Tema2.1Sincronizacion-en-memoria...

beatrizmartin05

Sistemas Concurrentes y Distribuidos

2º Grado en Ingeniería Informática

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

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. Este tipo de soluciones se pueden dividir en dos categorías:

- **Soluciones de bajo nivel con espera ocupada**

Cuando un proceso debe esperar a que ocurra un evento o sea cierta determinada condición, entra en un bucle indefinido donde continuamente comprueba si la situación ya se da o no (espera ocupada).

- **Soluciones de alto nivel**

Se ofrecen interfaces de acceso a estructuras de datos y además se usa bloqueo de procesos en lugar de espera ocupada:

1. Semáforos
2. Secciones críticas condicionales
3. Monitores

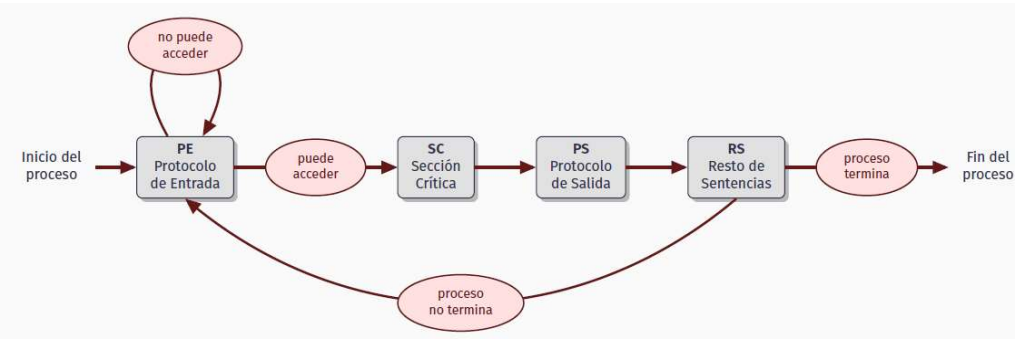
2. SOLUCIONES SOFTWARE CON ESPERA OCUPADA PARA E.M.

En esta sección veremos diversas soluciones para lograr exclusión mutua en una sección crítica usando variables compartidas entre los procesos o hebras involucrados.

2.1. Estructura de los procesos con secciones críticas

Entrada y salida en secciones críticas

1. **Protocolo de entrada** (PE): una serie de instrucciones que incluyen posiblemente espera, en los casos en los que no se pueda conceder acceso a la sección crítica.
2. **Sección crítica** (SC): instrucciones que solo pueden ser ejecutadas por un proceso como mucho.
3. **Protocolo de salida** (PS): instrucciones que permiten que otros procesos puedan conocer que este proceso ha terminado la sección crítica.



2.2. Propiedades para exclusión mutua

Para que un algoritmo para EM sea correcto, se deben cumplir cada una de 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

Propiedad de exclusión mutua

Es la propiedad fundamental para el problema de la sección crítica establece que: en cada instante de tiempo, y para cada sección crítica existente, habrá como mucho un proceso ejecutando alguna sentencia de dicha región crítica.

Espera limitada

La propiedad de espera limitada establece que: un algoritmo de exclusión mutua debe estar diseñado de forma que n nunca será superior a un valor máximo determinado.

2.3. Refinamiento sucesivo de Dijkstra

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

Se comienza desde una versión muy simple, incorrecta (no cumple alguna de las propiedades), y se hacen sucesivas mejoras para intentar cumplir las tres propiedades. Esto ilustra muy bien la importancia de dichas propiedades.

La versión final correcta se denomina **Algoritmo de Dekker**.

EJ: Se asume que hay dos procesos, denominados P0 y P1, cada uno de ellos ejecuta un bucle infinito conteniendo: protocolo de entrada, sección crítica, protocolo de salida y otras sentencias del proceso.

- Versión 1

En esta versión se usa una variable lógica compartida (**p01sc**) que valdrá **true** solo si el proceso 0 o el proceso 1 están en SC, y valdrá **false** si ninguno lo está:

```
{ 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
```

Esa versión no es correcta. El motivo es que no cumple la propiedad de **exclusión mutua**, pues ambos procesos pueden estar en la sección crítica. Esto puede ocurrir si ambos leen p01sc y ambos la ven a false.

- Versión 2

Para solucionar el problema se usará una única variable lógica (**turno0**), cuyo valor servirá para indicar cuál de los dos procesos tendrá prioridad para entrar SC la próxima vez que lleguen al PE. La variable valdrá **true** si la prioridad es para el proceso 0, y **false** si es para el proceso 1:

```
{ 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
```

Esta segunda versión no es tampoco correcta.
Se cumple la propiedad de exclusión mutua pero no se cumple la propiedad de **progreso en la ejecución**.
El problema está en que este esquema obliga a los procesos a acceder de forma alterna a la sección crítica.

