Programación Concurrente
Práctico 7: Exclusión Mutua

J. Blanco, N. Wolovick

15 de mayo de 2003

1. Dado el siguiente multiprograma, muestre la exclusión mutua de los fragmentos $S.i$[1].

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

2. Considere un multiprograma con 2 componentes de tipo A, donde en los fragmentos cs no se menciona x.

Pre: x
$A : *[\ \mathbf{if}\ x \rightarrow x := false\ \mathbf{fi}$ $;\ cs$ $;\ x := true$ $]$

   a) Muestre exclusión mutua entre los fragmentos cs.

   b) Muestre la ausencia de deadlock total.

   c) ¿Hay progreso individual?

   d) Extienda las componentes utilizando solamente variables nuevas y skips guardados, de forma tal que el número de fragmentos cs que se ejecuten en una componente difieran a lo más en 7 respecto a la otra componente[2].

3. Considere el siguiente multiprograma con un número arbitrario de componentes[3]:

Pre: $m = 0 \wedge n = 0$
$Comp.p : *[\ ncs.p$ $;\ x.p, m := m, m+1$ $;\ \mathbf{if}\ x.p = n \rightarrow \mathbf{skip\ fi}$ $;\ cs.p$ $;\ n := n+1$ $]$

[1] Probablemente resulte más fácil derivarlo como una generalización del *Safe Sluice* que intentar una verificación.

[2] Es decir, pongale un poco de justicia a este algoritmo.

[3] Este es una variación de muy gruesa granularidad del Algoritmo del Panadero de Lamport [Lam74]. Intente derivar el algoritmo empezando desde la aserción $\{x.p = n?\}$ como pre de $cs.p$.

a) Muestre exclusión mutua entre los fragmentos *cs*.
b) Muestre ausencia de deadlock total.
c) Indique si hay progreso individual.

4. La complejidad de la solución del Problema de Exclusión Mutua para N procesos disminuye significativamente cuando la granularidad es ligeramente mayor a sentencias *one-point*.

   Por ejemplo con alguna de las siguientes instrucciones atómicas fetch-and-Φ[4] se pueden lograr algoritmos de mutex muy compactos.

   - $\mathit{TestSet}(v, p) : v, p := \mathit{true}, v$
   - $\mathit{FetchAdd}(w, p, e) : p, w := w, w + e$
   - $\mathit{TSL}(v, p) : v, p := \mathit{false}, v$
   - $\mathit{Swap}(v, w) : v, w := w, v$
   - $\mathit{CompareSwap}(t, x, o, n) : \langle t := (o = x); \mathbf{if}\ t \rightarrow x := n \ \square\ \neg t \rightarrow o := x\ \mathbf{fi}\ \rangle$
   - $\mathit{FetchOr}(w, p, e) : p, w := w, w \vee e$
   - $\mathit{Decr}(v, s) : v, s := v - 1, (v - 1 = 0)$

   a) Utilice *TestSet* para lograr un algoritmo de mutex entre N procesos.
   b) Demuestre la corrección para N cualquiera[5] .
   c) Investigue que alternativas existen para mejorar la performance de este algoritmo en máquinas multiprocesador con caches locales. Incluya un breve argumento que muestre que la corrección del punto anterior se aplica en estas optimizaciones.
   d) Elija alguna otra *interlock instruction*[6] y proponga el algoritmo de mutex.

5. Desarrolle una solución al problema de mutex de N procesos utilizando *Swap*.

   a) Demuestre la corrección.
   b) Optimice para que funcione bien en arquitecturas *MP* con cache. Explique el porque de las optimizaciones y como se mantiene la corrección.
   c) Utilizando *Swap* desarrolle una solucion *fair* al problema de la sección crítica, es decir que asegure que eventualmente todos los procesos en espera podrán entrar. La clave está en ordenar los procesos en, por ejemplo, orden *FIFO*. Asuma que cada proceso tiene una identidad única, es decir un entero en el rango $[0, N)$. Argumente de manera clara porque su solución es correcta y justa.

[4] También llamadas *inter lock instructions*.
[5] La demostración del punto 2 puede resultar útil.
[6] Menos *TSL*.