TEMA-2-SINCRONIZACION-EN-MEMORIA...

mrg23

Sistemas Concurrentes y Distribuidos

2º Grado en Ingeniería Informática

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

TEMA 2: SINCRONIZACIÓN EN MEMORIA COMPARTIDA

1. INTRODUCCIÓN A LA SINCRONIZACIÓN EN MEMORIA COMPARTIDA

Estudiaremos soluciones para exclusión mutua y sincronización basadas en el uso de memoria compartida entre los procesos involucrados. La sincronización en memoria compartida es crucial para gestionar el acceso concurrente de múltiples procesos a recursos compartidos.
Este tipo de soluciones se pueden dividir en dos categorías:

- **Soluciones de bajo nivel con espera ocupada** basadas en operaciones de lectura/escritura en memoria compartida y bucles de espera ocupada.
- **Soluciones de alto nivel** proporcionan interfaces de software para bloquear procesos para realizar las esperas requeridas por la sincronización, en lugar de usar espera ocupada.

a. SOLUCIONES DE BAJO NIVEL CON ESPERA OCUPADA

Cuando proceso debe esperar una condición, entra en un bucle que continuamente comprueba condición (espera ocupada). Dos categorías:

- Soluciones software: Usan operaciones estándar sencillas de lectura y escritura de datos simples (valores lógicos, enteros, ...) en memoria compartida,
- Soluciones hardware (cerrojos): Usan instrucciones máquina específicas del repertorio de los procesadores involucrados.

```
process P0 ;
begin
   while true do begin
      p0sc := true ;
      turno0 := false ;
      while p1sc and not turno0 do
         begin end
      { sección crítica }
      p0sc := false ;
      { resto sección }
   end
end
```

```
{ variables compartidas y valores iniciales }
var sc_ocupada : boolean := false ; { true solo s

{ procesos }
process P[ i : 1 .. n ];
begin
   while true do begin
      while TestAndSet( sc_ocupada ) do begin end
      { seccion critica }
      sc_ocupada := false ;
      { resto seccion }
   end
end
```

b. SOLUCIONES DE ALTO NIVEL

Las Soluciones de bajo nivel se prestan a errores, producen algoritmos complicados y consumen más tiempo de CPU (bucles de espera ocupada).
Soluciones de alto nivel: ofrecen interfaces de acceso a estructuras de datos y usan bloqueo de procesos en lugar de espera ocupada. Veremos algunas:

- Semáforos: se construyen directamente sobre las soluciones de bajo nivel, usando además servicios del SO que dan la capacidad de bloquear y reactivar procesos.
- Regiones críticas condicionales: de más alto nivel que los semáforos, y que se pueden implementar sobre ellos.
- Monitores: Soluciones de más alto nivel que las anteriores. Se pueden implementar en algunos lenguajes orientados a objetos como Java, Python o C++.

2. SOLUCIONES SOFTWARE CON ESPERA OCUPADA PARA E.M.

Veremos diversas soluciones algorítmicas para lograr EM en una sección crítica usando:

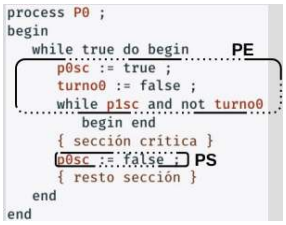
- Variables compartidas entre los procesos involucrados.
- Espera ocupada cuando sea necesario en el protocolo de entrada.

Veremos dos algoritmos para 2 procesos: Algoritmos de Dekker y de Peterson.
Estructura de los procesos en estos algoritmos:

- **Protocolo de entrada (PE)**: Instrucciones que incluyen espera cuando no se pueda acceder sección crítica.
- **Sección crítica (SC)**: Instrucciones que solo pueden ser ejecutadas por un proceso como mucho.
- **Protocolo de salida (PS)**: permiten que otros procesos sepan que este proceso ha terminado SC.
- **Resto de Sentencias (RS)**: Sentencias que no formanparte de las etapas anteriores.

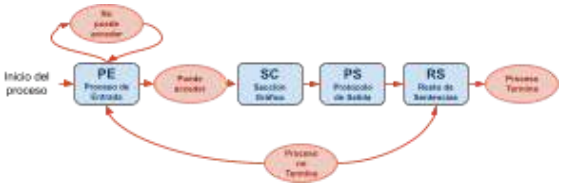
```
PE
p0sc := true ;
turno0 := false ;
while p1sc and not turno0 do
   begin end
{ sección crítica }
p0sc := false ; PS
{ resto sección }
```

a. CONDICIONES COMPORTAMIENTO PROCESOS



Para simplificar análisis, se hacen suposiciones:

- Cada proceso tiene una única SC, formada por un único bloque contiguo de instrucciones.
- Proceso es un bucle infinito que ejecuta dos pasos repetidamente:
  1) Sección crítica (con PE antes y el PS después)
  2) Resto de sentencias: se emplea un tiempo arbitrario no acotado, e incluso el proc. podría finalizar en esta sección.
- No suposición sobre cuántas veces un proceso intenta acceder SC.

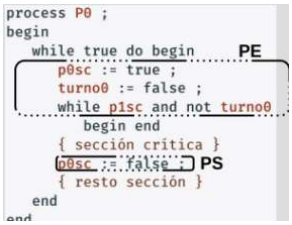


Para implementar soluciones correctas al problema de EM, es necesario suponer que:

Los procesos siempre terminan una SC y la ejecutan en un intervalo de tiempo finito.

Durante ejecución de la SC, el proceso:

- NO finaliza/aborta o es finalizado/abortado externamente.
- NO entra en bucle infinito.
- NO es bloqueado o suspendido indefinidamente de forma externa.



En general, es deseable que el tiempo empleado en SC sea el menor posible.

b. PROPIEDADES REQUERIDAS ALGORITMO EM

Para que un algoritmo para EM sea correcto, se deben cumplir estas tres propiedades mínimas:

1) Exclusión mutua
2) Progreso
3) Espera limitada

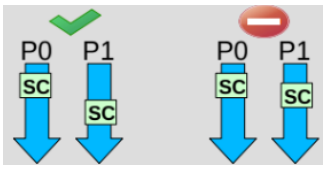
Además, hay propiedades deseables adicionales que también deben cumplirse:

4) Eficiencia
5) Equidad

Si bien consideramos correcto un algoritmo que no sea muy eficiente o para el que no pueda demostrarse claramente la equidad.

c. PROPIEDADES MÍNIMAS REQUERIDAS

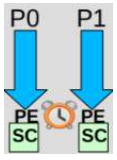
Exclusión Mutua



En cada instante de tiempo, y para cada SC existente, habrá como mucho un proceso ejecutando alguna sentencia de dicha SC.

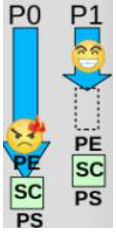
- Propiedad fundamental pero no suficiente.

Propiedad de Progreso

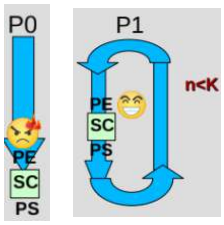


1) Tras intervalo de tiempo finito desde que ingresó el 1er proceso al PE, uno de los procesos en el PE podrá acceder a SC.
   - Si se incumple ⇒ posibilidad interbloqueo.

2) La elección del proceso que accede es completamente independiente del comportamiento de procesos que durante ese intervalo ni han estado en SC ni han intentado acceder.



d. PROPIEDADES MÍNIMAS. ESPERA LIMITADA



Espera Limitada

Supongamos que un proceso emplea un intervalo de tiempo en el PE intentando acceder a SC. Durante ese intervalo, cualquier otro proceso activo puede entrar n veces a ese PE y acceder a SC (incluyendo n = 0). La propiedad establece:

Un algoritmo de EM debe diseñarse de forma que n nunca superará un valor determinado.