- Versión 3

Para solucionar el problema de la alternancia, ahora usamos dos variables lógicas (**p0sc**, **p1sc**) en lugar de solo una. Cada variable vale **true** si el correspondiente proceso está en la sección crítica:

```
{ 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
```

De nuevo, esta versión no es correcta.
Sí se cumple la propiedad de progreso, pero no se cumple la exclusión mutua.

- Versión 4

Para solucionar el problema anterior se puede cambiar el orden de las dos sentencias del PE. Ahora las variables lógicas **p0sc** y **p1sc** están a **true** cuando el correspondiente proceso está en SC, pero también cuando está intentando entrar (en el 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
```

De nuevo, esta versión no es correcta.
Ahora es fácil demostrar que sí se cumple la E.M.
También se permite el entrelazamiento con regiones no críticas.
Sin embargo, no se cumple el progreso en la ejecución, ya que puede ocurrir interbloqueo.

- 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
```

Ahora se cumple exclusión mutua pero no es posible afirmar que es imposible que se produzca interbloqueo en el PE
La posibilidad de interbloqueo es pequeña, y depende de cómo se seleccionen las duraciones de los tiempos de la espera de cortesía.

2.4. Algoritmo de Dekker

El algoritmo de Dekker debe su nombre a su inventor, es un algoritmo correcto (es decir, cumple las propiedades mínimas establecidas), y se puede interpretar como el resultado final del refinamiento sucesivo de Dijkstra.

- Al igual que en la versión 5, cada proceso incorpora una espera de cortesía durante la cual le cede al otro la posibilidad de entrar en SC, cuando ambos coinciden en el PE.
- Para evitar interbloqueos, la espera de cortesía solo la realiza uno de los dos procesos, de forma alterna, mediante una variable de turno (parecido a la versión 2).
- La variable de turno permite también saber cuándo acabará la espera de cortesía, que se implementa mediante un bucle (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 }
```

```
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
```

2.5. Algoritmo de Peterson

Este algoritmo (que también debe su nombre a su inventor), es otro algoritmo correcto para EM, que además es más simple que el algoritmo de Dekker.

- Al igual que el algoritmo de Dekker, usa dos variables lógicas que expresan la presencia de cada proceso en el PE o la SC, más una variable de turno que permite romper el interbloqueo en caso de acceso simultáneo al PE.
- A diferencia del algoritmo de Dekker, el PE no usa dos bucles anidados, sino que unifica ambos en uno solo.

El esquema del algoritmo queda como sigue:

```
{ 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
```

Progreso en la ejecución

Para asegurar el progreso es necesario asegurar

- **Ausencia de interbloqueos** en el **PE**: esto es fácil de demostrar pues si suponemos que hay interbloqueo de los dos procesos, eso significa que son indefinida y simultáneamente verdaderas las dos condiciones de los bucles de espera, y eso implica que es verdad turno0 and not turno0, lo cual es absurdo.
- **Independencia de procesos** en **RS**: Si un proceso (p.ej. el 0) está en PE y el otro (el 1) está en RS, entonces p1sc vale false y el proceso 0 puede progresar a la SC independientemente del comportamiento del proceso 1 (qur podría terminar o bloquearse estando en RS, sin impedir por ello el progreso del proc.0). El mismo razonamiento puede hacerse al revés. Luego es evidente que el algoritmo cumple la condición de progreso.

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

Los cerrojos constituyen una solución hardware basada en espera ocupada que puede usarse en procesos concurrentes con memoria compartida para solucionar el problema de la exclusión mutua.

Vemos un esquema para procesos que ejecutan SC y RS repetidamente:

```
{ variables compartidas y valores iniciales }
var sc_ocupada : boolean := false ; { cerrojo: verdadero sólo si SC ocupada }

{ procesos }
process P[ i : 1 .. n ];
begin
   while true do begin
      while sc_ocupada do begin end
      sc_ocupada := true ;
      { sección crítica }
      sc_ocupada := false ;
      { resto de sentencias }
   end
end
```

La solución anterior no es correcta, ya que no garantiza exclusión mutua al existir secuencias de mezclado de instrucciones que permiten a más de un proceso ejecutar la SC a la vez.
Una solución es usar instrucciones máquina atómicas como por ej. TestAndSet.

La instrucción TestAndSet

- Admite como argumento la dirección de memoria de la variable lógica que actúa como cerrojo.
- Se invoca como una función desde LLPP de alto nivel, y ejecuta estas acciones:
    1. Lee el valor anterior del cerrojo
    2. Pone el cerrojo a true
    3. Devuelve el valor anterior del cerrojo
- Durante su ejecución, ninguna otra instrucción ejecutada por otro proceso puede leer ni escribir la variable lógica: por tanto, se ejecuta de **forma atómica.**

La forma adecuada de usar `TestAndSet` es la que se indica en este esquema:

```
{ variables compartidas y valores iniciales }
var sc_ocupada : boolean := false ; { true sólo si la SC está ocupada }

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

Uso de cerrojos

Las desventajas indicadas hacen que el uso de cerrojos sea restringido:

- Por seguridad, normalmente sólo se usan desde componentes software que forman parte del sistema operativo
- Para evitar la pérdida de eficiencia que supone la espera ocupada, se usan sólo en casos en los que la ejecución de la SC conlleva un intervalo de tiempo muy corto.

