GIIM-ADE. Relación de problemas. Tema 2-1. 19/10/2023

Problemas de semáforos generales.

50. Sean los procesos $\mathtt{P}_1$, $\mathtt{P}_2$, y $\mathtt{P}_3$, cuyas secuencias de instrucciones son las que se muestran en el cuadro.

```
{variables globales}
Proceso P1;          Proceso P2;        Proceso P3;
  begin                begin              begin
    while true do       while true do      while true do
     begin                begin              begin
      a;                   d;                 g;
      b;                   e;                 h;
      c;                   f;                 i;
     end                  end                end
  end                  end                end
```

Se pide:resolver los siguientes problemas de sincronización, considerando que son independientes unos de otros, con semáforos:

(a)P_2 podrá pasar a ejecutar e sólo si P_1 ha ejecutado a o P_3 ha ejecutado g

(b)P_2 podrá pasar a ejecutar e sólo si P_1 ha ejecutado a y P_3 ha ejecutado g

(c)Sólo cuando P_1 haya ejecutado b, podrá pasar P_2 a ejecutar e y P_3 a ejecutar h

(d)Sincroniza los procesos de forma que las sentencias b en P_1, f en P_2, y h en P_3, sean ejecutadas como mucho por 2 procesos simultáneamente.

51. El cuadro que sigue nos muestra dos procesos concurrentes, P_1 y P_2, que comparten una variable global x y las restantes variables son locales a los procesos.
Se pide:

(a)Sincronizar los procesos para que P_1 use todos los valores x suministrados por P_2

(b)Sincronizar los procesos para que P_1 utilice un valor *sí y otro no* de la variable x, es decir, utilice los valores primero, tercero, quinto, etc. que vaya alcanzando dicha variable.

```
{variables globales}
proceso P1;                          proceso P2;
  var m: integer;                      var d: integer;
  begin                                begin
    while true do                        while true do
      begin                                begin
        m:= 2*x - n;                         d:= leer_teclado();
        print(m);                            x:= d - c*5;
      end                                  end
  end                                  end
```

52. Supongamos que estamos en una discoteca y resulta que está estropeado el servicio de chicas y todos tienen que compartir el de chicos. Se pretende establecer un protocolo de entrada al servicio usando semáforos que asegure siempre el cumplimiento de las siguientes restricciones:
    - Chicas: sólo puede estar 1 dentro del servicio
    - Chicos: pueden entrar más de 1, pero como máximo se admitirán a 5 dentro del servicio

- Versión machista del protocolo: los chicos tienen preferencia sobre las chicas Esto quiere decir que si una chica está esperando entrar al servicio y llega un chico, este puede pasar y ella sigue esperando. Incluso si el chico que ha llegado no pudiera entrar inmediatamente porque ya hay 5 chicos dentro del servicio, sin embargo, pasará antes que la chica cuando salga algún chico del servicio.
- Versión feminista del protocolo: las chicas tienen preferencia sobre los chicos. Esto quiere decir que si un chico está esperando y llega una chica, ésta debe pasar antes. Incluso si la chica que ha llegado no puede entrar inmediatamente al servicio porque ya hay una chica dentro, pasará antes que el chico cuando salga la chica que está dentro.

Se pide: implementar las 2 versiones del protocolo anterior utilizando semáforos POSIX. Ayuda sobre la sintaxis de las operaciones de los semáforos (*no nombrados*) de POSIX 1003:

inicialización : `int sem_init(sem_t* semaforo, int pcompartido, unsigned int contador)`

destrucción : `int sem_destroy(sem_t* semaforo)`

sincronización-espera : `int sem_wait(sem_t* semaforo)`

sincronización-señala : `int sem_post(sem_t* semaforo)`

Notas:

(a) El valor inicial del semáforo se le asigna a *contador*. Si *pcompartido* es distinto de cero, entonces el semáforo puede ser utilizado por hilos que residen en procesos diferentes; si no, sólo puede ser utilizado por hilos dentro del espacio de direcciones de un único proceso.

(b) Para que se pueda destruir, el semáforo ha debido ser explícitamente inicializado mediante la operación `sem_init(...)`. La operación anterior no debe ser utilizada con semáforos *nombrados*.

