, la traza concreta de llamadas a los procedimientos del monitor, por lo que, el enfoque de verificación que vamos a seguir es utilizar un invariante de monitor.

Definición invariantes de un monitor (IM) a una propiedad que el monitor cumple siempre, pero que es específico de cada monitor diseñado por un programador.

La demostración de corrección se basa en el IM, que es una relación constante entre los valores permitidos de las variables permanentes del monitor.

- **Axioma inicialización de variables:** El invariante del monitor debe ser cierto en su estado inicial, después de la inicialización de las variables permanentes.

$$IM \models \text{inicializaci}\ddot{\text{o}}n\ \text{variables permanentes} \wedge IM$$

- El invariante del monitor debe ser cierto antes y después de cada llamada a un procedimiento del monitor.

$$IM \models \text{IN} \wedge IM \models \text{procedimiento } f \text{ OUT} \wedge IM \models$$

- El invariante del monitor debe ser cierto antes de cada operación wait y después de cada operación signal. Además, justo antes de una operación signal sobre una variable condición c, debe ser cierta la condición lógica asociada a dicha variable.

$$\longrightarrow (\text{Para poder desbloquear un proceso})$$

- Axioma operación c.wait():

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

No tiene en cuenta que el proceso termine de ejecutar c.wait()

- Axioma operación c.signal():

$$IM \models T \text{resivo}(c) \wedge L \wedge C \wedge c.\text{signal}() \wedge IM \models C$$

Se expone semántica desplazante. El monitor mantiene el estado expresado por C hasta que un proceso bloqueado se reanude y continúe su ejecución dentro del monitor. Además no tiene en cuenta el orden de desbloqueo de los procesos.

Notas de afirmaciones leídas en el libro

- Para que las variables permanentes de un monitor no alcancen valores erróneos durante la ejecución (ya que son modificadas por parte de los procesos concurrentes del programa), no se puede utilizar dentro del texto de los procedimientos ninguna variable declarada fuera del monitor.
- Los procedimientos del monitor pueden verse interrumpidos varias veces durante su ejecución, por lo que deben poseer la propiedad de reentrancia (código compartido por varios procesos, que pueden ejecutar parte del mismo, interrumpirlo y volver a ejecutarse donde se quedaron sin que se produzca pérdida de información)

Condiciones de Dijkstra

Las condiciones que debe cumplir una solución correcta al problema de la exclusión mutua son los siguientes:

- 1) No hacer suposiciones acerca de las instrucciones o número de procesos soportados por la máquina.
- 2) No hacer ninguna suposición acerca de la velocidad de ejecución de los procesos, excepto que no es cero.
- 3) Cuando un proceso está en sección no-crítica no puede impedir a otro que entre en ella.
- 4) La sección crítica será alcanzada por alguno de los procesos que intentan entrar (Excluye la posibilidad de que en una ejecución se llegara a un interbloqueo, es decir, que ningún proceso consiga jamás entrar en sección crítica, pero no asegura que todos los procesos consigan alguna vez entrar o se trage de forma esporádica).

Método de refinamiento sucesivo

Dijkstra propuso un método para detener el algoritmo en 4 pasos o modificaciones de un esquema inicial. Inicialmente, se asume que los procesos alternan su entrada a la sección crítica según el valor de una variable global compartida entre ellos, que asigna el "turno".

Primera Etapa

Se utiliza una variable turno, que contiene el identificador del proceso que puede entrar a la sección crítica:

```

P1
-----
while true do begin
  Resto;
  while turno <= 1 do
    nothing;
  enddo;
  S.C.
  turno := 2; combi de turno
end
enddo;

P2
-----
while true do begin
  Resto;
  while turno <= 2 do
    nothing;
  enddo;
  S.C.
  turno := 1;
end
enddo;

```

La solución garantiza la exclusión mutua de los procesos, por lo que es segura, pero no garantiza la 3º condición de Dijkstra, ya que los procesos solo pueden entrar a la sección crítica alternativamente.

Segunda Etapa

La alternancia estricta en el acceso a la sección crítica que se producía en la primera etapa era debido a que, para decidir qué proceso entraba en sección crítica, se tenía que almacenar la información en la variable global turno, que sólo la cambia un proceso al salir de esta.

La idea ahora es asociar a cada proceso su información de estado a una variable clave, que indique si el proceso está en sección crítica o no:

```

Inicialmente: c1=c2= 1;
P1
-----
while true do begin
  Resto;
  while c2=0 do
    nothing;
  enddo;
  c1:= 0; mientras el otro esté en la S.C.
  S.C.
  c1:= 1; haci una espera ociosa
  esperar ociosa;
  El 1 indica que no están en S.C. y el 0 que sí
  end;
enddo;

P2
-----
while true do begin
  Resto;
  while c1=0 do
    nothing;
  enddo;
  c2:= 0;
  S.C.
  c2:= 1;
end
enddo;

```

En este caso, falla la propiedad de seguridad, ya que si P1 y P2 se ejecutan a la misma velocidad, comprobarán que el otro proceso no está en la S.C. y entrarán ambos a la vez, incumpliendo la exclusión-mutua.
Además, como los procesos hacen espera ociosa, puede ocurrir que un proceso esté comprobando el estado del otro, a la vez que este lo modifica.

Tercera Etapa

La solución que ahora se propone consiste en cambiar la posición de la sentencia de asignación de la clave del proceso antes del bucle de espera ociosa. De esta forma, sería imposible que un proceso tome dicho bucle con un valor de su clave distinto de cero, evitándose así el peor escenario de falta de seguridad.

```

Inicialmente: c1=c2= 1;
P1
-----
while true do begin
  Resto;
  c1:= 0;
  while c2=0 do
    nothing;
  enddo;
  S.C.
  c1:= 1;
end
enddo;

P2
-----
while true do begin
  Resto;
  c2:= 0;
  while c1=0 do
    nothing;
  enddo;
  S.C.
  c2:= 1;
end
enddo;

```

Sin embargo, esta solución sigue sin ser correcta, ya que, nuevamente, nos encontramos con una condición de carrera, ya que si ambos procesos tienen la misma velocidad, se podría producir una situación de interbloqueo. Ambos procesos podrían combinar su clave a 0 y, a continuación, bloquearse realizando iteraciones de los bucles infinitamente.

Cuarta Etapa

Lo que causa el error de la tercera etapa es que cuando un proceso modifica el valor de su clave, no sabe si otro proceso está haciendo lo mismo, concurrentemente con él. Luego, la solución ahora sería permitir que un proceso vuelva a cambiar el valor de su clave a 1 si, después de asignar su clave a 0, comprueba que el otro proceso también asignó su clave a cero.

```

Inicialmente: c1=c2= 1;
P1
-----
while true do begin
  Resto;
  c1:= 0;
  while c2=0 do
    begin
      c1:= 1;
      while c2=0 do
        nothing;
      enddo;
      c1:=0;
    end
    enddo;
    S.C.
    c1:= 1;
  end
enddo;

P2
-----
while true do begin
  Resto;
  c2:= 0;
  while c1=0 do
    begin
      c2:= 1;
      while c1=0 do
        nothing;
      enddo;
      c2:= 0;
    end
    enddo;
    S.C.
    c2:= 1;
  end
enddo;

```

con esta solución, sigue existiendo la posibilidad de que se produzca interbloqueo, pero, ahora, es menos probable que en la tercera etapa

Algoritmo de Dijkstra

Se basa en la primera y cuarta etapa del algoritmo de Dijkstra. En este algoritmo, un proceso que intenta entrar en sección crítica asigna su clave a cero. Si encuentra que la clave del otro es 0, entonces, revisa el valor de turno. Si posee el turno, entonces insiste y comprueba sucesivamente la clave del otro proceso. Finalmente el otro proceso, cambiando su clave a 1, le cede el turno y este consigue entrar en la s.c.

```

P1
-----
while true do begin
  Resto;
  c1:= 0;
  while c2=0 do
    if turno= 2 then
      begin
        c1:= 1;
        while turno= 2 do
          nothing;
        enddo;
        c1:=0;
      end
    endif
    enddo;
    S.C.
    turno:= 2;
    c1:= 1;
  end
enddo;

P2
-----
while true do begin
  Resto;
  c2:= 0;
  while c1=0 do
    if turno= 1 then
      begin
        c2:= 1;
        while turno=1 do
          nothing;
        enddo;
        c2:= 0;
      end
    endif
    enddo;
    S.C.
    turno:= 1;
    c2:= 1;
  end
enddo;

```

El proceso P2 entra en S.C. sólo si $c_j = 1$. La clave de un proceso sólo la puede modificar el propio proceso. El proceso P1 comprueba la clave del proceso c_j sólo después de asignar su clave $c_i = 0$. Luego, cuando un proceso P1 entra y se mantiene en S.C. se verifica que $c_i = 0 \wedge c_j = 1$. Los valores de estas variables indican que sólo un proceso puede acceder a S.C.

