Modelo Asincrónico Adaptativo de Exclusión para Grupos de Procesos

Karina M. Cenci * Jorge R. Ardenghi **

Laboratorio de Investigación en Sistemas Distribuidos
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur

Resumen

Las aplicaciones distribuidas requieren mecanismos que garanticen el uso de los recursos compartidos. Para esto, se utilizan modelos de exclusión mutua. Estos modelos se los puede clasificar según su comportamiento en rápidos, adaptivos, basados en el tiempo y no atómicos. En este trabajo, se propone un algoritmo de exclusión mutua para grupos de procesos con la característica de *adaptivo*, teniendo en el peor caso para cada proceso que trabaja independientemente $(4 + n) + 6 + 8(log(n) - 1) + 1$ accesos a memoria y si hay l procesos trabajando concurremente requieren en total $(4+n)l+6+8(log(n)-1)+1$ accesos a memoria.

Palabras Claves: Sistemas Distribuidos - Exclusión Mutua - Exclusión Mutua de Grupo - Concurrencia

*e-mail: kmc@cs.uns.edu.ar
**e-mail: jra@cs.uns.edu.ar

1.- Introducción

Los sistemas distribuidos requieren concurrencia y a su vez exclusión mutua en el uso de los recursos. En algunos casos se resuelve el problema a través de la utilización de un protocolo que garantice exclusión mutua a un único proceso del sistema, este problema corresponde al problema tradicional, en el cuál un sólo proceso accede a la sección crítica. En el caso de los sistemas distribuidos, hay algunas aplicaciones tales como las que soportan trabajo cooperativo, es necesario imponer exclusión mutua sobre un conjunto de procesos, mientras que los procesos del mismo grupo comparten.

En la literatura existen una gran variedad de soluciones para este problema [1], [6], [7], [8], [9]. La idea de este trabajo es alcanzar un protocolo, utilizando el modelo de memoria compartida, que garantice las condiciones de exclusión mutua y concurrencia, que sea adaptivo a los grupos de procesos que están compitiendo y exista una función que limite la cantidad de accesos a memoria en el peor de los casos.

2.- Preliminares

Los algoritmos de exclusión mutua que utilizan el modelo de memoria compartida, se los puede clasificar de acuerdo a las propiedades que presentan como: algoritmos de exclusión mutua *rápidos*, algoritmos *adaptivos*, algoritmos de exclusión mutua basados en el tiempo, algoritmos *no atómicos*.

En los algoritmos de exclusión mutua *rápidos*[7], existe una gran diferencia en la complejidad de tiempo entre casos que están libres de contención y en los que presentan contención. Esto significa que la sección crítica está disponible, que el recurso no está siendo utilizado, por lo tanto la complejidad de tiempo será constante, y en los otros casos la complejidad podrá o no tener puntos de contención.

En los algoritmos de exclusión mutua *adaptivo*, el incremento en la complejidad de tiempo en la contención es más gradual, está en función del número de procesos en contención. En un algoritmo adaptivo la complejidad del paso de una operación depende solamente del número de procesos que realmente ejecutan el paso concurrentemente con esa operación, esto es, la complejidad de una operación está en función de la contención real que encuentra y no del número total de procesos. Las propiedades consideradas para la implementación de un algoritmo adaptivo están relacionadas con el nivel de contención y de adaptabilidad.

Estas características también se las puede aplicar a las extensiones del problema de exclusión mutua, para grupos de procesos, para k procesos.

Las consideraciones para las medidas en la complejidad de tiempo pueden ser varias:

- Complejidad en un paso remoto (referencias de memoria remota) de un algoritmo es el número máximo de operaciones de memoria compartida requeridas por un proceso para ingresar y salir de su sección crítica, asumiendo que cada sentencia *await* es contabilizada como un única operación.

- El tiempo de respuesta del sistema es el intervalo de tiempo entre entradas a la sección crítica.

Otro factor importante para determinar la velocidad de un algoritmo es la cantidad de tráfico de interconexión que él genera. En función de este otro parámetro se define a la *complejidad de tiempo* de un algoritmo de exclusión mutua a ser el peor caso en el número de referencias de memoria remotas por un proceso en orden para ingresar y salir de su sección crítica.

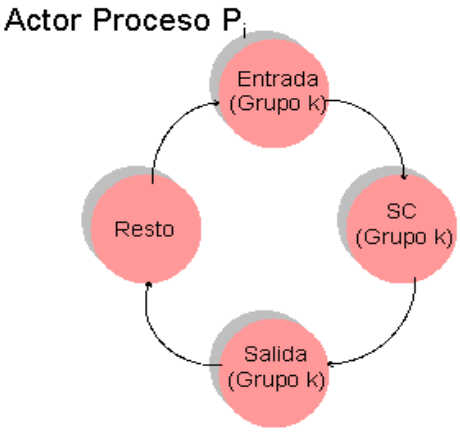
3.- Modelo Base del Algoritmo

El problema presenta dos tipos de actores: los *procesos* y los *grupos*, que se integran para competir por la utilización de un recurso. El modelo presentado en [5], tiene dos componentes que forman parte del mismo, el proceso que selecciona un grupo de trabajo, y el grupo que compite para acceder a la sección crítica.

