11 presentados

PROGRAMACIÓN CONCURRENTE. EXAMEN PARCIAL 29-04-2005

1-1,50
2-0,75
3-0,75
4-0,50
5-0,75
6-0,75
P1-2,50
P2-2,50
10,00

Apellidos: PLANTILLA **Nombre:**

PRIMERA PARTE

1 **[1,5]** El algoritmo de Knuth resuelve el problema de la *exclusión mutua para n-procesos*, para lo cual utiliza n variables booleanas, `flag: array[0..n-1]of (solicitando, enSC, pasivo)`;1 variable `turn: 0..n-1` y la variable local `j`.

a) **[1,0]** Demostrar que el algoritmo de Knuth verifica todas las propiedades exigibles a un programa concurrente, incluyendo la de equidad.

b) **[0,5]** Explicar en qué mejora la condición (turn=ivflag[turn]=pasivo) de la instrucción (14) al algoritmo de Knuth original (¿qué ocurriría si se omitiese la nueva condición?)

2 **[0,75]** El código del proceso P_2 de la figura 2 no es seguro, ya que se produce interferencia entre la acción atómica del proceso P_1 y los asertos del proceso P_2. Se pide:

a) escribir los asertos que faltan en la demostración del proceso P_2,

b) de los asertos A_1, A_2, B_1, B_2 y B_3, decir cuáles de ellos son críticos y cuáles no lo son; así como, justificar el por qué,

c) sustituir la acción atómica (<...>) del proceso P_1 por una nueva acción atómica con una condición de sincronización (utilizar <await ...>), de tal forma que se evite la interferencia entre la acción atómica de P_1 y los asertos de P_2 y, de esta forma, se pueda demostrar que el programa cumple la propiedad de seguridad. Explicar el significado de dicha condición de sincronización.

Pi

```
(1) <resto instrucciones>
repeat
(2) flag[i]:= solicitando;
(3) j:= turn;
(4) while j≠i do
(5)   if flag[j]≠pasivo
(6)   then j:= turn
(7)   else j:= (j+1) mod n;
(8)   endif;
(9) enddo;
(10) flag[i]:= enSC;
(11) j:=0;
(12) while(j<n)∧(j=ivflag[j]≠enSC) do
(13)     j:=j+1 enddo;
(14)     until(j≥n)∧(turn=ivflag[turn]=
     pasivo);
(15) turn:= i;
<<seccion critica>>
(16) j:= (turn+1) mod n;
(17) while(j≠turn)∧(flag[j]=pasivo) do
(18) j:=(j+1)mod n enddo;
(19) turn:=j;
(20) flag[i]:= pasivo;
```

Figura 1

3 **[0,75]**. Escribir un programa con 1 bucle que calcule $t = \sum_{j=1}^{n} i^2$ para un valor de *n* conocido de antemano y verificar su corrección utilizando para ello el invariante de bucle $t = \frac{i\cdot(i+1)\cdot(2i+1)}{6}$ $i \in 0...n$, donde *i* se incrementa en cada iteración del bucle. Demostrar que el valor de *t* que calcula el programa coincide con el resultado de la expresión: $\frac{n\cdot(n+1)\cdot(2n+1)}{6}$.

4 **[0,5]** Explicar las condiciones necesarias para que se puedan intercambiar las señales SC (*signal-and-continue*) y el de señales SW (*signal-and-wait*) de los monitores.

5 **[0,75]** Indicar con qué tipos de señales (SX, SW, SU ó SC) sería *correcto* el monitor que se escribe más abajo. Dicho monitor se propone como una solución al problema de la planificación de trabajos conocido como SJN (*el trabajo más corto primero*); esto es, cuando se produzca la competencia entre 2 o más peticiones realizadas simultáneamente por los clientes, será servida primero aquella que en el momento de la llamada al procedimiento *petición(tiempo)* tenga el menor valor del argumento (*tiempo* es lo que se tarda en terminar el trabajo del cliente). Justificar la respuesta para cada tipo de señal y dar el invariante del monitor.

