Sistemas Concurrentes y Distribuidos:
Problemas Resueltos.

2016-17

Contents

Chapter 1

Problemas resueltos: Introducción

1

Considerar el siguiente fragmento de programa para 2 procesos P_1 y P_2:

Los dos procesos pueden ejecutarse a cualquier velocidad. ¿ Cuáles son los posibles valores resultantes para x ?. Suponer que x debe ser cargada en un registro para incrementarse y que cada proceso usa un registro diferente para realizar el incremento.

```
{ variables compartidas }
var x : integer := 0 ;
```

```
process P1 ;
   var i : integer ;
begin
   for i := 1 to 2 do begin
      x := x+1 ;
   end
end
```

```
process P2 ;
   var j : integer ;
begin
   for j := 1 to 2 do begin
      x := x+1 ;
   end
end
```

Respuesta

Los valores posibles son 2, 3 y 4. Suponemos que no hay optimizaciones al compilar y que por tanto cada proceso hace dos lecturas y dos escrituras de x en memoria. La respuesta se basa en los siguientes tres hechos:

- el valor resultante no puede ser inferior a 2 pues cada proceso incrementa x dos veces en secuencia partiendo de cero, la primera vez que un proceso lee la variable lee un 0 como mínimo, y la primera vez que la escribe como mínimo 1, la segunda vez que ese mismo proceso lee, lee como mínimo un 1 y finalmente escribe como mínimo un 2.
- el valor resultante no puede ser superior a 4. Para ello sería necesario realizar un total de 5 o más incrementos de la variable, cosa que no ocurre pues se realizan únicamente 4.

- existen posibles secuencias de interfoliación que producen los valores 2,3 y 4, damos ejemplos de cada uno de los casos:

  resultado 2: se produce cuando todas las lecturas y escrituras de un proceso i se ejecutan completamente entre la segunda lectura y la segunda escritura del otro proceso j. La segunda lectura de j lee un 1 y escribe un 2, siendo esta escritura la última en realizarse y por tanto la que determina el valor de x

  resultado 3: se produce cuando los dos procesos leen y escriben x por primera vez de forma simultánea, quedando x a 1. Los otros dos incrementos se producen en secuencia (un proceso escribe antes de que lea el otro), lo cual deja la variable a 3.

  resultado 4: se produce cuando un proceso hace la segunda escritura antes de que el otro haga su primera lectura. Es evidente que el valor resultado es 4 pues todos los incrementos se hacen secuencialmente.

2

¿ Cómo se podría hacer la copia del fichero `f` en otro `g`, de forma concurrente, utilizando la instrucción concurrente **`cobegin`**–**`coend`** ? . Para ello, suponer que:

- los archivos son secuencia de items de un tipo arbitrario `T`, y se encuentran ya abiertos para lectura (`f`) y escritura (`g`). Para leer un ítem de `f` se usa la llamada a función **`leer`**`(f)` y para saber si se han leído todos los ítems de `f`, se puede usar la llamada **`fin`**`(f)` que devuelve verdadero si ha habido al menos un intento de leer cuando ya no quedan datos. Para escribir un dato x en `g` se puede usar la llamada a procedimiento **`escribir`**`(g,`x`)`.
- El orden de los ítems escritos en `g` debe coincidir con el de `f`.
- Dos accesos a dos archivos distintos pueden solaparse en el tiempo.

Respuesta

Los ítems deben ser escritos en secuencia para conservar el orden, así que la lectura y la escritura puede hacerse en un bucle secuencial. Sin embargo, se puede solapar en el tiempo la escritura de un ítem leído y la lectura del siguiente, y por tanto en cada iteración se usará un **`cobegin`**–**`coend`** con la lectura solapada con la escritura.