4.ESTRUCTURAS Y OPERACIONES DE LOS SEMÁFOROS

4.1. Estructura y operaciones de los semáforos

Los semáforos constituyen un mecanismo que soluciona o aminora los problemas de las soluciones de bajo nivel y tienen un ámbito de uso más amplio:

- No se usa espera ocupada, sino bloqueo de procesos.
- Resuelven fácilmente el problema de exclusión mutua con esquemas de uso sencillos.
- Se pueden usar para resolver problemas de sincronización.
- El mecanismo se implementa mediante instancias de una estructura de datos a las que se accede únicamente mediante subprogramas específicos. Esto aumenta la seguridad y simplicidad de los programas.

Bloqueo y desbloqueo de procesos

Los semáforos exigen que los procesos bloqueados no ocupen CPU. Esto implica:

- Un proceso en ejecución debe poder solicitar quedarse bloqueado.
- Un proceso bloqueado no puede ejecutar instrucciones en la CPU.
- Un proceso en ejecución debe poder solicitar que se desbloquee (se reanude) algún otro proceso bloqueado.
- Deben poder existir simultáneamente conjuntos de procesos bloqueados.

Estructura de un semáforo

Contiene los siguientes elementos:

1. Un conjunto de procesos bloqueados.
2. Un valor natural (un valor entero no negativo), el valor del semáforo.

Estas estructuras de datos residen en memoria compartida. Un programa que usa semáforos debe poder **inicializarlos** al comienzo de su ejecución.

Operaciones sobre los semáforos

Además de la inicialización, sólo hay dos operaciones básicas que se pueden realizar sobre los semáforos:

- **sem_wait(s)**:
    1. Si el valor de s es 0, bloquear el proceso. Éste se reanudará después en un instante en que el valor ya es 1.
    2. Decrementar el valor del semáforo en una unidad.

- **sem_signal(s)**:
    1. Incrementar el valor de s en una unidad.
    2. Si hay procesos esperando en s, reanudar o despertar a uno de ellos (ese proceso pondrá el valor del semáforo a 0 al salir).

Implementación:

```
procedure sem_wait( s :semaphore);
begin
   if s.valor == 0 then
      bloquearme( s.procesos ) ;
   s.valor := s.valor - 1 ;
end
```

```
procedure sem_signal( s :semaphore);
begin
   s.valor := s.valor + 1 ;
   if not vacio( s.procesos ) then
      desbloquear_uno( s.procesos ) ;
end
```

4.2. Uso de semáforos: patrones sencillos

Cada patrón es un esquema que permite solucionar, usando semáforos, la sincronización de un problema típico sencillo.
En concreto, veremos estos tres problemas:

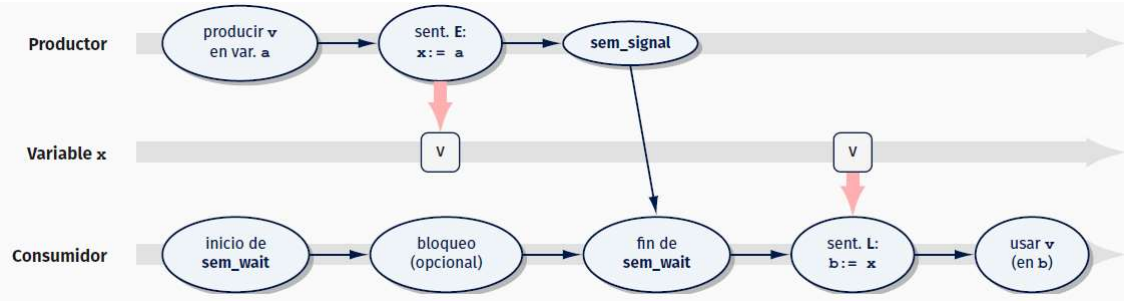
1. **Espera única**: un proceso, antes de ejecutar una sentencia, debe esperar a que otro proceso complete otra sentencia.
2. **Exclusión mutua**: acceso en exclusión mutua a una sección crítica por parte de un número arbitrario de procesos.
3. **Problema del Productor/Consumidor**: similar a la espera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

Espera única: solución con un 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
```



Uso de semáforos para **exclusión mutua**

```
{ 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 ) ;

   { sección crítica: ..... }

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

   { resto de sentencias: ..... }
 end
end
```

Solución del **Productor/Consumidor** 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
```

Limitaciones de los semáforos

Los semáforos resuelven de una forma eficiente y sencilla el problema de la exclusión mutua y problemas sencillos de sincronización, sin embargo:

- Los problemas más complejos de sincronización se resuelven de forma algo más compleja (difícil verificar su validez,fácil que sean incorrectos).
- Al igual que los cerrojos, programas erróneos o malintencionados pueden provocar que haya procesos bloqueados indefinidamente o en estados incorrectos.