Aclaración: Sólo existe 1 procesador que ejecuta el trabajo completo de 1 cliente cada vez. La variable *libre* del monitor refleja el estado del procesador en cada momento. Los procesos clientes llaman al procedimiento *petición* del monitor para solicitar que el procesador comience a ejecutar su trabajo y después, transcurrido el valor del argumento *tiempo* desde que comenzó, llaman al procedimiento *liberar* para dejarlo libre.

Invariante: libre ⇒ turno.queue() = false

```
Monitor SJN;
  var libre: boolean;
      turno: cond (*prioritaria*);
procedure peticion(tiempo:integer);
 begin
   if (not libre) then
       turno.wait(tiempo);
   libre:= false;
end;

Procedure liberar();
 begin
   libre:= true;
   turno.signal;
end;

begin libre:= true;
end;
```

no es correcto con señales SC

Pero se puede producir un robo de señal cuando el val libre=true y se desbloquea a un proceso de la cola turno; el nuevo proceso que ejecuta petición deja libre=false → para el proceso anteriormente desbloqueado no se cumplirá la cond. de desbloqueo (libre=true)

---

(1) a) alcanzabilidad: "turn" permanece constante (cuando hay competencia entre los procesos por entrar en S.C.), de tal forma que el primer proceso en el orden cíclico "turn, turn-1, ..., 1, 0, n-1, ..., turn+1" que pase al val de clave "solicitando" consigue entrar

invariante de los procesos: se cumple, ya que al salir un proceso de S.C., éste designa al primer proceso según el turno cíclico que esté esperando como su sucesor → espera máxima de un proceso cuando todos los demás están solicitando está limitada (2^{n-1} turnos)

[0.75] Suponiendo que los procedimientos de un monitor hayan sido programados correctamente. Explicar por qué la ejecución de los procedimientos de un monitor por parte de un conjunto de procesos no limita la concurrencia de los programas (≡ ¿Por qué razón la secuencia de ejecución de un programa con 1 monitor y procesos no es equivalente, en general, a la ejecución secuencial de los procedimientos del monitor en un orden no conocido de antemano?)

PROBLEMAS:

[2,5] 1. ***Asignador de memoria SJN.*** Un asignador de memoria posee 2 operaciones `request(num_bloques:TAM_MIN..TAM_MAX,i:1..N)` , `release(cantidad: TAM_MIN..TAM_MAX)` donde `num_bloques` y `cantidad` son enteros positivos. Cuando un proceso llama al procedimiento `request`, se suspende hasta que estén disponibles en memoria un número de bloques igual o mayor al indicado en su primer parámetro. Los procesos devuelven a la memoria disponible un número de bloques igual a `cantidad` cada vez que llaman a la operación `release`. Un proceso puede devolver una cantidad de bloques diferente de la que adquirió (hacemos abstracción de que las direcciones de comienzo de las páginas de memoria podrían limitarlo).

```
Var a:array[1..n] of int;
{TOT=a[1]+...+a[n]= cte}

P1::
A1:{a[x]=X ∧ a[y]= Y}
<a[x]:= a[x]-K; a[y]:= a[y]+K>
A2:{a[x]= X-K ∧ a[y]= y+K}

P2::
Var S:=0; i:=1; Error:= false;
{S= a[1]+a[2]+ ... a[i-1]}
while i≤n do begin
  B1:{S = a[1]+a[2]+ ... + a[i-1] ∧ i≤n+1}
  S:=S+ a[i];
  B2:{S = a[1]+a[2]+ ... + a[i] ∧ i≤n+1}
  i:= i+1;
  B3:{S = a[1]+ ... +a[i-1] ∧ i≤n+1}
end; enddo;
{S= a[1]+ ... +a[n] ∧ i=n+1}

if S≠ TOT then Error:=true
```

Figura 2

Se pide: programar una solución correcta (hay que dar el invariante) al problema anterior con regiones críticas condicionales utilizando una política de atención de peticiones SJN (se atenderá primero aquella petición pendiente con menor valor de `num_bloques`; las peticiones de igual número de bloques, se atenderán en el orden en que fueron realizadas). Por ejemplo, si hay 3 peticiones pendientes P1:300B, P2:150B y P3:150B, se atenderán en el orden P2, P3, P1, cuando el planificador posea más de 600B disponibles.