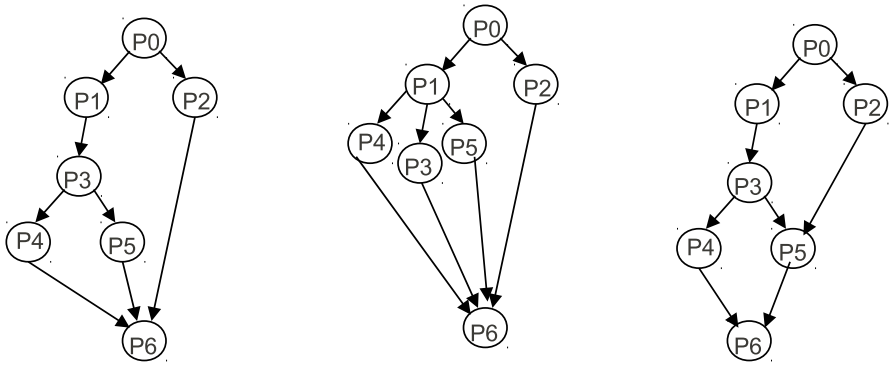
La solución más obvia sería usar una variable `v` (compartida entre la lectura y la escritura) para esto, es decir, usar en cada iteración la solución que aparece en la figura de la izquierda. El problema es que en esta solución la variable `v` puede ser accedida simultáneamente por la escritura y la lectura concurrentes, que podrían interferir entre ellas, así que es necesario usar dos variables. El esquema correcto quedaría como aparece en la figura de la derecha.

```
process Incorrecto ;
   var v  : T ;
begin
   v := leer(f) ;
   while not fin(f) do
   cobegin
      escribir(g,v);
      v := leer(f) ;
   coend
end
```

```
process Correcto ;
   var v_ant, v_sig : T ;
begin
   v_sig := leer(f) ;
   while not fin(f) do begin
      v_ant := v_sig ;
      cobegin
         escribir(g,v_ant);
         v_sig := leer(f) ;
      coend
   end
end
```

3

Construir, utilizando las instrucciones concurrentes **cobegin**–**coend** y **fork**–**join**, programas concurrentes que se correspondan con los grafos de precedencia que se muestran a continuación:



Respuesta

A continuación incluimos, para cada grafo, las instrucciones concurrentes usando **cobegin**–**coend** (izquierda) y **fork**–**join** (derecha)

(a)

```
begin
   P0 ;
   cobegin
      begin
         P1 ; P3 ;
         cobegin
            P4 ; P5 ;
         coend
      end
      P2 ;
   coend
   P6 ;
end
```

```
begin
   P0 ; fork P2 ;
   P1 ; P3 ; fork P4 ; fork P5 ;
   join P2 ; join P4 ; join P5 ;
   P6 ;
end
```

(b)

```
begin
   P0 ;
   cobegin
      begin
         P1 ;
         cobegin
            P3 ; P4 ; P5 ;
         coend
      end
      P2 ;
   coend
   P6 ;
end
```

```
begin
   P0 ; fork P2 ;
   P1 ; fork P3 ; fork P4 ; fork P5 ;
   join P2 ; join P3 ;
   join P4 ; join P5 ;
   P6 ;
end
```

(c) en este caso, **cobegin–coend** no permite expresar el simultáneamente el paralelismo potencial que hay entre `P4` y `P2` y el que hay entre `P4` y `P5`, mientras **fork–join** sí permite expresar todos los paralelismos presentes (es más flexible).

```
begin
   P0 ;
   cobegin
      begin
         P1 ; P3 ;
      end
      P2 ;
   coend
   cobegin
      P4 ; P5 ;
   coend
   P6 ;
end
```

```
begin
   P0 ;
   cobegin
      begin
         P1 ; P3 ; P4 ;
      end ;
      P2 ;
   coend
   P5 ; P6 ;
end
```

```
begin
  P0 ; fork P2 ;
  P1 ;
  P3 ; fork P4 ;
  join P2 ;
  P5 ;
  join P4 ;
  P6 ;
end
```