Necesidad de S.C.:

- Si apagamos verde P1 puede entrar, entonces encontrará $c_j = 1$ y podrá entrar.
- Si P1 y P2 intentan entrar con $turno = i$, entonces:
 - Pj encuentra $c_i = 1 \Rightarrow$ entra Pj
 - Pj encuentra $c_i = 0 \wedge turno = 1$, entonces se detiene en el bucle interno y cambia $c_j = 1$. Si P1 encuentra $c_j = 0$, se detiene en el bucle externo, porque $c_j = 1 \wedge$, por último, entra P1 a S.C.

Existe una posible iniciación en caso de que P2 haga un proceso muy rápido y repetitivo, por lo que podría estar entrando continuamente en las s.c. e impidiendo a P1 hacerlo (P1 sale del bucle interior, pero nunca puede escribir el valor de c_i , ya que P2 lo está leyendo y se lo impide).

La rapidez del algoritmo dependerá del hardware.

Algoritmo de Peterson

Solución para 2 procesos

Las variables compartidas por los procesos son:

```

var
  c: array [0..1] of boolean; → Se inicializa a false e indica si P1 intenta o no entrar en S.C.
  turno: 0..1; → Sirve para resolver conflictos cuando ambos intentan acceder simultáneamente.

  ... → Supongamos los valores i=0 y j=1
  c[i]:= true;
  turno:= i;
  while(c[j] and turno=i) do
    nothing;
  enddo;
  <sección crítica>
  c[i]:= false;
  ...
  
```

Para demostrar que este algoritmo cumple la propiedad de exclusión mutua, se sigue con un razonamiento por reducción al absurdo. Suponer P0 y P1 se encuentran en su S.C., entonces los valores de sus claves son necesariamente $c[0]=c[1]=\text{TRUE}$, sin embargo, las condiciones de espera de los dos procesos no han podido ser simultáneamente ciertas, ya que la variable turno ha debido tomar el valor 0 ó 1.

Por tanto, sólo uno de los procesos ha podido entrar en la sección crítica. Digamos que es P_1 , ya que este entró en el turno j . P_1 no pudo haber entrado en S.C. junto con P_2 , ya que para esto habría necesitado encontrar $\text{turno} = i$, sin embargo, la inicial asignación que P_2 le pidió haber hecho a la variable turno le era desfavorable.

También podemos demostrar la ausencia de interbloqueo. Basta suponer que el proceso P_3 está pasivo y no intenta ejecutarse, o bien que P_3 está esperando a entrar en la S.C. continuamente. En el primer caso, $c[i:j] = \text{FALSE}$ y P_3 puede entrar en la S.C. El segundo caso es imposible, ya que la variable compartida turno hace ser 0 ó 1, y hará alguna de las condiciones de espera cierta, permitiendo entrar en la S.C. correspondiente. Por último, podemos demostrar la equidad de los procesos. Supongamos que P_0 está bloqueado, esperando a la S.C. y que P_1 está monopolizando el acceso. Esto lleva a una contradicción, ya que si P_1 intenta entrar de nuevo en la S.C. hará $\text{turno} = 1$, validando la condición de entrada de P_0 .

Solución para N procesos

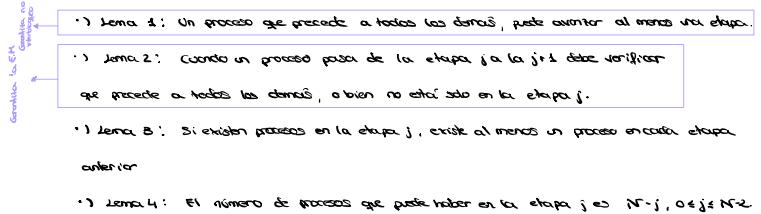
La generalización consiste en utilizar repetidamente $(N-1)$ -veces la solución de dos procesos, para dejar esperando al menos 1 proceso, hasta que solo quede uno, el cual puede entrar a la S.C. Los valores iniciales de todos los valores de los arrays c y turno son -1 y 0 , respectivamente. $c[i:j]=-1$ indica P_i pasivo y no ha intentado entrar en el protocolo y $c[i:j]=j$ significa que P_j está en la etapa j -ésima del protocolo.

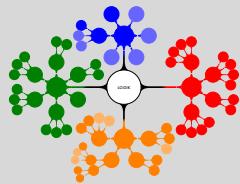
```
var c: array[0..N-1] of -1..N-2; /*los valores del array c representan
                                    las etapas por las que pasa un proceso
                                    antes de entrar en s.c.*/
turno: array[0..N-2] of 0..N-1; /*representa el no. de proceso que tiene el
                                    turno en cada etapa, para resolver posibles
                                    conflictos*/

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-instrucción*/
            (14)    <sección crítica>
            (15)    c[i]:= -1
        end
    enddo;
```

se dice que P_i precede a P_k si $c[i] > c[k]$ (esta una etapa adelantada)

La verificación debe cumplir:





Ejemplo: verificación de un monitor con señales desplazantes

```

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

```

```

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

```

Monitores como mecanismo de alto nivel

Definición de monitor

Funcionamiento de los monitores

Sincronización en monitores

Verificación de monitores

Patrones de solución con monitores

Colas de prioridad

El problema de los Lectores/escritores

Semántica de las señales de los monitores

Implementación de los monitores

Exclusión mutua

Condiciones de Dijkstra

Método de refinamiento sucesivo

Algoritmo de Dijkstra y problemas de vivacidad

Algoritmo de Knuth y equidad relativa en el acceso a la SC

Solución totalmente correcta para N procesos

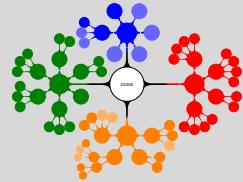
Algoritmos distribuidos para el problema de

Patrones de uso de monitores

Sincronización en memoria compartida

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)



Monitores como mecanismo de alto nivel

Definición de monitor

Funcionamiento de los monitores

Sincronización en monitores

Verificación de monitores

Patrones de solución con monitores

Colas de prioridad

El problema de los Lectores/escritores

Semántica de las señales de los monitores

Implementación de los monitores

Exclusión mutua

Condiciones de Dijkstra

Método de refinamiento sucesivo

Algoritmo de Dijkstra y problemas de vivacidad

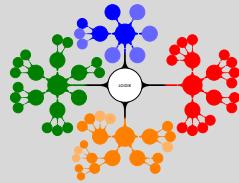
Algoritmo de Knuth y equidad relativa en el acceso a la SC

Solución totalmente correcta para N procesos

Algoritmos distribuidos para el problema de

Monitor para Espera Única (EU)

Sincronización en memoria compartida



```
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;
//Invariant: 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;
```

Monitores como mecanismo de alto nivel

Definición de monitor

Funcionamiento de los monitores

Sincronización en monitores

Verificación de monitores

Patrones de solución con monitores

Colas de prioridad

El problema de los Lectores/escritores

Semántica de las señales de los monitores

Implementación de los monitores

Exclusión mutua

Condiciones de Dijkstra

Método de refinamiento sucesivo

Algoritmo de Dijkstra y problemas de vivacidad

Algoritmo de Knuth y equidad relativa en el acceso a la SC

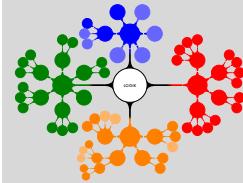
Solución totalmente correcta para N procesos

Algoritmos distribuidos para el problema de

Patrón de Exclusión Mutua (EM)

Sincronización en memoria compartida

```
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
```