- Esperas en PE siempre serán finitas (si procs. emplean tiempo finito en SC).

e. PROPIEDADES DESEABLES. EFICIENCIA Y EQUIDAD

Eficiencia:

Los protocolos de entrada y salida deben emplear poco tiempo de procesamiento (excluyendo esperas ocupadas en PE), y las variables compartidas deben usar poca cantidad de memoria.

Equidad:

Cuando haya varios procesos compitiendo por acceder a SC (de forma repetida en el tiempo), no debería existir posibilidad de que sistemáticamente se perjudique a algunos y se beneficie a otros.

f. REFINAMIENTO SUCESIVO DE DIJKSTRA

El Refinamiento sucesivo de Dijskstra hace referencia a una serie de algoritmos que intentan resolver el problema de EM.

- Comienza con una versión muy simple, incorrecta (no cumple alguna propiedad), y se hacen sucesivas mejoras para intentar cumplir las 3 propiedades mínimas.
- Muestra patologías comunes de estos algoritmos e ilustra muy bien la importancia de las propiedades.
- La versión final correcta se denomina Algoritmo de Dekker.
- Por simplicidad, únicamente algoritmos para 2 procesos .
- Suponemos 2 procesos, P0 y P1, cada uno ejecuta bucle infinito conteniendo: PE, SC, PS y RS.

VERSIÓN 1

Usa variable lógica compartida p01sc que valdrá true solo si algún proceso está en SC.

```
{ variables compartidas y valores iniciales }
var p01sc : boolean := false ; { indica si la SC esta ocupada }
```

```
process P0 ;
begin
   while true do begin
      while p01sc do begin end
      p01sc := true ;
      { sección crítica }
      p01sc := false ;
      { resto sección }
   end
end
```

```
process P1 ;
begin
   while true do begin
      while p01sc do begin end
      p01sc := true ;
      { sección crítica }
      p01sc := false ;
      { resto sección }
   end
end
```

VERSIÓN 2

Se usará una única variable lógica (turno0), cuyo valor indicará cuál debe entrar a SC. Valdrá true si es P0, y false si es P1.

```
{ variables compartidas y valores iniciales }
var turno0 : boolean := true ; { podría ser también false }
```

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

```
process P1 ;
begin
   while true do begin
      while turno0 do begin end
      { sección crítica }
      turno0 := true ;
      { resto sección }
   end
end
```

VERSIÓN 3

Para impedir la alternancia, se usan dos variables lógicas (p0sc, p1sc) en lugar de solo una. Cada variable vale true si el correspondiente proceso está en SC.

```
{ variables compartidas y valores iniciales }
var p0sc  : boolean := false ; { verdadero solo si proc. 0 en SC }
    p1sc  : boolean := false ; { verdadero solo si proc. 1 en SC }
```

```
process P0 ;
begin
   while true do begin
      while p1sc do begin end
      p0sc := true ;
      { sección crítica }
      p0sc := false ;
      { resto sección }
   end
end
```

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

VERSIÓN 4

Para solucionar el problema anterior se puede cambiar el orden de las dos sentencias del PE. Variables lógicas p0sc y p1sc están a true cuando el correspondiente proceso está en SC o intenta entrar a SC desde PE.

```
{ variables compartidas y valores iniciales }
var p0sc  : boolean := falso ; { verdadero solo si proc. 0 en PE o SC }
    p1sc  : boolean := falso ; { verdadero solo si proc. 1 en PE o SC }
```

```
process P0 ;
begin
   while true do begin
      p0sc := true ;
      while p1sc do begin end
      { sección crítica }
      p0sc := false ;
      { resto sección }
   end
end
```

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

VERSIÓN 5

Para solucionarlo, si un proceso ve que el otro quiere entrar, el primero pone su variable temporalmente a false.

```
var p0sc : boolean := false ; { true solo si proc. 0 en PE o SC }
    p1sc : boolean := false ; { true solo si proc. 1 en PE o SC }
```

```
process P0 ;
begin
   while true do begin
      p0sc := true ;
      while p1sc do begin
        p0sc := false ;
        { espera durante un tiempo }
        p0sc := true ;
      end
      { sección crítica }
      p0sc := false ;
      { resto sección }
   end
end
```

```
process P1 ;
begin
   while true do begin
      p1sc := true ;
      while p0sc do begin
        p1sc := false ;
        { espera durante un tiempo }
        p1sc := true ;
      end
      { sección crítica }
      p1sc := false ;
      { resto sección }
   end
end
```

g. ALGORITMO DE DEKKER

El algoritmo de Dekker, es un algoritmo correcto (cumple propiedades mínimas).

- Se puede considerar el resultado final del refinamiento de Dijkstra:
- Incorpora espera de cortesía en cada proceso (como versión 5), durante la cual cede al otro la posibilidad de acceder SC, cuando ambos coinciden en PE.
- Evita interbloqueos mediante variable turno: la espera de cortesía solo la realiza uno de los dos procesos de forma alterna.
  - Variable turno permite también detectar final de espera de cortesía, implementada mediante espera ocupada.

```
{ variables compartidas y valores iniciales }
var p0sc    : boolean := falso ; { true solo si proc.0 en PE o SC }
    p1sc    : boolean := falso ; { true solo si proc.1 en PE o SC }
    turno0  : boolean := true  ; { true ==> pr.0 no hace espera de cortesia
```

PSEUDOCÓDIGO

```
{ variables compartidas y valores iniciales }
var p0sc    : boolean := falso ; { true solo si proc.0 en PE o SC }
    p1sc    : boolean := falso ; { true solo si proc.1 en PE o SC }
    turno0  : boolean := true  ; { true ==> pr.0 no hace espera de cortesia
```

```
process P0 ;
begin
   while true do begin
      p0sc := true ;
      while p1sc do begin
         if not turno0 then begin
            p0sc := false ;
            while not turno0 do
               begin end
            p0sc := true ;
         end
      end
      { sección crítica }
      turno0 := false ;
      p0sc := false ;
      { resto sección }
   end
end
```

```
process P1 ;
begin
   while true do begin
      p1sc := true ;
      while p0sc do begin
         if turno0 then begin
            p1sc := false ;
            while turno0 do
               begin end
            p1sc := true ;
         end
      end
      { sección crítica }
      turno0 := true ;
      p1sc := false ;
      { resto sección }
   end
end
```

h. ALGORITMO DE PETERSON

El algoritmo de Peterson, es otro algoritmo correcto pero más simple y más eficiente que el algoritmo de Dekker.

- También usa dos variables lógicas que expresan la presencia de cada proceso en PE o SC, más una variable de turno que permite romper el interbloqueo en caso de acceso simultáneo al PE.
- Asignación de turno se hace al inicio del PE en lugar de en PS, con lo cual, en caso de acceso simultáneo al PE, el último proceso en ejecutar la asignación (atómica) al turno da preferencia al anterior.
- A diferencia de Dekker, el PE no usa dos bucles anidados, sino que unifica ambos en uno solo.

```
{ variables compartidas y valores iniciales }
var p0sc    : boolean := falso ; { true solo si proc.0 en PE o SC }
    p1sc    : boolean := falso ; { true solo si proc.1 en PE o SC }
    turno0  : boolean := true  ; { true ==> pr.0 no hace espera de cortesia
```

```
process P0 ;
begin
   while true do begin
      p0sc := true ;
      turno0 := false ;
      while p1sc and not turno0 do
         begin end
      { sección crítica }
      p0sc := false ;
      { resto sección }
   end
end
```

```
process P1 ;
begin
   while true do begin
      p1sc := true ;
      turno0 := true ;
      while p0sc and turno0 do
         begin end
      { sección crítica }
      p1sc := false ;
      { resto sección }
   end
end
```

DEMOSTRACIÓN EM

Hipótesis de partida: Existe instante t en que ambos procesos están en SC.

