Sincronització de fils: espera activa

Lluís Garrido

Octubre 2014

Algorisme 1

A la transparència es mostra el codi associat a dos fils. Cada fil comprova si l'altre fil ha entrat a la secció crítica. Per exemple, suposem que el fil 0 vol entrar a la secció crítica i que el fil 1 no es troba a l'interior d'aquest. El fil 0 crida a la funció *lock*, comprova que el fil 1 no es troba a la secció crítica i posa la seva variable *flag* a *true*. La funció *lock* retorna i el fil 0 pot procedir a executar la secció crítica. Si en aquest moment el fil 1 crida a la funció *lock* es quedarà esperant al *while* fins que el fil 0 surti de la secció crítica. En fil 0 surt de la secció cridant a *unlock* i posant la seva variable *flag* a *false*. El fil 1 veurà que el fil 0 ja no es troba a la secció crítica i podrà entrar-hi.

Observar que aquest algorisme no impedeix que els dos fils puguin entrar a la secció crítica. Per això comencem igual que abans: el fil 0 crida a la funció *lock* i executa la condició del *while*. Ja que el fil 1 no és a l'interior de la secció crítica, la condició del *while* és falsa i per tant se sortirà de seguida del *while*. Suposem que després del *while* i abans de posar el flag a *true* es fa un canvi de context al fil 1. El fil 1 també vol entrar a la secció crítica: crida a *lock,* comprovarà que el fil 0 no es troba a la secció crítica i hi entrarà. Un cop a dins de la secció crítica es pot produir un canvi de context al fil 0. Aquest posa el seu *flag* a *true* i entra a la secció crítica. Tenim doncs els dos fils a l'interior de la secció crítica. Tot depèn de com es produeix el canvi de context. Per tant aquest algorisme no proveeix exclusió mútua.

Algorisme 2

Hem invertit les dues instruccions de la funció *lock*. Aquest algorisme satisfà exclusió mútua ja que només un dels fils podrà entrar a la secció crítica. El problema en aquest algorisme es que es pot produir un *deadlock*, fent que els dos fils es quedin de forma indefinida comprovant la condició d'entrada. En particular, suposem que el fil 0 posa la seva variable a *true* i que en aquest moment (abans d'executar el *while*) es produeix un canvi de context al fil 1. Suposem que el fil 1 també vol entrar a la secció crítica: posa la seva variable a *true* i es queda esperant al *while* que el fil 0 surti de la secció crítica. A continuació hi ha un altre canvi de context al fil 0, que es posarà a executar el *while* esperant que el fil 1 surti de la secció crítica. És a dir, tots dos fils, el 0 i l'1, es quedaran esperant de forma indefinida que l'altre fil surti de la secció crítica, cosa que no passarà mai. Els dos fils es quedaran doncs "penjats" de forma indefinida.

Algorisme 3

Aquest és l'anomenat algorisme de *Peterson*, un dels algorismes clàssics de sincronització de dos fils. És un algorisme que proveeix exclusió mútua i a més no té *deadlock*. La variable *victima*, compartida entre els fils, s'utilitza per decidir quin dels dos fils entra a la secció crítica en cas que tots dos vulguin entrar al mateix temps. El fil senyalitza que surt de la secció crítica posant la variable *flag* a *false*.

Algorisme 4

Invertim les dues instruccions abans del *while* i veiem que aquest algorisme ja no permet obtenir exclusió mútua. En particular, suposem que al fil 1 executem *victima* =1 i fem un canvi de context al fil 0. Aquest executa *victima* = 0 i *flag*[0] = *true* i entra a la secció crítica. Un cop el fil 0 és a l'interior de la secció crítica fem un canvi de context al fil 1, que executa *flag*[1] = *true* i també podem entrar a la secció crítica.

Sembla doncs que només hem d'anar amb compte a l'hora de programar la funció de *lock*. Però no es així, ja que l'algorisme 3 tampoc funciona a les màquines multiprocessadores actuals. Les raons són que 1) el compilador (*gcc*, *java*) pot reordenar les instruccions per fer el codi més eficient, 2) un processador pot decidir reordenar les instruccions que executa, i 3) en un sistema multiprocessador les operacions d'escriptura es fan memòria cau i no tenen perquè ser visibles per la resta de processadors en el moment de realitzar-se l'operació d'escriptura.

És per això que els sistemes multiprocessadors actuals inclouen instruccions màquina específiques per la sincronització de fils. Entre aquestes instruccions hi trobem la barrera de memòria (*memory fence* en anglès). Aquesta instrucció fa que totes les instruccions d'escriptura que s'hagin produït en aquell processador es realitzin i s'escriguin realment a memòria en el moment d'executar la instrucció de barrera de memòria. A més de les instruccions de barrera, també hi ha operacions atòmiques que s'han dissenyat específicament per sincronitzar múltiples fils. Una d'aquestes és la instrucció *Get-and-Set* que es veurà a la següent transparència.

A l'actualitat, totes les funcions de sincronització de fils (espera activa, semàfors, monitors) que utilitzarem per programar (en el nostre cas les funcions *lock* i *unlock*) inclouen les barreres i les instruccions atòmiques necessàries perquè funcionin correctament.

Algorisme 5

L'algorisme utilitza la instrucció atòmica *Get-And-Set*. Aquesta instrucció té per paràmetre una adreça de memòria: s'emmagatzema el valor inicial en una variable, es modifica el seu valor a *true* i es retorna el valor inicial. Tot això sense que la instrucció pugui ser interrompuda. En sistemes multiprocessadors tipus *Intel* i *AMD*, una forma d'implementar aquesta instrucció a la CPU és fer que la CPU bloquegi el bus mentre estigui fent l'operació de forma que altres CPUs no puguin accedir al bus.

La funció *lock* comprova si el fil pot entrar a la secció crítica. Suposem que no hi ha ningú a la secció crítica. En cridar el primer fil a la funció *lock*, la funció *Get-And-Set* retornarà *false* i posarà *flag* a *true* (tot això de forma atòmica). El fil podrà entrar la secció crítica i tota la resta de fils que cridin a *lock* es quedaran esperant al *while*. En sortir el fil de la secció crítica posarà el *flag* al *false* (seguit d'una barrera de memòria) de forma que tots els fils que estiguin esperant al *while* de la funció *lock* competiran per entrar a la secció. Només un ho aconseguirà.

Algorisme 6

A l'algorisme anterior no tenim cap forma de controlar quin és l'ordre en què els fils entraran a la secció crítica. En aquest algorisme, el 6, cada fil té un torn, de forma similar als torns utilitzats en els supermercats. A mesura que els fils arriben a la funció *lock* se'ls assigna un número, que és únic, que li indica quan podrà procedir per entrar a la secció crítica. La variable *torn* fa referència al número que sortirà quan un fil demani tanda per entrar a la secció crítica, mentre que *torn_actual* fa referència al número que actualment se serveix, és a dir, al fil es troba a dins de la secció crítica. Quan un fil surt de la secció crítica, posa el seu *flag* a *false* i dóna el torn al següent fil. Si hi ha algun fil esperant, aquest podrà entrar-hi.