Monitores como mecanismo de alto nivel

Definición de monitor

Funcionamiento de los monitores

Sincronización en monitores

Verificación de monitores

Patrones de solución con monitores

Colas de prioridad

El problema de los Lectores/escritores

Semántica de las señales de los monitores

Implementación de los monitores

Exclusión mutua

Condiciones de Dijkstra

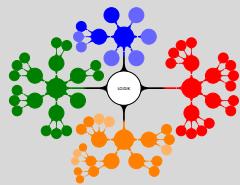
Método de refinamiento sucesivo

Algoritmo de Dijkstra y problemas de vivacidad

Algoritmo de Knuth y equidad relativa en el acceso a la SC

Solución totalmente correcta para N procesos

Algoritmos distribuidos para el problema de



Monitores como mecanismo de alto nivel

Definición de monitor

Funcionamiento de los monitores

Sincronización en monitores

Verificación de monitores

Patrones de solución con monitores

Colas de prioridad

El problema de los Lectores/escritores

Semántica de las señales de los monitores

Implementación de los monitores

Exclusión mutua

Condiciones de Dijkstra

Método de refinamiento sucesivo

Algoritmo de Dijkstra y problemas de vivacidad

Algoritmo de Knuth y equidad relativa en el acceso a la SC

Solución totalmente correcta para N procesos

Algoritmos distribuidos para el problema de

Monitor Productor/Consumidor

```

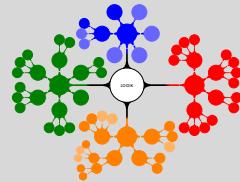
Monitor PC ;
var valor_com:integer; //valor compartido
pendiente:boolean; //true: valor escrito y no leido
cola_prod:condition;
//espera productor hasta que pendiente == false
cola_cons:condition;
//espera consumidor hasta que pendiente == true
procedure escribir( v:integer );
begin
  if pendiente then
    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;

```

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 arbitraria y después llaman a *cita*
- Ningún proceso termina *cita* antes de que haya al menos n de ellos que la hayan iniciado (donde $1 < n < p$). Despues 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)



Monitores como mecanismo de alto nivel

Definición de monitor

Funcionamiento de los monitores

Sincronización en monitores

Verificación de monitores

Patrones de solución con monitores

Colas de prioridad

El problema de los Lectores/escritores

Semántica de las señales de los monitores

Implementación de los monitores

Exclusión mutua

Condiciones de Dijkstra

Método de refinamiento sucesivo

Algoritmo de Dijkstra y problemas de vivacidad

Algoritmo de Knuth y equidad relativa en el acceso a la SC

Solución totalmente correcta para N procesos

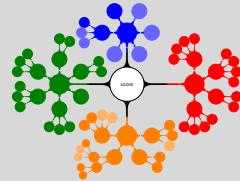
Algoritmos distribuidos para el problema de

Diseño del monitor *barrera parcial*

Sincronización en memoria compartida

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)



Monitores como mecanismo de alto nivel

Definición de monitor

Funcionamiento de los monitores

Sincronización en monitores

Verificación de monitores

Patrones de solución con monitores

Colas de prioridad

El problema de los Lectores/escritores

Semántica de las señales de los monitores

Implementación de los monitores

Exclusión mutua

Condiciones de Dijkstra

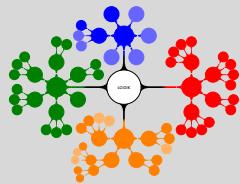
Método de refinamiento sucesivo

Algoritmo de Dijkstra y problemas de vivacidad

Algoritmo de Knuth y equidad relativa en el acceso a la SC

Solución totalmente correcta para N procesos

Algoritmos distribuidos para el problema de



Una posible implementación del monitor

```
Monitor BP //monitor Barrera Parcial 
var cola : {\bf condition}; //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(); //despertarlo
        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 }
```

Monitores como mecanismo de alto nivel

Definición de monitor

Funcionamiento de los monitores

Sincronización en monitores

Verificación de monitores

Patrones de solución con monitores

Colas de prioridad

El problema de los Lectores/escritores

Semántica de las señales de los monitores

Implementación de los monitores

Exclusión mutua

Condiciones de Dijkstra

Método de refinamiento sucesivo

Algoritmo de Dijkstra y problemas de vivacidad

Algoritmo de Knuth y equidad relativa en el acceso a la SC

Solución totalmente correcta para N procesos

Algoritmos distribuidos para el problema de

GIIM. Relación de problemas-Grupo A. Tema 2-1. 21/10/2021

51. Aunque un monitor garantiza la exclusión mutua, los procedimientos tienen que ser reentrantes. Explicar por qué.
52. Se consideran dos tipos de recursos accesibles por varios procesos concurrentes (denominamos a los recursos como recursos de tipo 1 y de tipo 2). Existen N_1 ejemplares de recursos de tipo 1 y N_2 ejemplares de recursos de tipo 2.

Para la gestión de estos ejemplares, queremos diseñar un monitor (con semántica SU) que exporta un procedimiento (`pedir_recurso`), para pedir un ejemplar de uno de los dos tipos de recursos. Este procedimiento incluye un parámetro entero (`tipo`), que valdrá 1 ó 2 indicando el tipo del ejemplar que se desea usar, asimismo, el monitor incorpora otro procedimiento (`liberar_recurso`) para indicar que se deja de usar un ejemplar de un recurso previamente solicitado (este procedimiento también admite un entero que puede valer 1 ó 2, según el tipo de ejemplar que se quiera liberar). En ningún momento puede haber un ejemplar de un tipo de recurso en uso por más de un proceso.

En este contexto, responde a estas cuestiones:

- (a) Implementa el monitor con los dos procedimientos citados, suponiendo que N_1 y N_2 son dos constantes arbitrarias, mayores que cero.
 - (b) El uso de este monitor puede dar lugar a interbloqueo. Esto ocurre cuando más de un proceso tiene algún punto en su código en el cual necesita usar dos ejemplares de distinto tipo a la vez. Describe la secuencia de peticiones que da lugar a interbloqueo.
 - (c) Una posible solución al problema anterior es obligar a que si un proceso necesita dos recursos de distinto tipo a la vez, deba de llamar a `pedir_recurso`, dando un parámetro con valor 0, para indicar que necesita los dos ejemplares. En esta solución, cuando un ejemplar quede libre, se dará prioridad a los posibles procesos esperando usar dos ejemplares, frente a los que esperan usar solo uno de ellos.
53. Escribir una solución al problema de lectores-escritores con monitores:
 - (a) *Con prioridad a los lectores*: quiere decir que, si en un momento puede acceder al recurso tanto un lector como un escritor, se da paso preferentemente al lector.
 - (b) *Con prioridad a los escritores*: quiere decir que, si en un momento puede acceder tanto un lector como un escritor, se da paso preferentemente al escritor.
 - (c) *Con prioridades iguales*: en este caso, los procesos acceden al recurso estrictamente en orden de llegada, lo cual implica, en particular, que si hay lectores leyendo y un escritor esperando, los lectores que intenten acceder después del escritor no podrán hacerlo hasta que no lo haga dicho escritor.
 54. Varios coches que vienen del norte y del sur pretenden cruzar un puente sobre un río. Solo existe un carril sobre dicho puente. Por lo tanto, en un momento dado, el puente solo puede ser cruzado por uno o más coches en la misma dirección (pero no en direcciones opuestas).
 - (a) Completar el código del siguiente monitor que resuelve el problema del acceso al puente suponiendo que llega un coche del norte (sur) y cruza el puente si no hay otro coche del sur (norte) cruzando el puente en ese momento.