- Supongamos última asignación a turno0 se hizo en un instante anterior s<t.
- En intervalo (s,t], ni p0sc (=true), ni p1sc(=true), ni turno0 cambian su valor.
- Si cualquiera (P0 ó P1) fue el que asignó turno0 en instante s ⇒ No entraría a SC durante (s,t] ya que condición del while sería true durante (s,t].
- Contradicción hipótesis ⇒ Nunca ambos estarán ejecutando su SC

ESPERA LIMITADA

Supongamos que hay un proceso (p.ej. P0) en espera ocupada en el PE, en un instante t, y veamos cuántas veces m puede entrar P1 a SC antes de que lo logre P0.

- P0 puede pasar a la SC antes que P1, en ese caso m = 0.
- P1 puede pasar a la SC antes que P0 (que continúa en bucle). En ese caso m = 1.

1/6

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

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

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

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

Quiero el cash

Consulta condiciones **aquí**

do your thing

En ambos casos, P1 no puede volver a SC mientras P0 continúa en el bucle, ya que pasaría por asignación turno0=true provocando que, tras un tiempo finito, P0 entrara a SC mientras P1 continúa en bucle. La cota que requiere la propiedad es n = 1.

PROCESO EN LA EJECUCIÓN

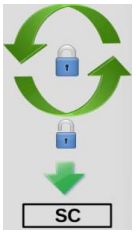
Debemos demostrar dos propiedades:

- Ausencia de interbloqueos en PE: Si suponemos que hay interbloqueo de ambos, eso significa que son indefinida y simultáneamente verdaderas ambas condiciones de sus bucles ⇒ turno0=true and (not turno0) = true) ⇒ ABSURDO.
- Independencia de procesos en RS: Si un proceso (p.ej. P0) está en PE y el otro (P1) está en RS, entonces p1sc= false y P0 puede progresar a la SC independientemente del comportamiento de P1 (que podría terminar o bloquearse estando en RS, sin impedir por ello el progreso de P0). El mismo razonamiento puede hacerse al revés.

3. SOLUCIONES HARDWARE CON ESPERA OCUPADA(CERROJOS) PARA E.M.

- Cerrojo: Solución hardware basada en espera ocupada que puede usarse en procesos concurrentes con mem. compartida para solucionar problema EM.

→Espera ocupada: Bucle que se ejecuta hasta que ningún otro proceso esté dentro de su SC.



- Valor lógico en posición de memoria compartida(cerrojo) indica si algún proceso está en SC.
- En PS se actualiza cerrojo para reflejar SC libre.

Veremos solución elemental (incorrecta) que ilustra necesidad de instrucciones hardware específicas (o soluciones más elaboradas).

a. POSIBLE SOLUCIÓN ELEMENTAL

Incorrecta. No garantiza EM. Existen secuencias de interfoliación que permiten a vario procesos acceder SC a la vez:

Situación (ya vista por nosotros)

- N procesos acceden a PE y todos leen el valor de sc_ocupada a false (ninguno lo escribe antes de que otro lo lea).
- Todos registran que SC está libre, y todos acceden.

```
var sc_ocupada : boolean := false ;

{ procesos }
process P[ i : 1 .. n ];
begin
   while true do begin
      while sc_ocupada do begin end
      sc_ocupada := true ;
      { seccion critica }
      sc_ocupada := false ;
      { resto seccion }
   end
end
```

Solución: Usar instrucciones máquina atómicas para acceso a la zona de memoria donde se aloja el cerrojo.

- Veremos una de ellas: TestAndSet.

b. POSIBLE SOLUCIÓN ELEMENTAL

TestAndSet: Instrucción máquina disponible en el repertorio de algunos procesadores.

- Argumento: Dirección de memoria de la variable lógica que actúa como cerrojo.
- Se puede invocar como una función desde LLPP de alto nivel.

Ejecuta atómicamente (no afectada por el efecto de instrucciones de otros procs.):

1) Lee valor anterior del cerrojo.
2) Pone cerrojo a true.
3) Devuelve valor anterior del cerrojo.

Solución basada en TestAndSet

```
var sc_ocupada : boolean := false ; { true solo si
                                      SC esta ocupada }
{ procesos }
process P[ i : 1 .. n ];
begin
   while true do begin
      while TestAndSet( sc_ocupada ) do begin end
      { seccion critica  }
      sc_ocupada := false ;
      { resto seccion }
   end
end
```

c. DESVENTAJAS Y USO DE LOS CERROJOS

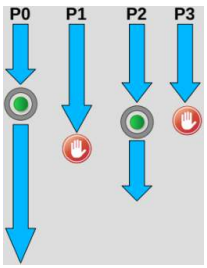
Cerrojos constituyen una solucion válida para EM: consume poca memoria, es eficiente en tiempo (excluyendo esperas ocupadas) y es válida para cualquier num. de procesos.
No obstante presenta Desventajas:

- Esperas ocupadas consumen tiempo de CPU que podría dedicarse a otros procesos para hacer trabajo útil.
- Se puede acceder directamente a los cerrojos y por tanto un programa erróneo o escrito malintencionadamente puede poner un cerrojo en un estado incorrecto, pudiendo dejar a otros procesos indefinidamente en espera ocupada.
- No se cumplen ciertas condiciones de equidad.

Desventajas hacen que su uso sea restringido:

- Por seguridad, solo se usan desde componentes software que forman parte del sistema operativo, librerías de hebras, de tiempo real o similares (suelen estar bien comprobados y libres de errores o código malicioso).
- Por eficiencia debido a la espera ocupada: solo en casos en los que la ejecución de la SC conlleva un intervalo de tiempo muy corto.

4. SEMÁFOROS PARA SINCRONIZACIÓN

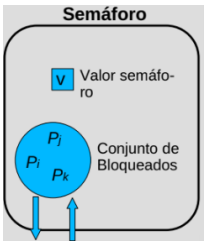


Semáforos: mecanismo que aminora problemas de soluciones de bajo nivel, y tiene un ámbito de uso más amplio.

- No se usa espera ocupada, sino bloqueo de procesos (Más eficiente).
- Resuelven fácilmente la Exclusión Mutua.
- Permiten resolver cualquier problema de sincronización (aunque esquemas de uso pueden ser complejos).
- Se implementa mediante instancias de una estructura de datos accesible únicamente mediante subprogramas específicos ⇒ aumenta la seguridad y simplicidad.

a. ESTRUCTURA DE UN SEMÁFORO

Un semáforo es un instancia de una estructura de datos que contiene los siguientes elementos en memoria compartida:



- Conjunto de procesos bloqueados (“esperando al semáforo”).
- Valor del semáforo: valor entero no negativo.