4

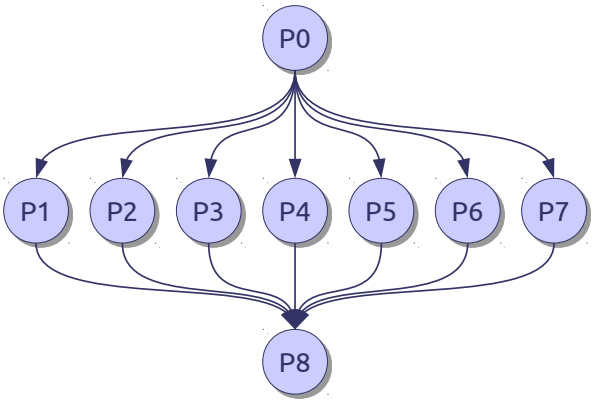
Dados los siguientes fragmentos de programas concurrentes, obtener sus grafos de precedencia asociados:

```
begin
   P0 ;
   cobegin
      P1 ;
      P2 ;
      cobegin
         P3 ; P4 ; P5 ; P6 ;
      coend
      P7 ;
   coend
   P8;
end
```

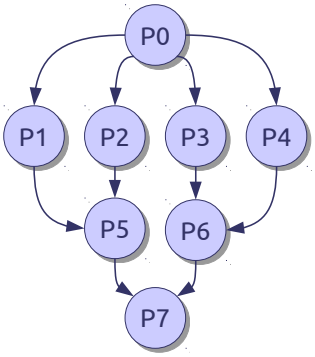
```
begin
   P0 ;
   cobegin
      begin
        cobegin
          P1;P2;
        coend
       P5;
      end
      begin
        cobegin
          P3;P4;
        coend
       P6;
      end
   coend
   P7 ;
end
```

Respuesta

En el caso a), anidar un bloque **cobegin**–**coend** dentro de otro, sin incluir ningún componente adicional en secuencia, tiene el mismo efecto que incluir directamente en el bloque externo las instrucciones del interno. Esta no es la situación en el el caso b), donde las construcciones **cobegin**–**coend** anidadas son necesarias para reflejar ciertas dependencias entre actividades.



(a)



(b)

5

Suponer un sistema de tiempo real que dispone de un captador de impulsos conectado a un contador de energía eléctrica. La función del sistema consiste en contar el número de impulsos producidos en 1 hora (cada *Kwh* consumido se cuenta como un impulso) e imprimir este número en un dispositivo de salida.

Para ello se dispone de un programa concurrente con 2 procesos: un proceso acumulador (lleva la cuenta de los impulsos recibidos) y un proceso escritor (escribe en la impresora). En la variable común a los 2 procesos `n` se lleva la cuenta de los impulsos. El proceso acumulador puede invocar un procedimiento `Espera_impulso` para esperar a que llegue un impulso, y el proceso escritor puede llamar a `Espera_fin_hora` para esperar a que termine una hora.

El código de los procesos de este programa podría ser el siguiente:

```
{ variable compartida: }
var n : integer; { contabiliza impulsos }
```

```
process Acumulador ;
begin
   while true do begin
      Espera_impulso();
      < n := n+1 > ; { (1) }
   end
end
```

```
process Escritor ;
begin
   while true do begin
      Espera_fin_hora();
      write( n ) ;    { (2) }
      < n := 0 > ;    { (3) }
   end
end
```

En el programa se usan sentencias de acceso a la variable `n` encerradas entre los símbolos < y >. Esto significa que cada una de esas sentencias se ejecuta en exclusión mutua entre los dos procesos, es decir, esas sentencias se ejecutan de principio a fin sin entremezclarse entre ellas.

Supongamos que en un instante dado el acumulador está esperando un impulso, el escritor está esperando el fin de una hora, y la variable `n` vale k. Después se produce de forma simultánea un nuevo impulso y el fin del periódo de una hora. Obtener las posibles secuencias de interfolicación de las instrucciones (1),(2), y (3) a partir de dicho instante, e indicar cuales de ellas son correctas y cuales incorrectas (las incorrectas son aquellas en las cuales el impulso no se contabiliza).