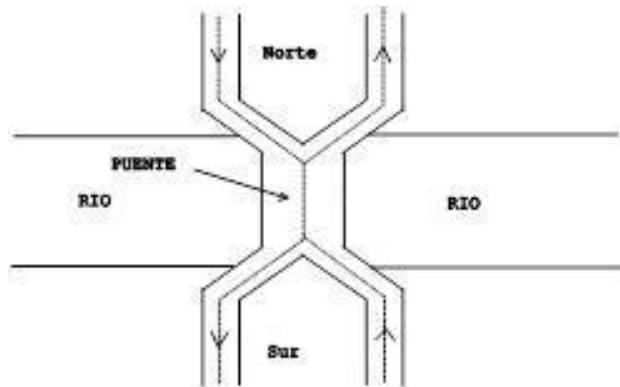


Figura 1: Coches que cruzan un puente

```

Monitor Puente
var ... ;
procedure EntrarCocheDelNorte()
begin
...
end
procedure SalirCocheDelNorte()
begin
...
end
procedure EntrarCocheDelSur()
begin
...
end
procedure SalirCocheDelSur()
begin
...
{ Inicializacion }
begin
...
end

```

- (b) Mejorar el monitor anterior, de forma que la dirección del tráfico a través del puente cambie cada vez que lo hayan cruzado 10 coches en una dirección, mientras 1 ó más coches estuviesen esperando cruzar el puente en dirección opuesta.
55. Una tribu de antropófagos comparte una olla en la que caben M misioneros. Cuando algún salvaje quiere comer, se sirve directamente de la olla, a no ser que ésta esté vacía. Si la olla está vacía, el salvaje despertará al cocinero y esperará a que éste haya llenado la olla con otros M misioneros. Para solucionar la sincronización usamos un monitor llamado Olla, que se puede usar así:

```

monitor Olla ;
...
begin
...
end;
process ProcSalvaje[ i:1..N ] ;
begin

```

```

while true do begin
    Olla.Servirse_1_misionero();
    Comer(); { es un retraso aleatorio }
end
end;
process ProcCocinero ;
begin
    while true do begin
        Olla.Dormir();
        Olla.Rellenar_Olla();
    end
end

```

Diseña el código del monitor Olla para la sincronización requerida, teniendo en cuenta que:

- La solución no debe producir interbloqueo.
- Los salvajes podrán comer siempre que haya comida en la olla.
- Solamente se despertará al cocinero cuando la olla esté vacía.

56. Una cuenta de ahorros es compartida por varias personas (procesos). Cada persona puede depositar o retirar fondos de la cuenta. El saldo actual de la cuenta es la suma de todos los depósitos menos la suma de todos los reintegros. El saldo nunca puede ser negativo. Queremos usar un monitor para resolver el problema.

El monitor debe tener 2 procedimientos: `depositar(c)` y `retirar(c)`. Suponer que los argumentos de las 2 operaciones son siempre positivos, e indican las cantidades a depositar o retirar. El monitor usará la semántica señalar y espera urgente (SU). Se deben de escribir varias versiones de la solución, según las variaciones de los requerimientos que se describen a continuación:

- (a) Todo proceso puede retirar fondos mientras la cantidad solicitada c sea menor o igual que el saldo disponible en la cuenta en ese momento. Si un proceso intenta retirar una cantidad c mayor que el saldo, debe quedar bloqueado hasta que el saldo se incremente lo suficiente (como consecuencia de que otros procesos depositen fondos en la cuenta) para que se pueda atender la petición. Hacer dos versiones: (a.1) colas normales FIFO sin prioridad y (a.2) con colas de prioridad.
- (b) El reintegro de fondos a los clientes se hace únicamente según el orden de llegada, si hay más de un cliente esperando, sólo el primero que llegó puede optar a retirar la cantidad que desea, mientras esto no sea posible, esperarán todos los clientes, independientemente de cuanto quieran retirar los demás. Por ejemplo, suponer que el saldo es 200 unidades y un cliente está esperando un reintegro de 300 unidades, entonces si llega otro cliente debe esperarse, incluso si quiere retirar 200 unidades. De nuevo, resolverlo utilizando dos versiones: (a.1) colas normales (FIFO) sin prioridad y (a.2) con colas de prioridad.

57. Los procesos P_1, P_2, \dots, P_n comparten un único recurso R , pero sólo un proceso puede utilizarlo cada vez. Un proceso P_i puede comenzar a utilizar R si está libre; en caso contrario, el proceso debe esperar a que el recurso sea liberado por otro proceso. Si hay varios procesos esperando a que quede libre R , se concederá al proceso que tenga mayor prioridad. La regla de prioridad de los procesos es la siguiente: el proceso P_i tiene prioridad i , (con $1 \leq i \leq n$), donde los números menores implican mayor prioridad (es decir, si $i < j$, entonces P_i pasa por delante de P_j). Implementar un monitor que implemente los procedimientos `Pedir(...)` y `Liberar()`.

58. En un sistema hay dos tipos de procesos: A y B. Queremos implementar un esquema de sincronización en el que los procesos se sincronizan por bloques de 1 proceso del tipo A y 10 procesos del tipo B. De acuerdo con este esquema:

- Si un proceso de tipo A llama a la operación de sincronización, y no hay (al menos) 10 procesos de tipo B bloqueados en la operación de sincronización, entonces el proceso de tipo A se bloquea.
- Si un proceso de tipo B llama a la operación de sincronización, y no hay (al menos) 1 proceso del tipo A y 9 procesos del tipo B (aparte de él mismo) bloqueados en la operación de sincronización, entonces el proceso de tipo B se bloquea.
- Si un proceso de tipo A llama a la operación de sincronización y hay (al menos) 10 procesos bloqueados en dicha operación, entonces el proceso de tipo A no se bloquea y además deberán desbloquearse exactamente 10 procesos de tipo B.
- Si un proceso de tipo B llama a la operación de sincronización y hay (al menos) 1 proceso de tipo A y 9 procesos de tipo B bloqueados en dicha operación, entonces el proceso de tipo B no se bloquea y además deberán desbloquearse exactamente 1 proceso del tipo A y 9 procesos del tipo B.
- No se requiere que los procesos se desbloqueen en orden FIFO.

Implementar un monitor (con semántica SU) que implemente procedimientos para llevar a cabo la sincronización requerida entre los diferentes tipos de procesos. El monitor puede exportar una única operación de sincronización para todos los tipos de procesos (con un parámetro) o una operación específica para los de tipo A y otra para los de tipo B.

59. El siguiente monitor (Barrera2) proporciona un único procedimiento de nombre entrada, que provoca que el primer proceso que lo llama sea suspendido y el segundo que lo llama despierte al primero que lo llamó (a continuación ambos continúan), y así actúa cíclicamente. Obtener una implementación de este monitor usando semáforos.

```

Monitor Barrera2 ;
var n : integer; { num. de proc. que han llegado desde el signal }
s : condicion ; { cola donde espera el segundo }
procedure entrada() ;
begin
    n := n+1 ; { ha llegado un proceso mas }
    if n < 2 then { si es el primero: }
        s.wait() { esperar al segundo }
        else begin { si es el segundo: }
            n := 0; { inicializa el contador }
            s.signal() { despertar al primero }
        end
    end
{ Inicializacion }
begin
    n := 0 ;
end

```

60. Este es un ejemplo clásico que ilustra el problema del interbloqueo, y aparece en la literatura con el nombre de el problema de los *filósofos-comensales*. Se puede enunciar como se indica a continuación: sentados a una mesa están cinco filósofos, la actividad de cada filósofo es un ciclo sin fin de las operaciones de pensar y comer; entre cada dos filósofos hay un tenedor y para poder comer, un filósofo necesita obligatoriamente dos tenedores: el de su derecha y el de su izquierda.

Se han definido cinco procesos concurrentes, cada uno de ellos describe la actividad de un filósofo. Los procesos usan un monitor, llamado `MonFilo`. Antes de comer cada filósofo debe disponer de su tenedor de la derecha y el de la izquierda, y cuando termina la actividad de comer, libera ambos tenedores. El *filósofo i* alude al tenedor de su derecha como el *número i*, y al de su izquierda como el *número i + 1 mód 5*. El monitor `MonFilo` exportará dos procedimientos: `coge_tenedor(num_tenedor, num_proceso)` y `libera_tenedor(num_tenedor)` para indicar que un proceso filósofo desea coger un tenedor determinado.

