Universidad de Huelva

Universidad de Huelva

Grado en Ingeniería Informática

Tema 3. Semáforos

Resumen

Autor: Alberto Fernández Merchán
Asignatura: Programación Concurrente y Distribuida

Septiembre 2021

1. Introducción

Dos o más procesos pueden cooperar por medio de señales. de forma que un proceso puede detenerse en una posición determinada hasta que reciba una señal.

Para enviar esas señales se utilizan variables especiales llamadas semáforos.

2. Definición de semáforo

Se puede definir como una variable de tipo semaphore. Puede ser usada para definir arrays y registros.

Un semáforo es un TAD. Consiste en una estructura de datos y un conjunto de operaciones.

Estructura de datos

- Contador: Es un entero positivo.
- Cola de procesos: Una cola de los procesos que están esperando por ese semáforo.

Operaciones

- wait(s): Si el contador del semáforo (s) es mayor que 0. se decrementa en 1 y el proceso continúa ejecutándose.
- signal(s):
  - Si el contador del semáforo (s) es igual a 0, se lleva el proceso a la cola de espera. Se suspende su ejecución y abandona el procesador.
  - Si el contador del semáforo (s) es mayor que 0, no hay ningún proceso en la cola de espera. Incrementa en 1 dicho contador y el proceso que ejecutó la operación continúa.
  - Si el contador del semáforo es igual a 0 y hay procesos esperando, se toma a uno de ellos y se le pone en un estado de preparado para ejecutarse. El proceso que ejecutó la operación continúa ejecutándose.
  - Si el contador del semáforo es igual a 0 y no hay procesos esperando, se incrementa en 1 y el proceso continúa ejecutándose.
- initial(s,valor): Establece el contador del semáforo s al valor pasado por parámetro. Si el semáforo solo admite como valores el 1 y el 0, estamos hablando de un semáforo binario. En otro caso sería un semáforo general.

Ambas operaciones (signal y wait) deben ejecutarse de forma atómica y deben ser mutuamente excluyentes. La cola de bloqueados suele ser una FIFO.

3. Resolución de problemas con semáforos

3.1. Exclusión Mutua

Usamos un semáforo binario inicializado a 1.

```
process P1;
begin
  ...
  wait(s);
  Sección Crítica;
  signal(s);
  ...
end;
```

```
process Pn;
begin
  ...
  wait(s);
  Sección Crítica;
  signal(s);
  ...
end;
```

Se utiliza un único semáforo (mutex) para las variables compartidas.

3.2. Condición de sincronización

Asignamos un semáforo general a cada condición. El valor inicial del semáforo será el de los recursos disponibles inicialmente. En cada momento, el valor del semáforo será el de los recursos disponibles en ese momento.

4. Problemas clásicos de concurrencia

4.1. Problema del productor-consumidor

Un proceso productor genera información que es usada por otro proceso consumidor. Esta comunicación se realiza a través de un buffer compartido (pila). Se debe sincronizar el proceso para que el consumidor no consuma elementos que no se han producido.

```
process producer;
var
  data: char;
begin
  for data := 'a' to 'z' do
    begin
      wait(spacesleft);
      wait(mutex);
      buffer[nextin] := data;
      nextin := (nextin + 1) mod (buffmax + 1);
      signal(mutex);
      signal(itemsready)
    end
end;
```

```
process consumer;
var
  data: char;
begin
  repeat
    begin
      wait(itemsready);
      wait(mutex);
      data := buffer[nextout];
      nextout := (nextout + 1) mod (buffmax + 1);
      signal(mutex);
      signal(spacesleft);
      write(data);
    end
  until data = 'z';
end;
```

```
begin
  initial(spacesleft,buffmax + 1);
  initial(itemsready,0);
  initial(mutex,1);
  nextin := 0;
  nextout := 0;
  cobegin
    producer;
    consumer
  coend
end.
```

4.2. Problema de los lectores-escritores

Existe un recurso que debe ser compartido por varios procesos concurrentes. Existen procesos lectores (solo quieren leer) y otros procesos escritores (solo quieren escribir). Pueden acceder lectores simultáneamente, pero un escritor necesita acceso exclusivo.

```
process type Lectores(id:integer);
begin
   repeat
      wait(mutex);
      nl:=nl+1;
      if nl=1 then wait(escritura);
      signal(mutex);
      {SECCIÓN CRÍTICA}
      wait(mutex);
      nl:=nl-1;
      if nl=0 then signal(escritura);
      signal(mutex);
   forever
end;
```

```
process type Escritores(id:integer);
begin
   repeat
      wait(escritura);
      {SECCIÓN CRÍTICA}
      signal(escritura);
   forever
end;
```

```
var
  Escritor: array[1..NESC] of Escritores;
  Lector: array[1..NLEC] of Lectores;
  i:integer;

begin
  nl:=0;
  initial(mutex,1);
  initial(escritura,1);
  cobegin
     for i := 1 to NLEC do Lector[i](i);
     for i := 1 to NESC do Escritor[i](i)
  coend
end.
```

Podemos tener prioridad en lectura o en escritura.

4.3. Problema de la comida de los filósofos

Ilustra el problema del interbloqueo. Un hilo puede obtener 2 recursos sucesivos, pero si no está disponible uno de ellos se debe quedar esperando ocupando el otro recurso. Esto produce el interbloqueo de todos los hilos.

```
process type Filosofo(id : integer);
begin
  repeat
    (* PENSANDO *)
    wait(mutex);
    while not (libres[id] and libres[(id mod N)+1]) do begin
      signal(mutex);
      wait(mutex);
    end;
    libres[id]:=false;
    libres[(id mod N)+1]:=false;
    signal(mutex);
    (* COMIENDO *)
    wait(mutex);
    libres[id]:=true;
    libres[(id mod N)+1]:=true;
    signal(mutex);
  forever
end;
```

```
var
  Filo: array[1..N] of Filosofo;
  i : integer;

begin
  for i := 1 to N do initial(tenedor[i],1);
  cobegin
    for i := 1 to N do Filo[i](i);
  coend
end.
```

5. Inconvenientes de los semáforos

- Mecanismo de bajo nivel. Puede conducir a errores.
- No es posible restringir el tipo de operaciones realizadas sobre los recursos.
- Es fácil olvidar bloquear las instrucciones de la sección crítica.
- Se usan los mismos métodos para realizar la exclusión mutua como condición de sincronización.
- Los programas con semáforos son difíciles de mantener ya que el código de sincronización está disperso por todo el código.