Al inicio de un programa que usa un semáforo, debe poder inicializarse:

- Su conjunto de procesos esperando estará vacío
- Se debe indicar un valor inicial del semáforo

b. OPERACIONES SOBRE UN SEMÁFORO

Además de inicialización, solo hay dos operaciones:

sem_wait(s)

```
If (s.valor ==0)
        Bloquea proceso
s.valor = s.valor - 1;
```

sem_signal(s)

```
s.valor = s.valor + 1;
If (hay procesos esperando s)
        Reanuda uno;
```

- s.valor nunca es negativo, ya que antes de decrementar se espera a que sea 1.
- Solo puede haber procesos esperando s cuando s.valor es 0.
- Estas operaciones se ejecutan en exclusión mutua sobre cada semáforo (excluyendo periodo bloqueo wait),

c. INVARIANTE DE UN SEMÁFORO

● Dado un semáforo s, que se inicializó a v0, con valor vt en un instante de tiempo, se verifica:

$$v_t = v_0 + n_s - n_w \geq 0$$

donde:

– n_s es el número de llamadas a sem_signal completadas

– n_w es el número de llamadas a sem_wait completadas.

● Los 4 valores son enteros no negativos.
● La igualdad se deriva de que sem_signal siempre incrementa el valor y sem_wait siempre lo decrementa (pero espera antes cuando es 0).
● Se mantiene cuando no se está ejecutando sem_wait o sem_signal.
● Solo cuentan llamadas a sem_wait completadas totalmente en t.

d. PATRONES DE USO SENCILLOS

→ Espera única: Un proceso, antes de ejecutar una sentencia, debe esperar a que otro proceso complete otra sentencia (típicamente cuando un proceso debe leer una variable escrita por otro proceso).

→ Exclusión Mutua: acceso a una sección crítica por parte de un número arbitrario de procesos.

→ Productor/Consumidor: un proceso escribe sucesivos valores en una variable, y cada uno de ellos debe ser leído una única vez por otro proceso.

e. ESPERA ÚNICA. PROBLEMA

```
{ variables compartidas y valores iniciales }
var x              : integer ;        { variable escrita por Productor }
    puede_leer     : semaphore := 0 ; { 1 si x ya escrita y aun no leída}
```

```
process Productor ; { escribe 'x' }
   var a : integer ;
begin
   a := ProducirValor() ;
   x := a ; { sentencia E }
end
```

```
process Consumidor  { lee 'x' }
   var b : integer ;
begin
   b := x ; { sentencia L }
   UsarValor(b) ;
end
```

- Sentencias E y L son atómicas.
- Correctas interfoliaciones en las que E antes que L.

f. ESPERA ÚNICA. SOLUCIÓN CON SEMÁFORO

```
{ variables compartidas y valores iniciales }
var x          : integer ;       { variable escrita por Productor }
    puede_leer : semaphore := 0; { 1 si x ya escrita y aun no leída }
```

```
process Productor ; { escribe 'x' }
   var a : integer ;
begin
   a := ProducirValor() ;
   x := a ; { sentencia E }
   sem_signal( puede_leer ) ;
end
```

```
process Consumidor  { lee 'x' }
   var b : integer ;
begin
   sem_wait( puede_leer ) ;
   b := x ; { sentencia L }
   UsarValor(b) ;
end
```

g. EXCLUSIÓN MUTUA. SOLUCIÓN CON SEMÁFORO

```
{ variables compartidas y valores iniciales }
var sc_libre : semaphore := 1 ; { 1 si SC está libre, 0 si SC ocupada }
                                { (núm. de procs. que pueden entrar a SC) }

process P[ i : 0..n ];
begin
  while true do begin

   { esperar hasta que sc_libre sea 1, entonces ponerla a 0 }
   sem_wait( sc_libre ) ;

   { seccion critica: ..... }

   { poner sc_libre a 1, y desbloquear un proceso en espera si hay alguno}
   sem_signal( sc_libre ) ;

   { resto seccion: ..... }
 end
end
```

h. PRODUCTOR-CONSUMIDOR. PROBLEMA

```
{ variables compartidas y valores iniciales }
var x : integer ; { contiene cada valor producido y pte. de leer }
```

```
process Productor ; { calcula x }
var a : integer ;
begin
   while true do begin
      a := ProducirValor() ;
      x := a ; { sentencia E }
   end
end
```

```
process Consumidor  { lee x }
var b : integer ;
begin
   while true do begin
      b := x ; { sentencia L }
      UsarValor(b) ;
   end
end
```

- Son correctas las interfoliaciones en las que E y L se alternan, comenzando en E.

i. PRODUCTOR-CONSUMIDOR. SOLUCIÓN CON SEMÁFOROS

```
{ variables compartidas y valores iniciales }
var x              : integer ;          { contiene cada valor producido }
    puede_leer     : semaphore := 0 ; { 1 se puede leer x, 0 no }
    puede_escribir : semaphore := 1 ; { 1 se puede escribir x, 0 no }
```

```
process Productor ; { escribe x }
var a : integer ;
begin
   while true do begin
      a := ProducirValor() ;
      sem_wait( puede_escribir  );
      x := a ; { sentencia E }
      sem_signal( puede_leer ) ;
   end
end
```

```
process Consumidor  { lee x }
var b : integer ;
begin
   while true do begin
      sem_wait( puede_leer ) ;
      b := x ; { sentencia L }
      sem_signal( puede_escribir ) ;
      UsarValor(b) ;
   end
end
```

5. MONITORES COMO MECANISMO DE ALTO NIVEL

a. MOTIVACIÓN. LIMITACIONES DE LOS SEMÁFOROS

Los semáforos resuelven fácil y eficientemente problemas de EM y de sincronización sencilla.
Sin embargo:
Difícil resolver problemas complejos de sincronización.

- Basados en variables globales: impide diseño modular y reduce escalabilidad (incorporar más procesos suele requerir replantear variables globales).
- Uso y función de las variables no explícito: difícil analizar corrección.
- Operaciones se encuentran dispersas y no protegidas: (posibilidad errores).

Programas erróneos o malintencionados pueden provocar bloqueos indefinidos o estados incorrectos.

NECESIDAD: Mecanismo que permita acceso estructurado y encapsulación, garantizando EM y permitiendo implementar condiciones de sincronización.

b. DEFINICIÓN DE MONITOR. ESTRUCTURA Y FUNCIONALIDAD

C.A.R. Hoare y Brinch Hansen, idearon el concepto de Monitor (1974): mecanismo de alto nivel que permite definir objetos abstractos compartidos, que incluyen:

- Colección de variables encapsuladas (datos) que representan un recurso compartido por varios procesos.
- Conjunto de procedimientos para manipular recurso: afectan variables encapsuladas.

Permiten al programador invocar los procedimientos de forma que:

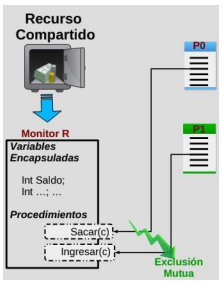
- Acceso en EM a las variables encapsuladas.
- Se implementa sincronización requerida por problema mediante esperas bloqueadas.

c. PROPIEDADES DEL MONITOR

MONITOR recurso compartido usado como objeto accedido concurrentemente.