El código del programa (sin incluir la implementación del monitor) es el siguiente:

```
monitor MonFilo ;
...
procedure coge_tenedor( num_ten, num_proc : integer );
...
procedure libera_tenedor( num_ten : integer );
...
begin
...
end
process Filosofo[ i: 0..4 ] ;
begin
  while true do begin
    MonFilo.coge_tenedor(i,i); { argumento 1=código tenedor }
    MonFilo.coge_tenedor(i+1 mod 5,i); { argumento 2=numero de proceso }
    comer();
    MonFilo.libera_tenedor(i);
    MonFilo.libera_tenedor(i+1 mod 5);
    pensar();
  end
end
```

Con este interfaz para el monitor, responde a las siguientes cuestiones:

- Diseña una solución para el monitor `MonFilo`
- Describe la situación de interbloqueo que puede ocurrir con la solución que has escrito antes.
- Diseña una nueva solución, en la cual se evite el interbloqueo descrito, para ello, esta solución no debe permitir que haya más de cuatro filósofos simultáneamente intentando coger su primer tenedor.

GIIM. Relación de problemas-Grupo A. Tema 2-2. 11/11/2021

71. Un algoritmo para el cual sólo pudiésemos demostrar que cumple las 4 condiciones de Dijkstra, ¿qué tipo de propiedades concurrentes satisfacería: (a) seguridad, (b) vivacidad, (c) equidad? Justificar las respuestas.
72. Diseñar una solución hardware basada en espera ocupada para el problema de la exclusión mutua utilizando la instrucción máquina swap(x, y) (en lugar de usar LeerAsignar) cuyo efecto es intercambiar los dos valores lógicos almacenados en las posiciones de memoria x e y.
73. Supongamos que tres procesos concurrentes acceden a dos variables compartidas (x e y) según el siguiente esquema:

```
var x, y : integer ;
```

```
//accede a x
process P1 ;
begin
    while true do
        begin
            x := x+1 ;
            // ....
        end
end
```

```
//accede a x e y
process P2 ;
begin
    while true do
        begin
            x := x+1 ;
            y := x ;
            // ....
        end
end
```

```
//accede a y
process P3 ;
begin
    while true do
        begin
            y := y+1 ;
            // ....
        end
end
```

con este programa como referencia, realiza estas dos actividades:

- (a) utilizando un único semáforo para exclusión mutua, completa el programa de forma que cada proceso realice todos sus accesos a x e y sin solaparse con los otros procesos (ten en cuenta que el proceso 2 debe escribir en y el mismo valor que acaba de escribir en x).
- (b) la asignación $x := x+1$ que realiza el proceso 2 puede solaparse sin problemas con la asignación $y := y+1$ que realiza el proceso 3, ya que son independientes. Sin embargo, en la solución anterior, al usar un único semáforo, esto no es posible. Escribe una nueva solución que permita el solapamiento descrito, usando dos semáforos para dos secciones críticas distintas (las cuales, en el proceso 2, aparecen anidadas).
74. En algunas aplicaciones es necesario tener exclusión mutua entre procesos con la particularidad de que puede haber como mucho n procesos en una sección crítica, con n arbitrario y fijo, pero no necesariamente igual a la unidad, sino posiblemente mayor. Diseña una solución para este problema basada en el uso de espera ocupada y cerrojos. Estructura dicha solución como un par de subrutinas (usando una misma estructura de datos en memoria compartida), una para el protocolo de entrada y otro el de salida, e incluye el pseudocódigo de las mismas.
75. Sean los procesos P1, P2 y P3, cuyas secuencias de instrucciones son las que se muestran en el cuadro.

```
//variables globales
```

<pre>process P1 ; begin while true do begin a b c end end</pre>	<pre>process P2 ; begin while true do begin d e f end end</pre>	<pre>process P3 ; begin while true do begin g h i end end</pre>
---	---	---

Resolver los siguientes problemas de sincronización (son independientes unos de otros):

- (a) P2 podrá pasar a ejecutar e solo si P1 ha ejecutado a o P3 ha ejecutado g.
 - (b) P2 podrá pasar a ejecutar e solo si P1 ha ejecutado a y P3 ha ejecutado g.
 - (c) Solo cuando P1 haya ejecutado b, podrá pasar P2 a ejecutar e y P3 a ejecutar h.
 - (d) Sincroniza los procesos de forma que las secuencias b en P1, f en P2, y h en P3, sean ejecutadas como mucho por dos procesos simultáneamente.
76. El cuadro que sigue nos muestra dos procesos concurrentes, P1 y P2, que comparten una variable global x (las restantes variables son locales a los procesos).

<pre>//variables globales</pre>

<pre>process P1 ; var m : integer ; begin while true do begin m := 2*x-n ; print(m); end end</pre>	<pre>process P2 var d : integer ; begin while true do begin d := leer_teclado(); x := d-c*5 ; end end</pre>
--	---

- (a) Sincronizar los procesos para que P1 use todos los valores x suministrados por P2.
 - (b) Sincronizar los procesos para que P1 utilice un valor sí y otro no de la variable x, es decir, utilice los valores primero, tercero, quinto, etc...
77. ¿Podría pensarse que una posible solución al problema de la exclusión mutua, sería el siguiente algoritmo que no necesita compartir una variable turno entre los 2 procesos? Demostrar (sí o no) se satisfacen las siguientes propiedades:
- (a) ¿la exclusión mutua? (propiedad de seguridad)
 - (b) ¿la ausencia de interbloqueo? (propiedad de alcanzabilidad)

<pre>//variables compartidas y valores iniciales var b0 : boolean := false; //true si P0 quiere acceder o esta en SC b1 : boolean := false ; //true si P1 quiere acceder o esta en SC</pre>

```

Process P0 ;
begin
    while true do begin
        //protocolo de entrada:
        b0 := true ; //indica quiere
                     entrar
        while b1 do begin //si el
                           otro tambien:
            b0 := false ;//cede
                           temporalmente }
            while b1 do begin end
                           //espera
            b0 := true ; //
                           vuelve a cerrar
                           paso
        end
        //seccion critica ....
        //protocolo de salida
        b0 := false ;
        //resto sentencias ....
    end
end

```

```

Process P1 ;
begin
    while true do begin 3
        //protocolo de entrada:
        b1 := true ; //indica quiere
                     entrar
        while b0 do begin //si el
                           otro tambien:
            b1 := false ; //cede
                           temporalmente
        while b0 do begin end // esperas
            b1 := true ;//vuelve
                           a cerrar paso
        end
        //seccion critica ....
        //protocolo de salida
        b1 := false ;
        //resto sentencias ....
    end
end

```

78. Al siguiente algoritmo se le conoce como solución de Hyman al problema de la exclusión mutua (fue publicado en una revista de impacto en 1966¹). ¿Es correcta dicha solución?

```

//variables compartidas y valores iniciales
var c0 : integer := 1 ; c1 : integer := 1 ; turno : integer := 1 ;

```

```

process P0 ;
begin
    while true do begin
        c0 := 0 ;
        while turno != 0 do begin
            while c1 = 0 do begin end
            turno := 0 ;
        end
        //seccion critica
        c0 := 1 ;
        //resto sentencias }
    end
end

```

```

process P1 ;
begin
    while true do begin
        c1 := 0 ;
        while turno != 1 do begin
            while c0 = 0 do begin end
            turno := 1 ;
        end
        //seccion critica
        c1 := 1 ;
        //resto sentencias
    end
end

```

79. Supongamos el algoritmo de exclusión mutua que expresamos a continuación. Tenemos los procesos: 0, 1, ... n - 1. Cada proceso i tiene una variable $s[i]$, inicializada a 0, que puede tomar los valores 0/1. El proceso i puede entrar en la sección crítica si:

$$s[i] \neq s[i-1] \text{ para } i > 0;$$

$$s[0] = s[n-1] \text{ para } i = 0;$$

Tras ejecutar su sección crítica, el proceso i deberá hacer:

$$s[i] = s[i-1] \text{ para } i > 0;$$

$$s[0] = (s[0] + 1) \bmod 2 \text{ para } i == 0;$$

¹Harris Hyman, "Comments on a problem in concurrent programming control", Communications of the ACM, v.9 n.1, p.45, 1966

