Licenciatura en Ciencias de la Computación, Facultad de Ciencias, UNAM.
Computación concurrente.
Profesor: Carlos Zerón Martínez.
Ayudante: Manuel Ignacio Castillo López.

Notas de clase

Algoritmo de Kessels

Consideremos el algoritmo de *Peterson* y las modificaciones que propuso J. L. W. Kessels en 1982. Las modificaciones de Kessels se basan en las siguientes ideas:

- Alterar el uso del registro *turno* que usa Peterson en **dos procesos que son capaces de leerlo y escribirlo**.
- Vamos a garantizar que sólo uno de los dos procesos puede escribir en un momento dado; ie. **Garantiza exclusión mutua**. La idea es resolver exclusión mutua garantizando que sólo un proceso pueda manipular el *turno*; en lugar de concentrarse en sólo proteger la sección crítica.
- Usamos dos registros: *turn[0]* y *turn[1]*, (sólo toman valor 1 ó 0).
- También usamos otros dos registros booleanos *b[0]* y *b[1]*.
- El proceso P0 puede **escribir** en los registros *turn[0]* y *b[0]*; y **leer** en los registros *turn[1]* y *b[1]*.
- El proceso P1 puede **escribir** en los registros *turn[1]* y *b[1]*; y **leer** en los registros *turn[0]* y *b[0]*.
- *La idea es emular el algoritmo de Peterson, cambiando el registro **turn** por los registros **turn[0]** y **turn[1]** de la siguiente manera:*
    - turn = 0 si y sólo si turn[0] = turn[1]
    - turn = 1 si y sólo si turn[0] =/= turn[1].

Este tipo de algoritmos que usan un sólo registro para escribir, pueden ser fácilmente implementados en una red basada en paso de mensajes.

Algoritmo de Kessels

Adicionalmente a los mencionados, el algoritmo usa un par de registros locales; uno para cada proceso: *local[0]* que sólo existe para P0 y *local[1]* que sólo existe para P1. Estos registros locales, de la misma forma que los *turn*, sólo pueden tomar 0 o 1 como valor.

Condiciones iniciales: b[0] = b[1] = false y los valores iniciales de turn[0] y turn[1] no importan; como tampoco importan los valores iniciales de local[0] ni local[1]:

	P0	P1
1.	b[0] ← true	b[1] ← true
2.	local[0] ← turn[1]	local[1] ← 1- turn[0]
3.	turn[0] ← local[0]	turn[1] ← local[1]
4.	await(b[1] = false | local[0] =/= turn[1]);	await(b[0] = false | local[1] = turn[0]);
5.	{sección crítica}	{sección crítica}
6.	b[0] ← false	b[1] ← false

- Observe que podemos reescribir la instrucción 2 del proceso P1 como local[1] ← ¬turn[0], si pensamos en los registros turn cómo booleanos.
- También podríamos reescribir las esperas ocupadas de ambos proceso (instrucciones 4) cómo await(b[i] = false | local[i] =/= (turn[1 -i] +i) mod 2).
- Este algoritmo es libre de hambruna, pues ninguno de los procesos se encerrará en las esperas ocupadas, puesto que ocurre siempre alguna de las siguientes:
    - turn[0] = turn[1] ó
    - turn[1] =/= turn[0]
    - Cuando uno de los procesos termina de ejecutar su sección crítica, le cede el turno al otro al hacer b[i] ← false
- Por las mismas razones de que los registros en turn siempre o serán iguales o diferentes; sin importar el estado de los procesos, el algoritmo de Kessels también es libre de abrazos mortales.
- El valor de fragmentar la bandera *turn* original del algoritmo de Peterson en dos, nos ayuda a considerar y evitar el problema de que la escritura y lectura de memoria no necesariamente se realiza como una instrucción atómica, y esto permite una revalidación del apropiamiento de la sección crítica. Esto se aprecia mejor con la inclusión de los registros locales *local*; que nos ayudan a forzar una relectura de los registros compartidos *turn*.

Pruebas de integridad del algoritmo de Kessels

El algoritmo de Kessels satisface exclusión mutua.