- Acceso estructurado: Usuario (proceso) solo puede acceder al recurso mediante un conjunto de operaciones.
- Encapsulación: Usuario ignora variables que representan al recurso e implementación procedimientos.
- Acceso en EM a los procedimientos
  - Garantizada por definición.
  - Implementación garantiza: Nunca 2 procesos estarán ejecutando a la vez procedimientos.

d. VENTAJAS SOBRE LOS SEMÁFOROS



Frente a los semáforos, el uso de los monitores facilitan diseño e implementación seguros:

- Variables protegidas: sólo pueden leerse o modificarse desde el código del monitor, no desde otro punto programa.
- EM garantizada: Programador no tiene que usar mecanismos explícitos de EM para acceso a variables compartidas.
- Operaciones wait-signal se usan solo dentro monitor: Más fácil verificar corrección.

e. SINTAXIS DE UN MONITOR (PSEUDOCÓDIGO)

Una instancia del monitor se declara especificando las variables permanentes, los procedimientos del monitor y el código de inicialización.

```
monitor nombre_monitor ; { identificador con el que se referencia }
  var                    { decl. variables permanentes (privadas) }
    ..... ;              { (puede incluir valores iniciales)      }
  export                 { nombres de procedimientos públicos     }
    nom_exp_1,           { (si no aparece, todos son públicos)    }
    nom_exp_2, .... ;
  { declaraciones e implementación de procedimientos  }
  procedure nom_exp_1( ........ ); { nombre y parámetros formales }
    var .....   ;        { variables locales del procedimiento    }
  begin
     ...                 { código que implementa el procedimiento }
  end
  .......                { resto de procedimientos del monitor    }
begin                    { código inicialización de vars. perm.   }
  ....                   { (opcional)                             }
end                      { fin de la declaración del monitor      }
```

f. COMPONENTES DE UN MONITOR

Variables permanentes: **Estado interno del recurso compartido**.

- Sólo accedidas dentro del monitor (en cuerpo de procedimientos y código inicialización).
- Sin modificaciones entre dos llamadas consecutivas a procedimientos.

Procedimientos: **Modifican estado** (en EM).

- Pueden tener variables locales, que toman un nuevo valor en cada activación del procedimiento.
- Algunos forman la interfaz externa del monitor y podrán ser llamados por procesos compartiendo recursos.

Código de inicialización: **fija estado inicial.**

- Se ejecuta una única vez, antes de cualquier llamada a procedimiento.

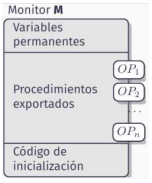
```
monitor nombre_monitor ;
  var
    ..... ;
  export
    nom_exp_1,
    nom_exp_2, .... ;
  procedure nom_exp_1( ..
    var .....   ;
  begin
     ...
  end
  .......
begin
  ....
end
```

g. DIAGRAMA DE COMPONENTES DE UN MONITOR

→ Interfaz con el exterior: el uso que se hace del monitor se hace exclusivamente usando los procedimientos exportados.

→ Encapsulación: Variables permanentes y procedimientos no exportados no son accesibles desde fuera.

→ Ventaja: Implementación operaciones se puede cambiar sin modificar la semántica.



EJEMPLO DE MONITOR

Varios procesos pueden incrementar (en una unidad) una variable compartida y examinar su valor en cualquier momento, invocando operaciones.

```
{ declaracion del monitor }
monitor VarCompartida ;

 var x : integer; { permanente }
 export incremento, valor;

 procedure incremento( );
 begin
   x := x+1 ; {incrementa valor}
 end;
 function valor() : integer ;
 begin
   result := x; { escribe result.}
 end;

begin { código de inicialización}
  x := 0 ; { inicializa valor }
end { fin del monitor}
```

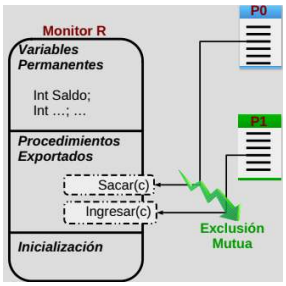
Código Proceso Ususario

```
{ ejemplo de uso del monitor  }
{ (debe aparecer en el ámbito }
{ de la declaración)          }

{ incrementar valor: }
VarCompartida.incremento();

{ copiar en k el valor: }
k := VarCompartida.valor() ;
```

h. FUNCIONAMIENTO DE LOS MONITORES



Comunicación Monitor-Mundo exterior: Cuando un proceso necesita operar sobre recurso compartido controlado por un monitor deberá invocar uno de los procedimientos exportados con los parámetros apropiados.

– Mientras el proceso P está ejecutando un procedimiento del monitor decimos que P está dentro del monitor.

Exclusión mutua: Si proceso P está dentro monitor, cualquier otro proceso que invoque un procedimiento deberá esperar a que P salga:

– Asegura: las variables permanentes nunca son accedidas concurrentemente.

– Debe garantizarse en la implementación del monitor.

Son objetos pasivos: Tras ejecutar código de inicialización, un monitor es un objeto pasivo y sus procedimientos sólo se ejecutan cuando son invocados.

Instanciación de clases de monitores: En algunos casos es conveniente crear múltiples instancias independientes de un tipo de monitor:

– Cada instancia tiene sus variables permanentes propias.

– La E.M. ocurre en cada instancia por separado.

```
class monitor nombre_clase_monitor( parametros_formales ) ;
    .... .......... { cuerpo del monitor semejante a los anteriores }
end
var nombre_instancia_1 : nombre_clase_monitor( parametros_actuales_1 ) ,
    nombre_instancia_2 : nombre_clase_monitor( parametros_actuales_2 ) ;
```

i. INSTALACIÓN DE UNA CLASE DE MONITOR. EJEMPLO

```
{ declaracion de la clase monitor }

class monitor VarComp(pini,pinc : integer)

   var x, inc : integer ;
   export incremento, valor;

   procedure incremento( );
   begin
      x := x+inc ;
   end;
   function valor(): integer ;
   begin
      result := x ;
   end;

begin
   x:= pini ; inc := pinc ;
end
```

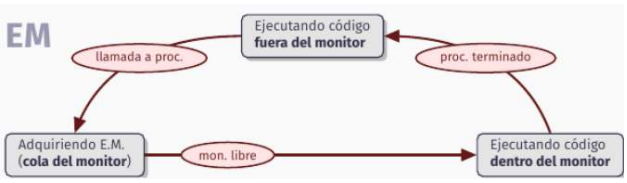
```
{ ejemplo de uso }

var mv1   : VarComp(0,1);
    mv2   : VarComp(10,4);
    i1,i2 : integer ;
begin
 mv1.incremento();
 i1:= mv1.valor();{ i1 == 1 }
 mv2.incremento();
 i2:= mv2.valor();{ i2 == 14 }
end
```

j. COLA DEL MONITOR PARA EM

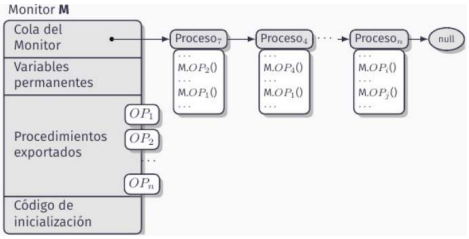
Control de la EM se basa en existencia de una Cola del monitor:

- Si proceso P dentro monitor y otro proc. Q intenta ejecutar procedimiento, Q queda bloqueado, insertándose en la cola del monitor.
- Proceso abandona el monitor (finaliza ejec. procedimiento) ↦ Se desbloquea un proceso de la cola, y puede entrar.
- Cola vacía y ningún proceso dentro ↦ Monitor libre. Primer proceso que invoque procedimiento, entrará.
- Planificación de la cola debe seguir una política FIFO ↦ Garantiza vivacidad.



k. ESTADO DEL MONITOR

Estado del monitor: incluye la cola de procesos esperando ejecución procedim.



l. SINCRONIZACIÓN CON MONITORES

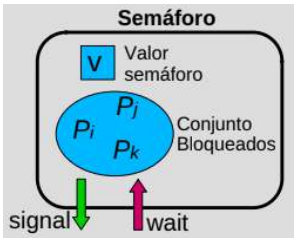
Implementar sincronización requiere permitir esperas bloqueadas en los procesos, hasta que una condición sea cierta:

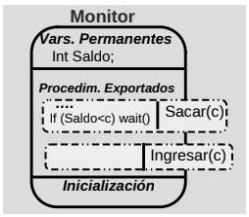
Semáforos

- Bloqueo (sem_wait) y activación (sem_signal) + Valor del semáforo (indica si condición se cumple o no).

