.

Algoritmo de Kessels

Consideremos el algoritmo de ***Peterson***,y las modificaciones que propuso J. L. W. Kessels en 1982:

- Alterando el uso y concepto 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**.
- 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]yturn[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

Condiciones iniciales: b[0] = b[1] = false y los valores iniciales de turn[0] y turn[1] no importan:

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

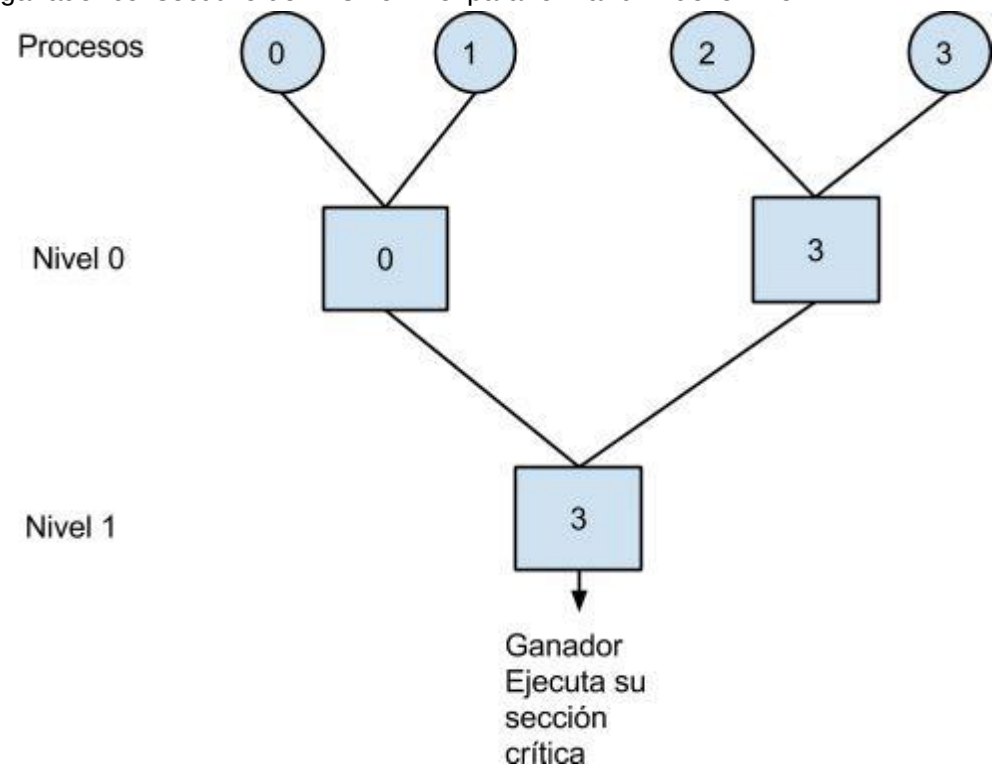
- Observe que podemos reescribir la instrucción 2 del proceso P1 como local[1] ← ¬turn[0], si pensamos en los registros turn cómo booleanos.

- También podríamos reescbirir 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]
- Puesto que es libre de hambruna, entonces también es libre de abrazos mortales.
- Si uno de los proceso permanece en una ejecución antes del código de entrada, entonces su b permanece falsa, lo que permite al primer proceso entrar en su sección crítica.
- Si por el contrario, uno de los procesos está justo en su código de entrada, cambia el valor de su registro turn y permite continuar al otro proceso (no lo bloquea de ninguna manera).
- El valor de fragmentar el indicador turn en dos, es considerar que la escritura y lectura no necesariamente se realiza como una instrucción atómica, y esto permite una revalidación del apropiamiento de la sección crítica.

Vamos a ejemplificar este algoritmo y cómo garantiza la exclusión mutua con el siguiente ejemplo:

Algoritmo de torneo

Vamos a construir un algoritmo para n procesos a partir de cualquier otro algoritmo de exclusión mutua para sólo dos procesos. En particular, vamos a usar ***el algoritmo de Kessels***.La solución general para n procesos, se construye utilizando una estrategia de ***divide y vencerás*** .La estrategia consiste en dividir en grupos de 2 sucesivamente. En cada grupo (binario), los procesos compiten entre ellos dos para entrar a su sección crítica. El ganador, compite contra el ganador consecutivo del mismo nivel para formar un nuevo nivel.



Para implementar esta estrategia usando Kessels, vamos a usar dicho algoritmo en cada nodo del árbol descrito anteriormente. Por simplicidad, vamos a asumir que el número de procesos es potencia de 2. Los procesos se indexan del 0 al n -1 (haciendo un total de n; que es potencia de 2).