80. Se tienen 2 procesos concurrentes que representan 2 máquinas expendedoras de tickets (señalan el turno en que ha de ser atendido el cliente), los números de los tickets se representan por dos variables n1 y n2 que valen inicialmente 0. El proceso con el número de ticket más bajo entra en su sección crítica. En caso de tener 2 números iguales se procesa primero el proceso número 1.
- Demostrar que se verifica la ausencia de interbloqueo (propiedad de *alcanzabilidad* de la sección crítica), la ausencia de inanición (propiedad de *vivacidad*) y la exclusión mutua (una propiedad de *seguridad*).
 - Demostrar que las asignaciones n1 := 1 y n2 := 1 son ambas necesarias.

```
//variables compartidas y valores iniciales
var n1 : integer := 0 ; n2 : integer := 0 ;
```

```
process P1 ;
begin
  while true do begin
    n1 := 1 ; // E1.1
    n1 := n2+1 ; // L1.1 ; E2.1
    while n2 != 0 and // L2.1
      n2 < n1 do begin end; //L3.1
    // sección crítica } { SC.1 }
    n1 := 0 ; // E3.1
    // resto sentencias { RS.1 }
  end
end
```

```
process P2 ;
begin
  while true do begin
    n2 := 1 ; // E1.2
    n2 := n1+1 ; // L1.2 ; E2.2
    while n1 != 0 and //L2.2
      n1<= n2 do begin end; //L3.2
    // sección crítica { SC.2 }
    n2 := 0 ; // E3.2
    // resto sentencias { RS.2 }
  end
end
```

81. El siguiente programa es una solución al problema de la exclusión mutua para 2 procesos. Discutir la corrección de esta solución: si es correcta, entonces probarlo. Si no fuese correcta, escribir escenarios que demuestren que la solución es incorrecta.

```
//variables compartidas y valores iniciales
var c0 : integer := 1; c1 : integer := 1 ;
```

```
process P0 ;
begin
  while true do begin
    repeat
      c0 := 1-c1 ;
    until c1 != 0 ;
    // sección crítica
    c0 := 1 ;
    //resto sentencias }
  end
end
```

```
process P1 ;
begin
  while true do begin
    repeat
      c1 := 1-c0 ;
    until c0 != 0 ;
    // sección crítica
    c1 := 1 ;
    // resto sentencias
  end
end
```

82. Con respecto al algoritmo de Peterson para N–procesos: ¿sería posible que llegaran 2 procesos a la etapa N-2, 0 procesos a la etapa N-3 y en todas las etapas anteriores existiera al menos 1 proceso? Justificar la respuesta.
83. En el *algoritmo de Peterson* para N procesos y considerando cualquier escenario de ejecución de dicho algoritmo, el número máximo de turnos que tiene que esperar cualquier proceso para entrar en sección crítica es N-1 turnos.

84. Con respecto al algoritmo de la siguiente figura (algoritmo de Dijkstra para N procesos), demostrar la falsedad de la siguiente proposición: *si un conjunto de procesos está intentando pasar simultáneamente el primer bucle (5), y el proceso que tiene el turno está pasivo, entonces siempre conseguirá entrar primero en sección crítica el proceso de dicho grupo que consiga asignar la variable turno en último lugar.*

```

var turn :0..N-1;
flag: array [0 .. N-1] of (pasivo,
solicitando, enSC);
flag:= pasivo;
Process P(i);
begin
(1)   <resto instrucciones>
(2)   repeat
(3)       flag[i] := solicitando;
(4)       j := turn;
(5)       while (turno !=i) do
(6)           if (flag[turno] =  pasivo) then
(7)               turn:=i;
(8)           endif;
(9)       enddo;
(10)      flag[i] := enSC;
(11)      j := 0;
(12)      while ( j < N )and
( (j = i)or flag[j]!= enSC )do
(13)          j := j + 1;
(14)      enddo;
(15) until (j >= N);
<<sección critica>>
(16) flag[i] := pasivo;
end

```

85. El algoritmo de la figura siguiente (algoritmo de Knuth para N-procesos) resuelve el problema de la exclusión mutua para N-procesos, para lo cual utiliza N variables booleanas,

flag: array[0..N - 1] of (solicitando, enSC, pasivo);
una variable turn: 0..n - 1 y la variable local j.

- (a) Demostrar que el algoritmo de Knuth verifica todas las propiedades exigibles a un programa concurrente, incluyendo la de equidad.
- (b) Escribir un escenario en el que 2 procesos consiguen pasar el bucle de la instrucción (5), suponiendo que el turno lo tiene inicialmente el proceso p (0) .

```

var flag: array [0 .. N-1] of (pasivo,
solicitando, enSC);
flag:= pasivo;
turn := 0;
Process P(i);
begin
(1)   <resto instrucciones>
(2)   repeat
(3)       flag[i] := solicitando;
(4)       j := turn;
(5)       while (j !=i) do
(6)           if flag[j] !=  pasivo then

```

```

(7)           j := turn
(8)           else j := ( j - 1 ) mod N;
(9)           endif;
(10)          enddo;
(11)          flag[i] := enSC;
(12)          j := 0;
(13)          while ( j < N ) and
( (j = i) or flag[j] != enSC ) do
( j := j + 1;
(14)          enddo;
(15) until (j >= N);
(16) turn := i;
<<sección critica>>
(17) j := (turn + 1) mod N;
(18) turn := j;
(19) flag[i] := pasivo;
end

```

86. Si en el algoritmo de Dijkstra se cambia la instrucción (6) por esta otra: if flag[turno] !=SC, entonces el algoritmo dejaría de ser correcto. Indicar qué propiedad(es) de corrección fallaría(n) y justificar por qué.
87. Si en el algoritmo de Knuth se hacen las siguientes sustituciones:
- La condición de la instrucción until de (15) por la condición: (j >= N) and (turno=i or flag[turno]= pasivo)
 - Se inserta el siguiente bucle después de la instrucción (17):

```

while (j !=turn) and( flag[j]=pasivo) do
(19)      j := j + 1;
(20)  enddo;

```
- (a) Verificar las propiedades de exclusión mutua, alcanzabilidad de la sección crítica, vivacidad y equidad del algoritmo.
 - (b) Calcular el número de turnos máximo que puede llegar a tener que esperar un proceso que quiera entrar en su sección crítica con el algoritmo anterior.
88. Demostrar que las instrucciones (13)-(16) del algoritmo de exclusión mutua distribuido de Ricart-Agrawala no necesitan ser protegidas dentro de la sección crítica definida por las operaciones wait(), signal() del semáforo "s".

```

ns: 0..+INF;
mns: 0..INF;
numrepesperadas:0..n-1;
intentaSC: boolean;
prioridad: boolean;
repretrasadas: array[1..N] of boolean;

```

```

Process Pi;
begin
(1) wait(s);
(2) intentaSC:= TRUE;
(3) ns:= mns +1;
(4) numrepesperadas:=n-1;
(5) signal(s);
(6) for j:= 1 to n do
(7) if j <> i then
(8) send(j, pet, ns, i);
(9) wait(sinc);
<<seccion critica>>
(10) wait(s);
(11) intentaSC:= FALSE;
(12) signal(s);
(13) for j:=1 to n do
(14) if repretrasadas[j] then begin
(15) repretrasadas[j]:= FALSE;
(16) send(j, rep);
(17) end;
end

Process Pet(i);
begin
(1) receive(pet, k, j);
(2) wait(s);
(3) mns:= max(mns, k);
(4) prioridad:= (intentaSC) and (
    k>ns or (k=ns and i<j));
(5) if prioridad then
    repretrasadas[j]:= TRUE
    else send(j, rep);
(6) signal(s);
end

Process Rep(i);
begin
(1) receive(rep);
(2) numrepesperadas:= numrepesperadas
    - 1;
(3) if numrepesperadas= 0 then signal
    (sinc);
end

```

89. Suponer que el algoritmo de Suzuki-Kasami para resolver el problema de la exclusión mutua distribuida para n-procesos se modifica como aparece en la siguiente figura. Explicar por qué dejaría de ser correcto el algoritmo, relacionándolo con cada una de las propiedades de corrección que se demuestran para el algoritmo original.

```

token_presente:boolean:=FALSE;
enSC:boolean:= FALSE;
peticion:array[1..n] of boolean:= FALSE;
//En el algoritmo original ->peticion: array[1..N] of 0..+INF
//ademas se declara otro array-> token: array[1..N] of 0..+INF

```

```

Process P(i);
begin
(0) wait(s);
(1) if NOT token_presente then begin
(2) broadcast(pet, i);
(3) receive(acceso);
(4) token_presente:= TRUE;
(5) end;
(6) enSC:= TRUE;
(7) signal(s);
<<seccion critica>>
(8) enSC:=FALSE;
(9) wait(s);
(10) for j:= i+1 to n, 1 to i-1 do
(11) if peticion[j] and
    token_presente then begin
(12) token_presente:= FALSE;
(13) send(j, acceso);
(14) peticion[j]:= FALSE;
(15)end;
(16) signal(s);
end

```

```

Process Pet(i);
begin
(1) receive(pet, j);
(2) wait(s);
(3) peticion[j]:= TRUE;
(4) if token_presente and NOT
    enSC then
<<< repetir (10)- (16) >>>
end

```

5) La propiedad de reentrancia asegura que cada proceso ve su propia copia de las variables y el contenido del programa. Esto se debe a que durante la ejecución de los procedimientos del monitor, un proceso puede verse interrumpido y puesto en espera bloqueada hasta que se cumpla una determinada condición. Una vez que esta condición se verifique, el proceso reanuda la ejecución del procedimiento, recuperando la información (contexto).