Monitores

- Sentencias de bloqueo y activación sin valor.
- Los valores de las variables permanentes determinan si la condición se cumple.

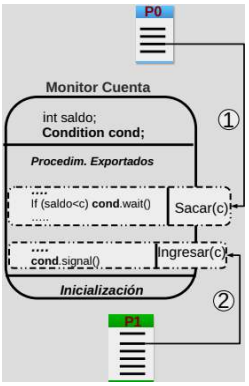




m. BLOQUEO Y ACTIVACIÓN CON VARIABLES CONDICIÓN

Para cada condición de espera distinta, se debe de declarar una variable permanente de tipo ***condition***. A esas variables las llamamos señales o variables condición:

- Cada ***variable condition*** tiene asociada una cola de procesos esperando condición cierta.
- Sobre una ***var. condition cond***, un proceso puede invocar dos operaciones:
  – cond.wait(): Espero a que alguna condición ocurra.
  – cond.signal(): Señalo que condición ocurre.



Dada una variable condición cond, se definen:

- cond.wait(): Bloquea al proceso invocador y lo introduce en la cola de la variable cond.
- cond.signal(): si hay procesos bloqueados en cond, libera uno de ellos.
  – Política FIFO: Reactivará al que lleve más tiempo esperando.
  Evita inanición: cada proceso en cola obtendrá eventualmente turno.
- cond.queue(): true si cola de condición no vacía, y false en caso contrario.

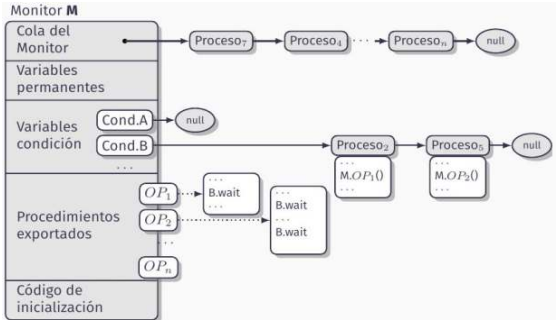
n. ESPERAS BLOQUEADAS Y EM EN EL MONITOR

Los procesos pueden estar dentro del monitor, pero bloqueados:

- Cuando el proceso llama a wait y queda bloqueado, se libera EM del monitor.
  – En caso contrario ⇨ interbloqueo (proceso quedaría bloqueado y el resto al intentar entrar).
- Cuando proceso es reactivado, adquiere EM antes de ejecutar la sentencia siguiente a wait.
- Más de un proceso podrá estar dentro del monitor, aunque solo uno de ellos estará ejecutándose, el resto estarán bloqueados en colas de condición.

o. ESTADO DE UN MONITOR CON VARIAS COLAS

Supongamos: los procesos 2 y 5 ejecutan las operaciones OP1 y OP2, ambas producen esperas de la condición B.



p. ESPERA ÚNICA. INTERACCIÓN ENTRE LOS PROCESOS

Monitor EU se puede usar para sincronizar la lectura y escritura de una variable compartida, de esta forma:

```
{ variables compartidas }
var x : integer ;  { contiene cada valor producido }
```

```
process Productor ; { escribe x }
var a : integer ;
begin
   a := ProducirValor() ;
   x := a ;          { sentencia E }
   EU.notificar() ; { sentencia N }
end
```

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

q. ESPERA ÚNICA. MONITOR EU

```
monitor EU ;                { Monitor de Espera Única (EU)}

var terminado : boolean;    { true si se ha terminado E, false sino }
    cola    : condition;    { cola consumidor esperando terminado==true }
export esperar, notificar; { nombra procedimientos públicos }

procedure esperar();    { para llamar antes de L }
begin
  if not terminado then { si no se ha terminado E }
    cola.wait();        { esperar hasta que termine }
end
procedure notificar();  { para llamar después de E }
begin
  terminado := true;    { registrar que ha terminado E }
  cola.signal();        { reactivar el otro proceso, si espera }
end
begin                   { inicializacion: }
  terminado := false;   {   al inicio no ha terminado E }
end
```

r. EM. INTERACCIÓN ENTRE LOS PROCESOS

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

s. EM. MONITOR EM

```
monitor EM ;

var ocupada : boolean ;  { true hay un proceso en SC, false sino }
    cola    : condition; { cola de procesos esperando ocupada==false}
export entrar, salir ;   { nombra procedimientos públicos }

procedure entrar();   { protocolo de entrada (sentencia E)}
begin
  if ocupada then     { si hay un proceso en la SC }
    cola.wait();      {    esperar hasta que termine }
  ocupada := true;    { indicar que la SC está ocupada }
end
procedure salir();    { protocolo de salida (sentencia S)}
begin
  ocupada := false;   { marcar la SC como libre }
  cola.signal();      { si al menos un proceso espera, reactivar uno }
end

begin                 { inicializacion: }
  ocupada := false;   {   al inicio no hay procesos en SC }
end
```

t. PRODUCTOR-CONSUMIDOR. SINCRONIZACIÓN CON MONITOR PC

```
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
```

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

u. PRODUCTOR-CONSUMIDOR. MONITOR PC

```
Monitor PC ;
var valor_com : integer ;   { valor compartido }
    pendiente : boolean ;   { true solo si hay valor escrito y no leído }
    cola_prod : condition ; { espera productor hasta que pendiente == false }
    cola_cons : condition ; { espera consumidor hasta que pendiente == true }
```

```
procedure escribir( v : integer );
begin
  if pendiente then
    cola_prod.wait();
  valor_com := v ;
  pendiente := true ;
  cola_cons.signal();
end
```

```
function leer() : integer ;
begin
  if not pendiente then
    cola_cons.wait();
  result    := valor_com ;
  pendiente := false ;
  cola_prod.signal();
end
```

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

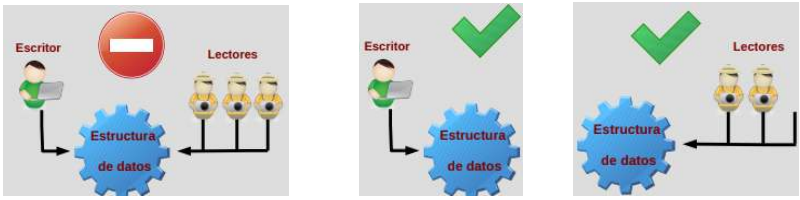
v. EL PROBLEMA DE LOS LECTORES/ESCRITORES (LE)

Dos tipos de procesos acceden concurrentemente a datos compartidos:

● Escritores: Escriben en la Estructura de Datos (ED). El código de escritura no puede ejecutarse concurrentemente con otra escritura ni lectura, ya que ponen la ED en un estado no usable por otros procesos.

● Lectores: Leen la ED, pero no modifican su estado. La lectura puede ejecutarse concurrentemente por varios lectores de forma arbitraria, pero no con la escritura.

Solución: compleja con semáforos, pero sencilla con monitores.



w. EL PROBLEMA LE. SOLUCIÓN CON MONITOR

- Esta implementación da **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.