Queremos mostrar que sólo uno de los dos procesos; P0 y P1, puede ejecutar su sección crítica en un momento dado. Para ello, ambos deberían poder de superar su código de entrada al mismo tiempo; o cuando uno de ellos no ha terminado de ejecutar su SC. Esto significa que ninguno ha llegado a su código de salida.

Con lo anterior en mente, vamos a revisar el código de entrada de ambos procesos para comprobar que en efecto no puede ser superado por ninguno de los dos procesos en ningún caso; mientras uno de ellos esté ejecutando su sección crítica.

Observemos que pasa cuando cualquiera de los dos procesos llega primero a su sección crítica. Esto significa que ocurre cualquiera de las siguientes: b[i] es falso, local[0] =/= turn[1] ó local[1] = turn[0]. El caso donde b[i] es falso; podemos generalizarlo a los otros dos casos, puesto que por lo expuesto anteriormente nos interesa que ambos estén ejecutando su código de entrada (compitiendo por acceder a su SC) y comprobar que forzosamente uno de los dos deberá quedarse allí hasta que el otro termine de ejecutar su SC y por el hecho de que la primera instrucción del código de entrada de ambo procesos es asignar b[i] a verdadero.

Veamos los posibles casos:

1. Empecemos por suponer que P0 ha logrado ejecutarse hasta su SC mientras P1 aún ejecuta su código de entrada. Si P0 entró a su SC, es porqué local[0] =/= turn[1]. Como los registros que maneja el algoritmo de Kessels son binarios, entonces esto significa que inicialmente local[0] = 1 -turn[1]; pero esto no nos da

ninguna información de local[0] con respecto a turn[0] (y por ende, tampoco nos da información sobre cómo son los turn entre sí). Veamos las posibilidades:

a. Si P1 por alguna razón no ha alcanzado la instrucción donde invierte el valor de turn[0], considerando que local[0] =/= turn[1]; obtenemos que local[0] = turn[0] =/= turn[1] (pues P0 ya ha ejecutado su código de entrada). Cuando P1 renanude, asignará local[1] = turn[1]; pues ambos registros local son binarios y si turn[0] =/= turn[1], al invertir el valor de turn[0] se obtiene el valor de turn[1]. Así, al asignar turn[1] = local[1], no se altera el valor de turn[1]; por lo que se sigue satisfaciendo qué turn[0] =/= turn[1] y por tanto es falso que local[1] = turn[0] y P1 deberá permanecer en la espera ocupada.

b. Si P1 ya pasó la instrucción donde debió invertir el valor de turn[0] en el pasado y lo hizo antes de que P0 asignará el valor de turn[1] en local[0]; P1 debe ser interrumpido antes de asignar local[1] en turn[1] mientras P0 alcanza la espera ocupada, de forma que pueda satisfacerse que que local[0] =/= turn[1]; pues de otra forma no sería posible que P0 esté ejecutando su sección crítica antes que P1. Si P0 continúa ejecutando su SC cuando P1 se reanuda, P1 asigna el valor inverso de turn[0]; almacenado en local[1], en turn[1] y al preguntar si local[1] = turn[0] será falso; pues turn[0] = local[0] =/= turn[1] = local[1] y P1 deberá permanecer en la espera ocupada.

Con lo anterior tenemos contemplados los posibles escenarios donde P0 ejecuta su sección crítica mientras P1 lo intenta y hemos mostrado que se quedará intentándolo: los estados de los registro local y turn le impiden a P1 salir de la espera ocupada y acceder a su sección crítica. Pero ¿qué pasa si P1 llega primero a ejecutar su SC?