5) a)

Previamente, tenemos que realizar una serie de suposiciones:

- $N_1 \geq 0$ y $N_2 \geq 0$ (N_1 suministros tipo 1, N_2 suministros tipo 2)
- Tipos de suministros = {Tipo 1, Tipo 2}

Tenemos representar la situación usando un array de longitud 2, de forma que la posición 0 almacene el valor de N_1 y la posición 1, el de N_2 . creamos una variable condición por cada tipo de suministro, ya que el problema nos da la restricción de que en ningún momento puede haber un ejemplo de un tipo de recurso en uso por más de un proceso (Esta condición equivale a que no haya valores negativos)

Monitor - Demanda-SV {

```

    var libros : array [1..2] of integer; // Inicializado a  $N_1$  y  $N_2$  respectivamente
    cola : array [1..2] of condition;

    procedure pedirRecurso (tipo: integer)
    begin
        if libros[tipo] == 0 then // Si no queda el tipo de suministro solicitado
            cola[tipo].wait(); // Se bloquea
        libros[tipo] -= 1; // En caso contrario, se otorga el suministro
    end

    procedure liberarRecursos (tipo: integer)
    begin
        libros[tipo] += 1; // Devuelve el recurso
        cola[tipo].signal(); // Si hubiera algún proceso esperando por el recurso, lo desbloquea
    end

```

b) Dos procesos intentando acceder a los dos recursos secuencialmente (en distinto orden)

Proceso P1	Proceso P2
<pre> begin Monitor. pedirRecurso(1); Monitor. pedirRecurso(2); end </pre>	<pre> begin Monitor. pedirRecurso(2); Monitor. pedirRecurso(1); end </pre>

c) Añadimos un nuevo caso a la variable condición y modificamos el método pedir recurso.

```

cola: array [0..2] of condition;
procedure pedirRecurso (tipo: integer)
begin
    if (tipo == 0) then // Nuevo caso (se pone primero porque tiene más prioridad)
        if libros[1] == 0 ó libros[2] == 0 // Espera a que haya de los dos tipos de recursos
            cola[0].wait();
    libros[1] -= 1; // Se le otorga los recursos
    libros[2] -= 1;
    else // Mismo caso que antes
        if libros[tipo] == 0
            cola[tipo].wait();
    libros[tipo] -= 1;
end

procedure liberarRecursos (tipo: integer)
begin
    if (tipo == 0)
        libros[1] += 1; libros[2] += 1;
    cola[0].signal();

```

```

este
libres [tipoT++]
cda [tipoT].signal;
end

```

② (Problema escritor - consumidor)

Comentemos haciendo una serie de suposiciones:

- Los lectores se pueden ejecutar concurrentemente entre ellos, pero se excluyen con los escritores.
- Los escritores se excluyen de los lectores y otros escritores.

a) Los lectores tienen prioridad

```

Monitor LectoresEscritores {
    var escribiendo : boolean;           // Inicializada a false
    num_lectores: integer;              // Inicializada a 0
    lectores, escritores: condition;

    procedure inicia_escritura()
    begin
        if escribiendo or num_lectores > 0      // Si hay lectores leyendo o otro escritor escribiendo se bloquea
            escritores.wait();
        escribiendo = true;                    // En caso contrario comienza a escribir
    end;

    procedure termina_escribir()
    begin
        escribiendo = false;                  // Termina de escribir
        if lectores.queue() then             // como los lectores tienen prioridad, si hay lectores a la espera los desbloqueamos
            lectores.signal();
        else                                // En caso contrario desbloqueamos un escritor
            escritores.signal();
    end;

    procedure inicia_lector()
    begin
        if escribiendo then                // Si hay un escritor escribiendo, se bloquea
            lectores.wait();
        num_lectores++;                   // como los lectores no se excluyen entre ellos, pueden leer varios a la vez
        lectores.signal();
    end;

    procedure fin_lector()
    begin
        num_lectores--;                  // Lectur dpa de leer
        if num_lectores == 0 then         // Si no quedan lectores, se desbloquea un escritor
            escritores.signal();
    end;
}

```

b) (Prioridad escritores)

Si en igual, pero en procedure termina_escribir() haremos la siguiente modificación:

```

...
if lectores.queue() then
    lectores.signal();
else
    escritores.signal();
...

```



```

...
if escritores.queue() then
    escritores.signal();
else
    lectores.signal();
...

```

Si lo quisieras sin prioridad, bastaría con eliminar esta sección de código

④ Monitor Piente {

```
var N-cruzando, S-cruzando: Integer; // se inicializan a 0
    norte, sur: condition;
procedure EntrarCocheDelNorte()
begin
    if S-cruzando > 0 then      // Si hay coches del sur cruzando, te espera
        norte.wait();
    N-cruzando++;
    norte.signal();
end;

procedure SalirCocheNorte()
begin
    N-cruzando--;
    if N-cruzando == 0 then    // Si no hay más coches del norte cruzando, se deja paso a los del sur.
        sur.signal();
end;
```

Análogo para los del sur

Ahora, la segunda parte nos pide que, si han cruzado 10 coches de un sentido, se permita el paso en el otro sentido, luego modificamos la condición

Monitor Piente {

```
var N-cruzando, S-cruzando: Integer;
    norte, sur: condition;
    N-pueden, S-pueden: Integer // Inicializadas a 10
procedure EntrarCocheDelNorte()
begin
    if S-cruzando > 0 or N-pueden == 0
        norte.wait();
    N-cruzando++;
    if sur.gueuel() then N-pueden--;
    if N-pueden > 0
        norte.signal();
end;

procedure SalirCocheDelNorte()
begin
    N-cruzando--;
    if N-cruzando == 0
        S-pueden = 10;
    sur.signal();
end;
```

Análogo para sur

55 Necesitamos 2 variables condición, una para el cocinero y otra para los salvajes:

```
Monitor Olla;
var num-misioneros: integer;
    salvajes, cocinero: condition;
procedure ServirAlMisionero()
begin
    if num-misioneros == 0 then
        cocinero.signal();
    salvajes.wait();
    num-misioneros--;
    if (num-misioneros > 0) then
        salvajes.signal();
    else
        cocinero.signal();
    end;
end;
```

```
procedure Dormir()
begin
    if num-misioneros > 0 then
        cocinero.wait();
    end;
procedure PonerOllaLlena()
begin
    num-misioneros = M;
    salvajes.signal();
    end;

```

56

Monitor Cuenta; // Implementación FIFO sin prioridad

```
var saldo: integer;
    cola: condition;
procedure retirar (cantidad: positive integer)
begin
    while cantidad > saldo do
        cola.signal();
        cola.wait();
        saldo -= cantidad;
        cola.signal();
    end
end;
```

```
procedure depositar (c: positive integer)
begin
    saldo += cantidad;
    cola.signal();
end
```

como es sincrónico SJ, simula rotar el proceso hacia atrás
combiendo cola.wait() por
cola.wait(cantidad) tendríamos la situación para colas con prioridad