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

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

x. EL PROBLEMA LE. DESCRIPCIÓN MONITOR Lec_Esc

```
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
```

```
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;
```

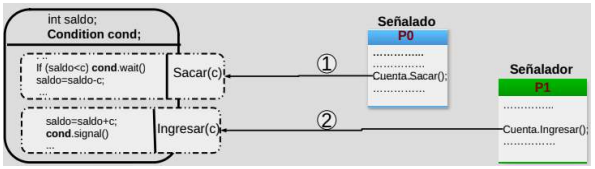
```
procedure fin_escritura()
begin
 { registrar que ya no hay escritor}
 escribiendo := false;
 { si hay lectores, despertar uno}
 { si no hay, despertar un escritor}
 if lectura.queue() then
    lectura.signal();
 else
    escritura.signal() ;
end;
```

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

y. SEMÁNTICA DE LAS SEÑALES DE LOS MONITORES

Supongamos que proceso invoca signal sobre cola no vacía (proc. señalador) y reactiva a un proceso que esperaba tras un wait (proc. señalado):

- Suponemos que hay código restante tras ese wait y ese signal.
- Al finalizar el signal, solo uno de los procs. puede continuar ejecutando su código restante. En otro caso, se violaría la EM del monitor.
- Semántica de señales: política concreta para resolver conflicto tras signal.



z. POSIBLES SEMÁNTICAS DE LAS SEÑALES. ESQUEMA GENERAL

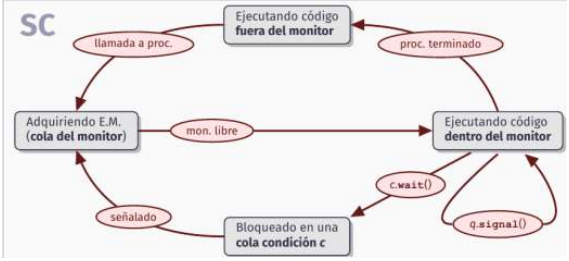
- Proceso señalador continúa la ejecución tras signal. Señalado se bloquea hasta adquirir E.M. de nuevo (SC: Señalar y Continuar).
- El proceso señalado se reactiva inmediatamente.
  - Señalador abandona monitor tras signal (SS: Señalar y Salir).
  - Señalador queda bloqueado en:
- Cola del monitor (EM) (SE: Señalar y Esperar).
- Cola con máxima prioridad (SU: Señalar y espera Urgente).

aa. SEMÁNTICA DE LAS SEÑALES. SEÑALAR Y CONTINUAR (SC)

Señalador continúa ejecución dentro del monitor después signal.

- Señalado abandona cola condición y espera en cola monitor a readquirir E.M.
- Tanto el señalador como otros procesos pueden hacer falsa condición de reactivación.
- En proceso señalado no se puede garantizar que la condición asociada a cond sea cierta al terminar cond.wait(). Esta semántica obliga a realizar wait en un bucle:

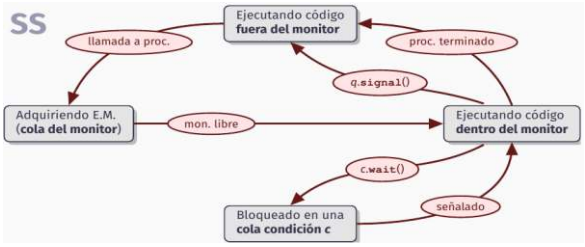
```
while not condicion_lógica_desbloqueo do
   cond.wait() ;
```



bb. SEMÁNTICA DE LAS SEÑALES. SEÑALAR Y SALIR (SS)

Señalador sale del monitor después cond.signal(). Si hay código tras signal, no se ejecuta. El proceso señalado reanuda inmediatamente ejecución.

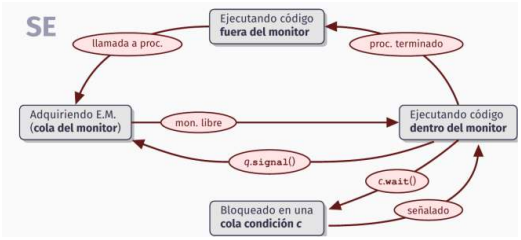
- Operación signal conlleva: a) Liberación Señalado y b) Terminación procedimiento que ejecutaba Señalador.
- Se cumple condición de activación señalador: Se asegura el estado que permite al Señalado continuar ejecución del procedimiento donde se bloqueó.
- Obliga a colocar signal como última instrucción procedimientos monitor.



cc. SEMÁNTICA DE LAS SEÑALES. SEÑALAR Y ESPERAR (SE)

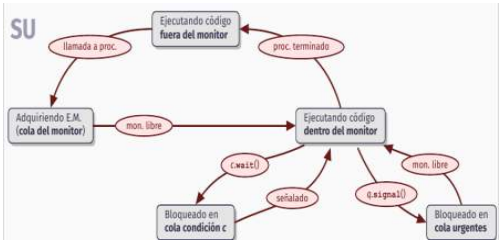
Señalador se bloquea en la cola del monitor justo después signal y Señalado reanuda inmediatamente.

- Se garantiza condición de activación para el señalado.
- Señalador debe competir por EM con resto procesos.
  - Semántica injusta respecto al proceso señalador (ya había obtenido el acceso).



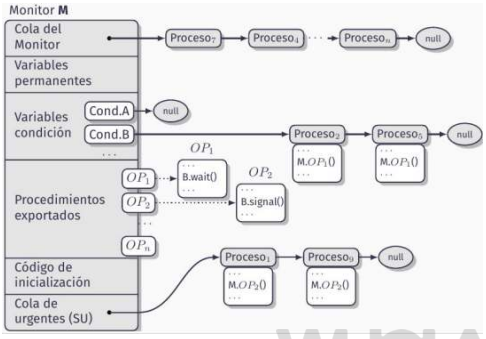
dd. SEMÁNTICA DE LAS SEÑALES. SEÑALAR Y ESPERA URGENTE (SU)

- Similar SE, pero se intenta corregir el problema de falta de equidad:
- Proceso señalador se bloquea tras signal y señalado reanuda inmediatamente, garantizándose condición de reactivación.
- Señalador entra en una nueva cola de procesos que esperan para acceder al monitor, llamada cola de procesos urgentes.
  – Procesos en esta tienen preferencia frente a procesos en cola monitor.



ee. SU. COLAS DE URGENTES

Procesos 1 y el 9 han ejecutado la **OP2**, que hace signal de cond.B.



Potencia expresiva

- Equivalentes: Todas las semánticas son capaces de resolver los mismos problemas.

Facilidad de uso

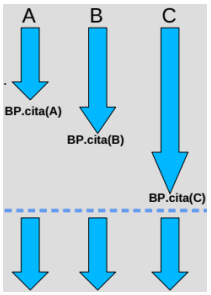
- Semántica SS condiciona el estilo de programación y puede llevar a aumentar de forma artificial el número de procedimientos.

Eficiencia

- SE y SU podrían resultar ineficientes cuando no hay código tras signal, ya que señalador debe bloquearse y reactivarse, para justo después abandonar el monitor.
- SC también es algo ineficiente al obligar usualmente a poner cada instrucción wait dentro de un bucle while.

ff. EJEMPLO COMPARATIVO SEMÁNTICAS. BARRERA PARCIAL

Ejemplo para mostrar diferencias entre semánticas:
Monitor barrera parcial (BP).



- Monitor con un único procedimiento público: cita.
- Hay 3 procesos A, B y C ejecutando un bucle, en cada iteración invocan cita pasando nombre como argumento.