Nota: para resolverlo utilizar un conjunto P de pares (num_bloques, id_proceso) ordenados por el menor valor de su primer elemento. Considerar que sobre conjunto P de pares ordenados están definidas tres operaciones de forma abstracta:

- `insert(num_bloques, id_proceso)` : inserta un nuevo par respetando el criterio de ordenación
- `remove(var id_proceso)` : devuelve el primer par de P (*primero* según el orden de menor primer elemento),
- `inspect(P)` : devuelve el número de bloques solicitado por el *primer* par de P (sin extraerlo de P)

[2,5] Suponer que hay `m` procesos productores y `n` procesos consumidores. Los productores llaman a la operación `broadcast(mem)` para enviar una copia del mensaje `men` a todos los consumidores. Para recibir una copia del mensaje, los consumidores han de llamar a `fetch(men)`, donde `men` es un argumento resultado. Escribir un monitor que implemente los procedimientos `broadcast` y `fetch` utilizando señales *de semántica no-desplazante* (SC). El monitor sólo ha de necesitar almacenar 1 mensaje pendiente de ser recibido por los consumidores cada vez que se comience a ejecutar el procedimiento `broadcast`, lo que quiere decir que cualquier nueva llamada a `broadcast`, por parte de otro proceso, ha de bloquearse hasta que el procedimiento `broadcast` actual complete su ejecución (es decir, todos los consumidores hayan recibido una copia del mensaje y se le permita terminar). Suponer que los mensajes son números enteros.

(2P)

```
Monitor BROADC;
  var entero: int; esperado: int
      enviado : boolean;
      no_recibido, esperar : cond;

procedure broadcast(mens)
begin
   while not enviado do
      no_recibido.wait();
   enviado:=true; mens:=entrega;
   esperar.signal_all();
   norecibido.wait();
   enviado:=false
   no_recibido.signal();
end;
```

```
procedure fetch(var mens)
begin
   esperado++;
   while not enviado do
      no_recibido.wait();
   esperado--;
   if(esperado=0) then norecibido.signalall();
mens:=entrega
end;
```

or !

```
while enviado do
   no_recibido.wait();
enviado:=true;
entrega:=mens;
esperar.signal_all();
no_recibido.wait();
enviado:=false;
no_recibido.signal();
```

```
esperado++;
while (not enviado) do
   no_recibido.wait();
esperando--;
if (esperado=0) then
   no_recibido.signal_all();
mem:=entrega;
```

```
Resource SJN ( memoria : 0..M ; siguiente : 0..N := 0 ; siHilo : boolean := false )

IL: siguiente = 0 => vacio(P)

procedure request ( num_bloques : TAM_MIN..TAM_MAX ; i : 1..N )
begin
    region SJN -> when not siHilo
        if (num_bloques > memoria) then insert (num_bloques, i)
            else begin
                siguiente := i;
                siHilo := true;
            end
    end region

    region SJN when ( siguiente = i or ~~siHilo~~ ) ->
        siHilo := false;
        memoria := memoria - num_bloques;
        if (inspect (P) ≤ memoria) then remove (siguiente)
                else siguiente := 0
    end region;

end;

procedure release ( cantidad : TAM_MIN..TAM_MAX)
begin
    region SJN when not siHilo ->
        memoria := memoria + cantidad;
        if (inspect(P) ≤ memoria) then remove (siguiente);
                else siguiente := 0
    end region

end;
```