Vamos a enumerar cada nodo del árbol con:

- La raíz es 1.
- El hijo izquierdo es 2v (v es el número que etiqueta al padre).
- El hijo derecho es 2v +1.
- Garantizamos que cada nodo se identifica de manera unívoca.
- En este árbol balanceado, la hoja identificada como n +i es el proceso i.
- Para simplificar la representación de ganadores entre procesos competidores por acceder a su sección crítica, para cada nodo interno v ∈ {1,...,n -1}, es indicado por "dónde va bajando el proceso ganador" un arreglo ***edge***[v] de enteros en el intervalo [0, 1] (podemos pensar que es booleano) el proceso correspondiente. Así, en edge[v] se indica el subárbol del que el ganador llega a la posición v (izquierdo o derecho, 0 ó 1).
- Vamos a considerar con cada nodo v los siguientes registros también:
    - b[v,0]
    - b[v,1]
    - turn[v,0]
    - turn[v,1]

  Estos registros representan las variables que usan los procesos para competir en el algoritmo de Kessels.
- Cada proceso tendrá los siguientes registros locales:
    - node - Identifica el nodo del árbol en el que el proceso está compitiendo actualmente.
    - id - Identifica de qué nodo en alguno de los subárboles llegó el proceso. Si id tiene un identificador que corresponde al subárbol izquierdo (id es de la forma 2v; ie. es par), el proceso toma el papel del proceso P0 en el algoritmo de Kessels. Por el contrario, si id es de la forma 2v +1 (es impar), el proceso provino del subárbol derecho y toma el papel de P1 en Kessels. (Esto va a aplicar de la misma manera con las entradas de ***edge***[v], que también son locales para cada proceso, ie cada proceso tiene su arreglo ***edge***[v] que indica por dónde "ha subido" cada vez que gana).
    - local - Es la variable local presente en cada proceso en Kessels.

Inicialmente: todos los registros b son falsos, mientras que los valores de los registros turn no importan.

Generalización del algoritmo de Kessels para n procesos, bajo el modelo de torneo.

1. node ← i +n
2. while(node > 1) do
3. id ← node mod 2

```
4.      node ← node /2
5.      b[node, id] ← true
6.      local ← (turn[mode ,1 -id] +id) mod 2;
7.      turn[node,id] ← local
8.      await(b[node, 1 -id] = false | local =/= (turn[node, 1 -id] +id) mod 2)
9.      edge[node] ← id
10. end
11. {sección crítica}
12. node ← 1
13. while node < n do
14.     b[node, edge[node]] ← false
15.     node ← 2node +edge[node]
16. end
```

Analicemos brevemente que hace el algoritmo:

1. Sabemos que el nodo v es dado por i +n, la instrucción 1 hace uso de esto.
2. El ciclo de la instrucción 2, se encarga de realizar la competencia contra todos los demás procesos con los que a este le toque competir (podemos ver esto como que el proceso sube por un camino del árbol hacia la raíz).
3. Para cada nodo con el que compite, el proceso se asigna el id del nodo del árbol por el que va subiendo. Note que su valor oscila entre 0 y 1, por lo que id explícitamente sólo indica si el nodo sube desde el subárbol izquierdo o el derecho.
4. Actualiza el nodo en el que el proceso se encuentra (sube por el árbol, por cada otro nodo con el que compite).
5. La instrucción 5 corresponde a la 1 del algoritmo original de Kessels.
6. La instrucción 6 abstrae ambas instrucciones 2 del algoritmo original de Kessels. En el original, esta instrucción varía “por una negación booleana” (P0 asigna en local turn[1] a secas, mientras que P1 lo asigna negado).
7. La siguiente también corresponde a la siguiente del original algoritmo de Kessels.
8. La espera ocupada que aparece en Kessels, también abstraída como ocurre en la instrucción 6 de esta versión del algoritmo generalizado. Notemos que en el original, el proceso P0 busca una igualdad en la espera, mientras que P1 espera una desigualdad.
9. Marcamos por qué nodo hemos subido.
10. Termina el ciclo.
11. Ejecuta su Sección Crítica.
12. Se marca como la raíz.
13. En este ciclo, el proceso baja por el árbol. Se prepara para volver a competir.
14. Marca como desocupado (no compitiendo) para cada nodo por el que subió.
15. Vuelve a su respectivo padre.
16. Fin.

Referencias

Gadi Taubenfeld. Synchronization algorithms and concurrent programming,Ed. Pearson Prentice Hall, Israel, pp 35 - 37, 40, 41