(c) Los hilos llamarán a la función int `sem_wait(sem_t* semaforo)`, pasándole un identificador de semáforo inicializado con el valor '0', para sincronizarse con una condición. Si el valor del semáforo fuera distinto de '0', entonces el valor de s se decrementa en una unidad y no bloquea.

(d) La operación `int sem_post(sem_t* semaforo)` sirve para señalar a los hilos bloqueadas en un semáforo y hacer que uno pase a estar preparado para ejecutarse. Si no hay hilos bloqueados en este semáforo, entonces la ejecución de esta operación simplemente incrementa el valor de la variable protegida (s) del semáforo. Hay que tener en cuenta que no existe ningún orden de desbloqueo definido si hay varios hilos esperando en la cola asociada a un semáforo, ya que la implementación a nivel de sistema de la operación anterior supone que el planificador puede escoger para desbloquear a cualquiera de los hilos que esperan. En particular, podría darse el siguiente escenario, otro hilo ejecutándose puede decrementar el valor del semáforo antes que cualquier hilo que vaya a ser desbloqueado como resultado de `sem_post(...)` lo pueda hacer y, posteriormente, se volvería a bloquear el hilo despertado.

Problemas de monitores.

53. Aunque un monitor garantiza la exclusión mutua, los procedimientos tienen que ser reentrantes. Explicar por qué.

54. 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, así mismo, 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, en algún punto en su código, tiene la necesidad de usar dos ejemplares de recursos de distinto tipo a la vez. Describe la secuencia de peticiones (llamadas al procedimiento correspondiente del monitor) 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.

55. 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.

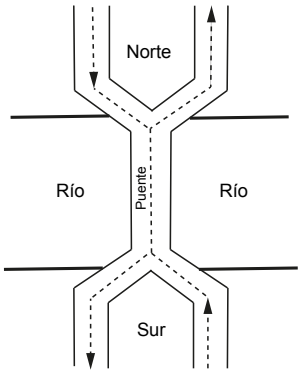

Figura 1: Problema de exclusión mutua en el acceso de coches desde 2 sentidos opuestos a un puente de un solo carril.

56. Varios coches que vienen del norte y del sur pretenden cruzar un puente sobre un río (ver Fig.1). Sólo 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.

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

    (b) Mejorar el monitor anterior, de forma que la dirección del trafico 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.

57. 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 rellenado la olla con otros M misioneros. Para solucionar la sincronización usamos un monitor llamado `Olla`, que se puede usar así:

```
monitor Olla ;
....
begin
....
end;
proceso ProcSalvaje[ i:1..N ] ;
begin
while true do begin
   Olla.Servirse_1_misionero();
   Comer(); { es un retraso aleatorio }
  end
end;
proceso ProcCocinero ;
begin
```

```
while true do begin
   Olla.Dormir();
   Olla.Rellenar_Olla();
  end
end;
```

Se pide: diseñar el código del monitor `Olla` para que se satisfaga la sincronización requerida en el enunciado del problema, teniendo en cuenta que:

- La solución propuesta no debe producir interbloqueos.
- Los salvajes podrán comer siempre que haya comida en la olla.
- Sólo se ha de *despertar* al proceso cocinero cuando la olla esté vacía.

58. 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 del monitor: (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: (b.1) colas normales (FIFO) sin prioridad y (b.2) con colas de prioridad.

59. Los procesos $\mathrm{P}_1, \mathrm{P}_2, \ldots, \mathrm{P}_\mathrm{n}$ 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 $1 \leq \mathrm{i} \leq \mathrm{n}$), donde los números menores implican mayor prioridad (es decir, si $\mathrm{i} < \mathrm{j}$, entonces P_i pasa por delante de P_j). Implementar un monitor que implemente los procedimientos `Pedir(...)` y `Liberar()` con un monitor que garantice la exclusión mutua y el acceso prioritario del procesos al recurso R.

60. 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
```

61. Este es un ejemplo clásico que ilustra el problema del interbloqueo, y aparece en la literatura informática 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 mod 5`. El monitor `MonFilo` exportará dos procedimentos: `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
  proceso Filosofo[ i: 0..4 ] ;
   begin
    while true do begin
      MonFilo.coge_tenedor(i,i); { argumento 1=codigo 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:

(a)Diseña una solución para el monitor `MonFilo`

(b)Describe la situación de interbloqueo que puede ocurrir con la solución que has escrito antes.

(c)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 intentado coger su primer tenedor.