Monitor Cuenta; // Orden de llegada sin prioridad

```
var saldo : integer;
    ventanilla, resto: condition;
procedure retirar (c: positive integer) begin
    if (ventanilla = open) then // Si hay un cliente siendo atendido
        resto.wait();           // el resto espera
    while cantidad > saldo do
        ventanilla.wait();
        saldo -= cantidad;
        resto.signal();
    end
procedure depositar (c: positive integer) begin
    saldo += cantidad;
    ventanilla.signal();
end
```



Concepto fundamental

- La cola de entrada al monitor la controlamos con un semáforo mutex.
- La cola de procesos urgentes controlada por el semáforo next.
- El número de procesos urgentes encola se contabiliza con la variable next-count.
- Las colas ordinarias vendrán controladas por el semáforo x-sem y el número de procesos por x-sem-count

Monitor:

```
var n: integer;
    s: Semaphore=0;
    mutex: Semaphore=1;
procedure Entrada()
begin
    sem-wait(mutex);
    n:=i;
    if (n<2) then
        sem-signal(mutex);
        sem-wait(s);
    else
        n=0;
        sem-signal(s);
        sem-signal(mutex);
```

74

Comencemos recordando las cuatro condiciones de Dijkstra son:

- 1) No hacer suposiciones sobre las instrucciones o el número de procesos.
- 2) No hacer suposiciones sobre la velocidad de ejecución de los procesos, salvo que es mayor que 0.
- 3) Un proceso que no esté en la S.C. no puede impedir que otro proceso entre a la misma.
- 4) La S.C. siempre será alcanzada por alguno de los procesos que intentan entrar en la misma.

La **seguridad** se verifica, puesto que las 4 condiciones existen para garantizar la **alcanzabilidad** y la **exclusión mutua**.

La **llegada** no se cumple, puesto que la **llegada** \Rightarrow no iniciación + la 4^a condición solo garantiza **alcanzabilidad**, pero no nos garantiza la **no-iniciación**.

La **equidad** no se garantiza puesto que no se garantiza la **no-iniciación**.

82

Falso, porque el teorema 3 nos indica que si comienzan, al menos, dos procesos en una etapa \Rightarrow Hay, al menos, un proceso en cada etapa anterior

- $j=0$ trivial (Todos los procesos están ahí)
- $j=1$

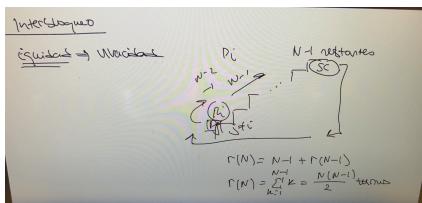
Supongamos dos procesos, P_1 y P_2 en la etapa $j=1$ y que P_2 ha sido el último en salir. Para que un proceso pueda pasar a la etapa $j+1$ verificar que no precede a ningún otro proceso o que no ha sido el último en llegar a la etapa j . En este caso, P_2 precedía a P_1 , luego, para poder pasar de $j=0 \rightarrow j=1$ no puede ser el último proceso en ingresar en la etapa $j=0 \Rightarrow$ Existe, al menos, un proceso en la etapa 0, que permite avanzar a P_2 a la etapa $j=1$.

• Supongamos cierto para $j=N-3$ y veremos para $N-2$. Supongamos que P_1 se encontraba en $j=N-2$ y P_2 avanza a este estado, haciendo que haya dos procesos en el estado $j=N-2$. Necesitamos, para que P_2 haya podido avanzar, su estado no debe ser precedido por el de ningún otro proceso (lo cual es falso, ya que está P_1) o no debe ser el último en llegar al estado $N-3 \Rightarrow$ Existe al menos otro proceso en $j=N-3$, pero P_2 estaba en $j=N-3$ (aplicando inducción) existe un proceso en los estados anteriores.

83 Falso, porque si un proceso, que no es el último, consigue superar las barreras (5) y (6) y asignar su flag a S.C en (10), será el primer proceso en entrar en S.C., ya que el resto quedarán bloqueados por no verificar (6).

84 El algoritmo deja de ser correcto porque cuando varios procesos superen la primera barrera (el while), será el último valor el que pise la variable turno.

Supongamos un escenario en el que el resto de los procesos siguen en el segundo bucle y con el repetit vuelven al primer while sin que el proceso que asignó turno haya ejecutado la asignación del flag, entonces la clave del flag del proceso que tiene el turno \neq nsc, lo que permitirá a estos procesos avanzar, cambiar el turno y producir una situación de interbloqueo.



Probar las propiedades de concurrencia equivale a probar:

- Exclusión Mutua
- Alcanzabilidad
- Asociatividad de Inclusión

(Puede producirse dentro de la ejecución \Rightarrow seguridad \Rightarrow llegada)

85 Comencemos probando que cumple las propiedades de un programa concurrente:

- **Exclusión-Mutua**: El bucle de la línea (8) sirve como barrera para comprobar que no hay ningún otro proceso en sección crítica.
- **Alcanzabilidad de la S.C.**: Cualquier proceso que cambie su flag a saliendo llegará a entrar en la S.C. en el momento que le llegue el turno.
- **No-iniciación y equidad**: Gracias a que el turno solo se pasa en un sentido, un proceso que acaba de salir de la S.C. no podrá volver a entrar (cuando sea más rápido y cambie nuevamente su flag a saliendo) hasta comprobar el resto de procesos. Si todos estén positivos, entonces podrá entrar, si no, tendrá que esperar a la rotación del turno.



Para el segundo apartado supongamos que los procesos 2, 3, 4, 5, 6, 7 y 8 ejecuta primero el que no posee el turno:

Rotación	i	c101	c111	turno
inicio	-	positivo	positivo	0
P1(3,4,5,6,8)	1	positivo	saliendo	0
P2(5,1)	1	positivo	SC	0
P0(3,4,5,9,11)	0	SC	SC	0

⑦

```

...
Process P(i); begin
(1)   <resto instrucciones>
(2)   repeat
(3)     flag[i] := solicitando;
(4)     j := turn;
(5)     while (j != i) do
(6)       if flag[j] != pasivo then
(7)         j := turn
(8)       else j := (j - 1) mod N;
(9)       endif;
(10)    enddo;
(11-12)   flag[i] := enSC; j := 0;
(13-14)   while (j < N) and
( (j = i) or flag[j] != enSC ) do
(15)     j := j + 1; enddo;
(16) until (j >= N) and (turno=i or flag[turno]=pasivo)
(17) turn := i;           No hay ninguno en SC
<<sección crítica>>           sea tu turno
(18) j := (turn + 1) mod N;      y del turno sea
(19) while (j != turno) and (flag[j]=pasivo) do
(20)   j := j + 1;
(21) enddo;
(22) turn := j;
(23) flag[i] := pasivo;
end

```

- La nueva sentencia (16) impide que un proceso que pose el turno (solicitando entrar en S.C.) consiga adelantarse varias veces porque nega a ejecutarse $\text{flag}[i] = \text{enSC}$ antes que el resto de procesos.

- Con las sentencias (18-19) se consigue que un proceso que sale de la S.C. no pueda adelantarse a cualquier otro proceso solicitante más de una vez.

⑧ demostrar el siguiente monitor usando las reglas de corrección con $M = 10 \leq n \leq M+1$

```

Monitor Bufer;
var n, frente, atras: integer;
no_vacio, no_lleno: condition;
buf: array[1..MAX] of tipo_dato;

```

<i>(1)</i> <i>después de la inicialización, debe ser cierto</i> procedure insertar(d: tipo_dato); begin if (atras mod MAX + 1 == frente) then <i>!atras mod MAX+1 == frente A M</i> no_lleno.wait() <i>CO..M+1</i> else NULL; <i>!atras mod MAX+1 == frente A M+1</i> buf[atras] = d; atras = atras mod MAX + 1; n++; <i>frente = atras</i> no_vacio.signal(); <i>!0 < n <= M+1</i> end; <i>!0 < n <= M+1</i>	procedure retirar(var x: tipo_dato); begin if (frente == atras) then no_vacio.wait(); <i>!frente==atras A M+1</i> else NULL; <i>!frente!=atras A M+1</i> x = buf[frente]; frente = frente mod MAX - 1; n--; <i>!atras mod MAX != frente</i> no_lleno.signal(); <i>!0 < n <= M - 1</i> end;
--	--

```

begin
  frente=1;
  atras= 1;
  n= 0;
end;

```