Programación Concurrente
Examen para la casa

J. Blanco, N. Wolovick

11 de julio de 2003

1. [5, ej.24]

   a) Para el siguiente multiprograma de N componentes, con $2 \leq N$, demuestre que el número de componentes ejecutando los fragmentos S es a lo más $N-1$.

Pre: $true$
Comp.p: $*$ [$v := p$; **if** $v \neq p \rightarrow$ **skip fi** ; $S.p$]

   b) Demuestre que la misma propiedad vale en el siguiente multiprograma:

Pre: $\langle \forall q :: \neg x.q \rangle$
Comp.p: $*$ [$x.p := true$; **if** $\langle \exists q : q \neq p : \neg x.q \rangle \rightarrow$ **skip fi** ; $S.p$; $x.p := false$]

   c) Explique si la situación se modifica cuando cambiamos las componentes a[1]:

Pre: $\langle \forall q :: \neg x.q \rangle$
Comp.p: $*$ [$x.p := true$; $v := p$; **if** $v \neq p \vee \langle \exists q : q \neq p : \neg x.q \rangle \rightarrow$ **skip fi** ; $S.p$; $x.p := false$]

2. [5, ej.32] Elimine las multiasignaciones utilizando Fortalecimiento de Guarda.

Pre: $x \wedge \neg y \wedge \neg z$		
A: $*$[**if** $x \rightarrow$ **skip fi**; $x, y := false, true$]	B: $*$[**if** $y \rightarrow$ **skip fi**; $y, z := false, true$]	C: $*$[**if** $z \rightarrow$ **skip fi**; $z, x := false, true$]

[1] la atomicidad es línea a línea.

3. [2, ej.4.32][4] Tres tipos de procesos comparten el acceso a una lista ligada: los que buscan (*searchers*), los que insertan (*inserters*) y los que borran (*deleters*). Los searchers sólo examinan la lista, por lo tanto pueden ejecutarse concurrentemente con sus pares. Los inserters agregan nuevos elementos al final de la lista, luego deben estar en exlusión mutua para prevenir que dos de ellos inserten nuevos datos al mismo tiempo. Sin embargo, un inserter puede proceder de forma paralela con cualquier cantidad de searchers. Finalmente, los deleters quitan datos de cualquier lugar de la lista, por lo tanto sólo puede haber un deleter a la vez el cual debe ser mutuamente exclusivo con searchers e inserters.

   a) Obtenga una especificación formal de este requerimiento

   b) Obtenga programas que satisfagan el punto anterior utilizando la técnica del semáforo binario dividido (*SBS*).

   c) Simplifique lo más posible los multiprogramas resultantes.

   d) Analice la posibilidad de otorgar prioridad a alguno de los tipos de procesos reduciendo el no-determinismo de las guardas de salida.

   e) Resolver este problema usando alguna clase de **monitores**.

4. Durante la materia vimos que incorporar algunas instrucciones de granularidad ligeramente mayor a *1-point*, simplificaba enormemente ciertos algoritmos de sincronización, como por ejemplo los de inclusión y exclusión mutua.

   Sin embargo, estas instrucciones *fetch-and-*Φ agregan complejidad al pipeline del microprocesador por el ciclo *read-modify-write*, y latencia a los protocolos de consistencia de cache y de interconexión del microprocesador[7, p.695].

   En [6] se presenta una nueva técnica actualmente utilizada en microprocesadores *RISC*, denominada **sincronización optimista**, completamente análoga al control de concurrencia optimista en el área de transacciones en bases de datos [3]. Esta técnica consiste en tener el par de instrucciones *load-linked*, *store-conditional*, de las cuales podemos observar si fueron ejecutadas de manera atómica[7, 6].

   A partir de este par de instrucciones podemos construir sentencias de mayor granularidad que *1-point*, como por ejemplo el incremento atómico y el *test-and-set*.

```
atomic_inc
try:  LL(R0,n)
      INC(R0)
      SC(n,R0)
      BZ try
```

```
test_and_set
try:  LL(R0,lock)
      OR(R1,R0,1)
      SC(lock,R1)
      BZ try
```

   Para representar estas instrucciones en nuestro formalismo podemos realizar algunas simplificaciones a fin de no inundar con detalles irrelevantes. Por ejemplo en el siguiente multiprograma que ***deberán demostrar correcto***, al tener sólo una variable compartida, invalidamos el *link register* sólo por cambios de contexto, y no por usar diferentes direcciones de memoria para el *SC*.

$Pre : n = 0 \wedge p_0 \neq p_1$	
P_0: $s_0 := false;$ **do** $\neg\ s_0 \rightarrow$ $\langle a_0 := n; lr := p_0 \rangle;$ $a_0 := a_0 + 1;$ $\langle$ **if** $lr = p_0 \rightarrow s_0, n := true, a_0$ $\Box\ lr \neq p_0 \rightarrow s_0 := false$ **fi**$\rangle$ **od**	P_1: $s_1 := false;$ **do** $\neg\ s_1 \rightarrow$ $\langle a_1 := n; lr := p_1 \rangle;$ $a_1 := a_1 + 1;$ $\langle$ **if** $lr = p_1 \rightarrow s_1, n := true, a_1$ $\Box\ lr \neq p_1 \rightarrow s_1 := false$ **fi**$\rangle$ **od**
$Post : n = 2$	

Notar que si demostramos este programa estamos estableciendo condiciones necesarias para la equivalencia entre los P_i y $\langle n := n + 1 \rangle$

Como ejercicio complementario escriba *test-and-set* en nuestro formalismo y realice alguna demostración que muestre que son intercambiables en el contexto de un multiprograma, como por ejemplo el problema de la sección crítica.

Referencias

[1] Gregory R. Andrews, *Concurrent Programming: principles and practice*, Benjamin Cummings, 1991

[2] Gregory Andrews, *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison Wesley, 2000

[3] P. A. Bernstein and V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, Massachussetts, 1987.

[4] Edsger W. Dijkstra, *A Tutorial on the Split Binary Semaphore*, EWD703, 1979

[5] W.H.J. Feijen and A.J.M. van Gasteren, *On a Method of MultiProgramming*, Monographs in Computer Science, Springer, 1999

[6] Jensen E. and Hagensen G. and Broughton J. *A new approach to exclusive data access in shared memory multiprocessors*, Technical Report UCRL-97663, Lawrence Livermore National Laboratory, Nov 1987.

[7] John L. Hennessy and David Goldberg and David A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 2nd Edition, 1996

[8] Rob Hoogerwoord, *Semaphores*, RH86

[9] Rob Hoogerwoord, *Two Applications of the Split Binary Semaphore*, RH221

[10] F. Schneider, *On Concurrent Programming*, Graduate Texts in Computer Science, Springer, 1997