Respuesta

Supongamos que hay una variable entera (ficticia) llamada **`OUT`**, que se crea al terminar el **`write`** (sentencia (2)) y tiene el valor impreso (esto permite incluir en el estado del programa dicho valor impreso).

En el estado de partida, se cumple `n`==k, y a partir de ahí pueden ocurrir tres interfoliaciones posibles de las sentencias etiquetadas con los dígitos 1,2, y 3. Estas interfoliaciones son: **(a)** 1,2,3, **(b)** 2,1,3 y **(c)** 2,3,1.

Para cada interfoliación podemos considerar los valores de las variables en cada estado al final de cada sentencia, y podemos examinar el estado final, esto es, el valor con el que queda `n` y el valor impreso (el valor de **`OUT`**).

(a)

Instr.	n	OUT
	k	
n:= n+1	$k+1$	
write(n)	$k+1$	$k+1$
n:= 0	0	$k+1$

(b)

Instr.	n	OUT
	k	
write(n)	k	k
n:= n+1	$k+1$	k
n:= 0	0	k

(c)

Instr.	n	OUT
	k	
write(n)	k	k
n:= 0	0	k
n:= n+1	1	k

Son correctas únicamente las interfoliaciones en las cuales en el estado final se cumple:

$$\texttt{OUT} + \texttt{n} \quad == \quad k+1$$

es decir, el valor impreso más el valor de contador es igual al número total de impulsos producidos desde que comenzó la hora que acaba. Evidentemente, las interfoliaciones **(a)** y **(c)** son **correctas**, mientras que la **(b)** es **incorrecta**.

6

Supongamos un programa concurrente en el cual hay, en memoria compartida dos vectores `a` y `b` de enteros y con tamaño par, declarados como sigue:

```
var a,b : array[1..2*n] of integer ; { n es una constante predefinida }
```

Queremos escribir un programa para obtener en `b` una copia ordenada del contenido de `a` (nos da igual el estado en que queda `a` después de obtener `b`).

Para ello disponemos de la función **Sort** que ordena un tramo de `a` (entre las entradas `s` y `t`, ambas incluidas). También disponemos la función `Copiar`, que copia un tramo de `a` (desde `s` hasta `t`) en `b` (a partir de `o`)

```
procedure Sort( s,t : integer );
   var i, j : integer ;
begin
  for i := s to t do
     for j:= s+1 to t do
        if a[i] < a[j] then
           swap( a[i], b[j] ) ;
end
```

```
procedure Copiar( o,s,t : integer );
   var d : integer ;
begin
   for d := 0 to t-s do
      b[o+d] := a[s+d] ;
end
```

El programa para ordenar se puede implementar de dos formas:

- Ordenar todo el vector `a`, de forma secuencial con la función **Sort**, y después copiar cada entrada de `a` en `b`, con la función `Copiar`.
- Ordenar las dos mitades de `a` de forma concurrente, y después mezclar dichas dos mitades en un segundo vector `b` (para mezclar usamos un procedimiento **Merge**).

A continuación vemos el código de ambas versiones:

```
procedure Secuencial() ;
   var i : integer ;
begin
   Sort( 1, 2*n );    { ordena a    }
   Copiar( 1, 2*n ); { copia a en b }
end
```

```
procedure Concurrente() ;
begin
  cobegin
     Sort( 1,   n   );
     Sort( n+1, 2*n );
  coend
  Merge( 1, n+1, 2*n );
end
```