Requisitos de Sincronización

1) Al inicio de cita, primero imprime “Llega P”, y al final, antes de terminar, imprime “Sale P”, (P∈{A,B,C}).
2) Cada 2 procesos consecutivos que invocan cita forman una pareja: el primero (X) debe de esperar al segundo (Y), de forma que tras llegar Y, ambos (X,Y) pueden comenzar a la vez.
3) Mensajes salida se deben imprimir en el mismo orden que los de llegada: : Llega X, Llega Y, Sale X, Sale Y.
4) Mensajes de dos citas distintas no pueden entremezclarse : 4 de una cita aparecen consecutivos.

gg. IMPLEMENTACIÓN DEL MONITOR BARRERA PARCIAL

- Única variable condición: al llegar a cita, cada proceso sabe si es el primero o el segundo simplemente viendo si la cola está vacía (primero) o no (segundo).
- Requerimientos obligan al segundo proceso a ejecutar el print después del signal, ya que por (1) ese segundo proceso no puede hacer nada después de ese print.

```
monitor BP ;

    var q : condition ; { cola donde el primero espera al segundo }

    procedure cita( p : char ) ;
    begin
        print( "Llega ", p );
        if q.queue() then q.signal(); { segundo: despierta al primero }
                     else q.wait();   { primero: espera al segundo }
        print( "Sale ", p );
    end
end { fin monitor BP }
```

hh. BARRERA PARCIAL. ANÁLISIS COMPORTAMIENTO

- Supongamos que la cola q está vacía, y entonces A, B y C llaman a cita. El proceso A entra al monitor (inicia cita), mientras B y después C (en ese orden) ingresan en la cola del monitor.
- Mensajes que deben aparecer: Llega A, Llega B, Sale A, Sale B, Llega C,
- Cuando A entra al monitor, imprime Llega A, ve la cola vacía y queda bloqueado en el wait. Entonces entra B a la cita, ve la cola no vacía y hace el signal (cuando C todavía está en la cola del monitor).
- Lo que pasa justo después depende de la semántica concreta.

```
monitor BP ;

    var q : condition ; { cola donde

    procedure cita( p : char ) ;
    begin
        print( "Llega ", p );
        if q.queue() then q.signal();
                     else q.wait();
        print( "Sale ", p );
    end
end { fin monitor BP }
```

Semántica SC (Señalar y Continuar)

- B continúa, imprime Sale B, después A se desbloquea e imprime Sale A: No se respeta el orden de los mensajes requerido. Incorrecto
- Mensajes: Llega A, Llega B, Sale B, Llega C, Sale A, ...

Semántica SS (Señalar y Salir)

- B acaba cita (sale del monitor) y no imprime nunca el mensaje final de cita: Incorrecto.
- Mensajes: Llega A, Llega B, Sale A, Llega C, ...

Semántica SE (Señalar y Esperar)

- B va la cola del monitor, después A se desbloquea, imprime “Sale A” y abandona monitor. Entonces C entra a cita, imprime “Llega C”, ve la cola q vacía y se bloquea, finalmente B vuelve al monitor e imprime “Sale B”. Se entremezclan mensajes de citas distintas: Incorrecto
- Mensajes: Llega A, Llega B, Sale A, Llega C, Sale B, ...

Semántica SU. (Señalar y Espera Urgente)

- B va a la cola de urgentes, después A se desbloquea, imprime Sale A, después B vuelve de urgentes, imprime Sale B y finalmente C inicia la siguiente cita, imprimiendo Llega C: Todo Correcto.
- Mensajes: Entra A, Entra B, Sale A, Sale B, Llega C, ...

ii. COLAS DE CONDICIÓN CON PRIORIDAD

Por defecto → colas de espera FIFO.
A veces resulta útil dar prioridad a unos procesos sobre otros, aportando un parámetro entero al invocar wait. Sintaxis: ***cond.wait(p)***

- p: entero no negativo que refleja la prioridad (cuanto menor mayor prioridad).
- cond.signal(): reanudará un proceso que especificó el valor mínimo de p de entre los que esperan (si hay más de uno, se usa política FIFO).
- Se deben evitar riesgos como la inanición.
- No tiene efecto sobre la corrección del programa: el funcionamiento es similar con/sin prioridad.
- Sólo mejoran las características dependientes del tiempo.

EJEMPLO COLAS CON PRIORIDAD. ASIGNADOR RECURSO

Los procesos, cuando adquieren un recurso, especifican un valor entero de duración de uso del recurso. Se prioriza a los que requieran menos tiempo.

```
monitor RecursoPrio;
   var  ocupado : boolean ;
        cola    : condition ;
   export adquirir, liberar ;

procedure adquirir(tiempo: integer);
begin
   if ocupado then
      cola.wait( tiempo );
   ocupado := true ;
end

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

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

EJEMPLO COLAS CON PRIORIDAD. RELOJ CON ALARMA

```
Monitor Despertador;
   var    ahora      : integer ;   { instante actual }
          despertar  : condition ; { procesos esperando a su hora }
   export despiertame, tick ;

procedure despiertame( n: integer );
   var alarma : integer;
begin
   alarma := ahora + n;
   while ahora < alarma do
      despertar.wait( alarma );
   despertar.signal();
   { por si otro proceso
     coincide en la alarma }
end

{ un proceso ejecuta esto }
{ regularmente, tras cada }
{ unidad de tiempo }

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

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

jj. IMPLEMENTACIÓN DE MONITORES CON SEMÁFOROS. ENFOQUE

Es posible implementar cualquier monitor usando semáforos para modelar las colas del monitor.

- Cola del monitor: Se implementa con un semáforo binario mutex (0 si algún proceso está ejecutando código y 1 en otro caso).
- Colas de condición: Para cada variable condición será necesario definir un semáforo (siempre a 0) y una variable entera que registra número de procesos esperando.
- Cola de procesos urgentes (en semántica SU): debe haber un contador entero (num. procesos esperando) y un semáforo (siempre a 0).

Limitaciones
→ No permite llamadas recursivas a los procedimientos.
→ No asegura orden FIFO en colas.

"Los semáforos y monitores son equivalentes en potencia expresiva pero los monitores facilitan el desarrollo".

kk. IMPLEMENTACIÓN CON SEMÁFOROS. EM

Implementación de la EM en el acceso a los procs

```
procedure P1(...) { impl. de un proc. }
begin             { del monitor }
   sem_wait(mutex);
   { cuerpo del procedimiento }
   sem_signal(mutex);
end
```

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

Con semántica SU: necesitamos un semáforo urgentes y un contador entero n_urgentes (núm. bloqueados en urgentes):

```
procedure P1(...) { impl. de un proc. }
begin             { del monitor }
   sem_wait(mutex);
   { cuerpo del procedimiento }
   if n_urgentes > 0 then sem_signal(urgentes);
                     else sem_signal(mutex);
end
```

```
{ inicialización }
urgentes   := 0 ;
n_urgentes := 0 ;
```

ll. IMPLEMENTACIÓN CON SEMÁFOROS. VARIABLES CONDICIÓN

Para cada variable condición cond definimos un semáforo asociado cond_sem, (siempre a 0) y un contador entero n_cond (inicializado a 0) para contabilizar procesos bloqueados en dicho semáforo:

Cond.wait()

```
n_cond := n_cond + 1 ;
if n_urgentes != 0 then
   sem_signal(urgentes) ;
else
   sem_signal(mutex);
sem_wait(cond_sem);
n_cond := n_cond - 1 ;
```

Cond.signal()

```
if n_cond != 0 then begin
   n_urgentes := n_urgentes + 1 ;
   sem_signal(cond_sem);
   sem_wait(urgentes);
   n_urgentes := n_urgentes - 1 ;
end
```