2. Supongamos ahora que P1 está ejecutando su sección crítica mientras P0 sigue ejecutando su código de entrada para también acceder a su sección crítica. En este caso, debe satisfacerse que local[1] = turn[0]. Veamos los posibles casos de nuevo:

   a. Si P0 por alguna razón no ha alcanzado la instrucción donde asigna el valor de turn[1] en local[0], cuando lo haga y continúe obtendremos que turn[0] = local[0] = turn[1]. Considerando que local[1] = turn[0], entonces local[1] = local[0] = turn[1] por lo que es falso que local[0] =/= turn[1] y P0 deberá permanecer en la espera ocupada.

   b. Cuando P0 logra asignar turn[1] en local[0] antes de que P1 asigne el valor de local[1] en turn[1]; que contiene el inverso del original turn[0], tenemos que local[0] = turn[1] y además local[1] =/= turn[0]. Si P1 continúa y asigna local[1] en turn[1], entonces turn[0] =/= turn[1] y después P0 asigna local[0] en turn[0] y para que P1 pueda acceder a su sección crítica antes que P0, debe cumplirse que inicialmente qué turn[0] =/= turn[1], de esta forma el valor que asigna P0 en este paso sí altera el valor de turn[0] y por lo anterior turn[0] = turn[1]. Así, P0 ve en la condición de permanencia de la espera ocupada que

local[0] = turn[1] y espera, mientras que P1 evalúa que local[1] = turn[0] y ejecuta su SC.

Con lo anterior vemos que sólo uno de los dos procesos ejecuta su sección crítica a la vez. En los casos 1.b y 2.b; si no se satisfacen las condiciones iniciales señaladas, la única diferencia es que se ejecuta la sección crítica del proceso puesto (P1 para 1.b y P0 para 2.b), pero el otro no podrá hacerlo.

Por tanto, el algoritmo de Kessels satisface exclusión mutua.

El algoritmo de Kessels es libre de hambruna.

Siguiendo los razonamientos anteriores, veamos que no hay forma en la que un proceso se quede esperando al otro y no pueda continuar su ejecución. Extendiendo los casos anteriores:

1. P0 ejecuta su sección crítica;
    a. Cuando P0 termine de ejecutar su sección crítica, lo inmediato siguiente que hace P0 es ejecutar su código de salida y asigna b[0] = false. Si P1 está activo en ese momento, entonces continúa ejecutándose y no sufre hambruna. En caso contrario, los estados de los registros local y turn funcionan como condiciones iniciales para el caso 2.a de la demostración anterior; donde P1 vuelve a tener la oportunidad de ejecutar su sección crítica cuando se reanude.

    b. Este caso se reduce al caso anterior, debido a que turn[1] = local[1] en todo caso cuando P1 espera por acceder a su sección crítica y de la misma forma, si P1 no está activo cuando P0 le cede el turno a través de b[0], en todo caso P0 va a asignar el valor de turn[1] en turn[0]; por lo que se satisfacerá que local[1] = turn[0] y P1 no sufre hambruna.

2. P1 ejecuta su sección crítica.
    a. Cuando P1 termine de ejecutar su sección crítica, lo inmediato siguiente que hace P1 es ejecutar su código de salida y asigna b[1] = false. Si P0 no está activo en ese momento, entonces continúa ejecutándose y no sufre hambruna. En caso de que P0 no esté activo, P1 puede volver a ejecutar su código de entrada y el estado de los registros funcionan como condiciones iniciales para el caso 1.a de la demostración anterior, donde P0 vuelve a tener la oportunidad de ejecutar su sección crítica cuando se reanude.

    b. También de forma análoga al caso previo, este se reduce al 2.a de esta demostración; debido a que en todo caso, cuando P0 espera por acceder a su sección crítica; turn[0] = local[0] y espera por qué local[0] =/= turn[1]. Y justo ocurre que P1 invierte el valor de turn[0] y eventualmente lo asigna en turn[1], por lo que local[1] = turn[1] =/= turn[0] = local[0]; dando oportunidad nuevamente a P0 de ejecutar su sección crítica y no sufre hambruna.

Por tanto, no existe ningún caso en el que alguno de los dos procesos se tenga que quedar esperando y sin poder continuar su ejecución y por tanto el algoritmo de Kessels es libre de hambruna.

Referencias

1. Taubenfeld, G., (2006), *Synchronization algorithms and concurrent programming*, Pearson Prentice Hall, Inglaterra.