El código de **Merge** se encarga de ir leyendo las dos mitades de a, en cada paso, seleccionar el menor elemento de los dos siguientes por leer (uno en cada mitad), y escribir dicho menor elemento en la siguiente mitad del vector mezclado b. El código es el siguiente:

```
procedure Merge( inferior, medio, superior: integer ) ;
   var escribir : integer := 1 ;         { siguiente posicion a escribir en b       }
   var leer1    : integer := inferior ; { siguiente pos. a leer en primera mitad de a }
   var leer2    : integer := medio    ; { siguiente pos. a leer en segunda mitad de a }
begin
   { mientras no haya terminado con alguna mitad }
   while leer1 < medio and leer2 <= superior do begin
      if a[leer1] < a[leer2] then begin { minimo en la primera mitad }
         b[escribir] := a[leer1] ;
         leer1 := leer1 + 1 ;
      end else begin { minimo en la segunda mitad }
         b[escribir] := a[leer2] ;
         leer2 := leer2 + 1 ;
      end
      escribir := escribir+1 ;
   end
   { se ha terminado de copiar una de las mitades, copiar lo que quede de la otra }
   if leer2 > superior then Copiar( escribir, leer1, medio-1  ); { copiar primera }
                       else Copiar( escribir, leer2, superior ); { copiar segunda }
end
```

Llamaremos $T_s(k)$ al tiempo que tarda el procedimiento **Sort** cuando actua sobre un segmento del vector con k entradas. Suponemos que el tiempo que (en media) tarda cada iteración del bucle interno que hay en **Sort** es la unidad (por definición). Es evidente que ese bucle tiene $k(k-1)/2$ iteraciones, luego:

$$T_s(k) = \frac{k(k-1)}{2} = \frac{1}{2}k^2 - \frac{1}{2}k$$

El tiempo que tarda la versión secuencial sobre $2n$ elementos (llamaremos S a dicho tiempo) será evidentemente $T_s(2n)$, luego

$$S = T_s(2n) = \frac{1}{2}(2n)^2 - \frac{1}{2}(2n) = 2n^2 - n$$

con estas definiciones, calcula el tiempo que tardará la versión paralela, en dos casos:

(1) Las dos instancias concurrentes de **Sort** se ejecutan en el mismo procesador (llamamos P_1 al tiempo que tarda).

(2) Cada instancia de **Sort** se ejecuta en un procesador distinto (lo llamamos P_2)

escribe una comparación cualitativa de los tres tiempos (S,P_1 y P_2).

Para esto, hay que suponer que cuando el procedimiento **Merge** actua sobre un vector con p entradas, tarda p unidades de tiempo en ello, lo cual es razonable teniendo en cuenta que en esas circunstancias **Merge** copia p valores desde `a` hacia `b`. Si llamamos a este tiempo $T_m(p)$, podemos escribir

$$T_m(p) \quad = \quad p$$

.

Respuesta (privada)

(1) Sobre un procesador el coste total de la versión paralela (P_1) sería el de dos ordenaciones secuenciales de n elementos cada una, (es decir $2T_s(n)$), más el coste de la mezcla secuencial (que es $T_m(2n)$), esto es:

$$P_1 \quad = \quad 2T_s(n) + T_m(2n) \quad = \quad \left(n^2 - n\right) + 2n \quad = \quad n^2 + n$$

Si comparamos $P_1 = 2n^2 - n$ con $S = n^2 + n$, vemos que, aun usando un único procesador en ambos casos, para valores de n grandes la versión potencialmente paralela tarda la mitad de tiempo que la secuencial.

(2) Sobre dos procesadores, el coste de la versión paralela (P_2) será el de la ejecución concurrente de dos versiones de **Sort** iguales sobre `n` elementos cada una, por tanto, será igual a $T_s(n)$. Después, la mezcla se hace en un único procesador y tarda lo mismo que antes, $T_m(2n)$, luego:

$$P_2 \quad = \quad T_s(n) + T_m(2n) \quad = \quad \left(\frac{1}{2}n^2 - \frac{1}{2}n\right) + 2n \quad = \quad \frac{1}{2}n^2 + \frac{3}{2}n$$

ahora vemoa que (de nuevo para n grande), el tiempo P_2 es aproximadamente la mitad de P_1, como era de esperar (ya que se usan dos procesadores), y por supuesto P_2 es aproximadamente la cuarta parte de S.

7

Supongamos que tenemos un programa con tres matrices (`a`,`b` y `c`) de valores flotantes declaradas como variables globales. La multiplicación secuencial de `a` y `b` (almacenando el resultado en `c`) se puede hacer mediante un procedimiento **MultiplicacionSec** declarado como aparece aquí:

```
var a, b, c : array[1..3,1..3] of real ;

procedure MultiplicacionSec()
   var i,j,k : integer ;
begin
   for i := 1 to 3 do
      for j := 1 to 3 do begin
         c[i,j] := 0 ;
         for k := 1 to 3 do
            c[i,j] := c[i,j] + a[i,k]*b[k,j] ;
      end
end
```

Escribir un programa con el mismo fin, pero que use 3 procesos concurrentes. Suponer que los elementos de las matrices `a` y `b` se pueden leer simultáneamente, así como que elementos distintos de `c` pueden escribirse simultáneamente.

Respuesta (privada)

Para implementar el programa, haremos que cada uno de esos 3 procesos concurrentes (llamados **`CalcularFila`**) calcule y escriba un conjunto distinto de entradas de `c`. Por simplicidad (y equidad entre los procesos), lo más conveniente es hacer que cada uno de ellos calcule una fila de `c` (o cada uno de ellos una columna)

```
var a, b, c : array [1..3,1..3] of real ;

process CalcularFila[ i : 1..3 ] ;
   var j, k : integer ;
begin
   for j := 1 to 3 do begin
      c[i,j] := 0 ;
      for k := 1 to 3 do
         c[i,j] := c[i,j] + a[i,k]*b[k,j] ;
   end
end
```

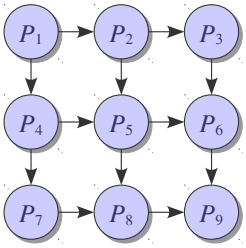
8

Un trozo de programa ejecuta nueve rutinas o actividades ($P_1, P_2, \ldots, P_9$), repetidas veces, de forma concurrentemente con **`cobegin coend`** (ver la figura de la izquierda), pero que requieren sincronizarse según determinado grafo (ver la figura de la derecha):

Trozo de programa:

```
while true do
cobegin
    P1 ; P2 ; P3 ;
    P4 ; P5 ; P6 ;
    P7 ; P8 ; P9 ;
coend
```

Grafo de sincronización:



Supón que queremos realizar la sincronización indicada en el grafo, usando para ello llamadas desde cada rutina a dos procedimientos (**EsperarPor** y **Acabar**). Se dan los siguientes hechos:

- El procedimiento **EsperarPor**(i) es llamado por una rutina cualquiera (la número k) para esperar a que termine la rutina número i, usando espera ocupada. Por tanto, se usa por la rutina k al inicio para esperar la terminación de las otras rutinas que corresponda según el grafo.
- El procedimiento **Acabar**(i) es llamado por la rutina número i, al final de la misma, para indicar que dicha rutina ya ha finalizado.
- Ambos procedimientos pueden acceder a variables globales en memoria compartida.
- Las rutinas se sincronizan única y exclusivamente mediante llamadas a estos procedimientos, siendo la implementación de los mismos completamente transparente para las rutinas.

Escribe una implementación de **EsperarPor** y **Acabar** (junto con la declaración e inicialización de las variables compartidas necesarias) que cumpla con los requisitos dados.

Respuesta (privada)

Una posible solución consiste en usar un vector de valores lógicos que indican si cada proceso ha terminado o no. Hay que tener en cuenta que, puesto que la ejecución concurrente de todas las rutinas está en un bucle, dicho vector debe reinicializarse entre una iteración del bucle y la siguiente. Para ello realizamos dicha reinicialización cuando el proceso 9 (el último) señale que ha acabado (en **Acabar**). La implementación queda como sigue:

```
{ compartido entre todas las tareas }
var finalizado : array [1..9] of boolean := (false,false,..., false) ;
```

```
procedure EsperarPor( i : integer )
begin
   while not finalizado[i] do
     begin end
end
```

```
procedure Acabar( i : integer )
var j : integer ;
begin
   if i < 9 then
      finalizado[i] := true ;
   else
      for j := 1 to 9 do
         finalizado[j] := false ;
end
```