Cuando un actor no está involucrado de ninguna manera con el recurso, se dice que está en la sección *resto*. Para obtener la admisión a la sección crítica, el actor ejecuta un protocolo de entrada (*trying*), después que utiliza el recurso, se ejecuta un protocolo de salida (*exit*). Este procedimiento puede repetirse, de modo que cada actor sigue un ciclo, desplazándose desde la *sección resto (R)*, a la *sección de entrada (T)*, luego a la *sección crítica (C)* y por último a la *sección de salida (E)*, y luego vuelve a comenzar el ciclo en la *sección resto*.

Como se observa en la figura 1, el primer paso que realiza el *actor proceso*, en la sección de entrada, es seleccionar el grupo en el cual desea participar de un conjunto de m grupos. El segundo paso es esperar hasta que el grupo seleccionado entre en la sección crítica para que pueda acceder a la misma. Cuando finaliza su actividad, sale de la sección crítica, se desvincula del grupo (sección de salida).

El *actor grupo* inicialmente está inactivo, en la sección resto, esto representa que ningún proceso lo ha seleccionado para participar en el mismo. El primer proceso que lo selecciona para participar, hace que comience la competencia por entrar en la sección crítica, y se lo identifica como el primer proceso que pertenece al grupo; pasa a la sección de entrada. Todos los procesos que lo seleccionen mientras se encuentra en competición por entrar a la sección crítica, se agregan a los procesos ya existentes. En el caso que el grupo esté en la sección crítica, si el proceso que activó al grupo está trabajando en la misma, entonces el proceso se



Proceso$_i$

1. ..... {Sección Resto}
2. El proceso selecciona el grupo de trabajo {Grupo$_k$}
3. Espera hasta que entra a la sección crítica {Sección de Entrada}
4. ..... { Sección Crítica}
5. Sale de la sección crítica y se desvincula del grupo.
6. ..... {Sección Resto}

Figura 1: *Esquema 1*

```
Grupo_k
1.- ..... {Sección Resto}
2.- Un proceso lo selecciona para trabajar en él.
{Sección de Entrada}
3.- Si es el primer proceso en el grupo entonces
     Comienza a competir en el ingreso a la S.C.
4.- sino
     Si no está en la S.C. entonces
        Agrega el proceso a la lista de procesos
     sino
        Si está el primero del Grupo entonces
           entra en la S.C. el proceso
        sino
           El proceso lo coloca en la lista de espera
           hasta que termine la sección crítica
           actual y compita por ingresar nuevamente
5.- ..... {Sección Resto}
```

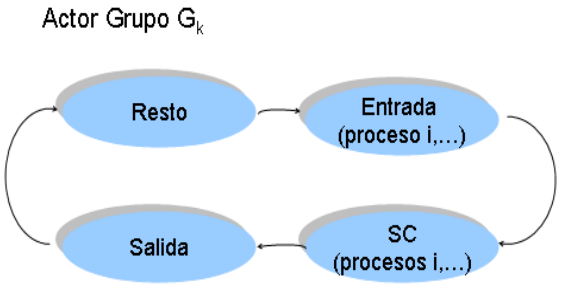


Figura 2: *Esquema 2*

incorpora, sino se pone en cola de espera hasta que termine la actual vuelta (todos los procesos que están trabajando finalicen su tarea y el grupo salga de la sección crítica), se reinicie el ciclo, esto es, compita nuevamente por el ingreso en la sección crítica. En la figura 2 se observa el comportamiento del actor grupo mientras se encuentra activo.

Los *actores grupos* compiten por alcanzar el permiso para acceder al recurso (acceso a la región crítica), y sólo un único grupo tiene derecho de utilizar el recurso en un determinado instante de tiempo.

El esquema base para la competencia de los grupos, está basado en el algoritmo de Tournament [9], con la extensión a m elementos, donde m no es necesariamente una potencia de 2 [4].

Las variables compartidas utilizadas son:

```
∀ flag(i): 0 ≤ i ≤ (m-1), flag(i) = 0 inicialmente
para cada cadena binaria x de a lo sumo longitud etapas-1
    turn(x) inicialmente arbitraria, escrito y leído por exactamente aquellos grupos i
    para los cuales x es un prefijo de la representación binaria de i.
etapas = (Si truncar(log(m)) = log(m) entonces etapas = truncar(log(m)) sino = truncar(log(m)) + 1)
∀ lista(i,j): 0 ≤ i ≤ (m-1), 0 ≤ j ≤ (n-1),
lista(i,j) = <0, resto> inicialmente
```

La variable *flag* indica si el grupo está compitiendo, esto es cuando flag(i) $\neq$ 0, y además en qué nivel se encuentra.

```
Grupo_i
Entrada_i
waitfor [∃ j: 1..n, lista[i,j] = < .., espera>]
Bucar_lider(lista,i)
para k = 1 hasta etapas hacer
    flag(i) = k
    turn(comp(i,k)) = role(i,k)
    Si (role(i,k)≠0) ó (i≤m-k) ó
       (i < 2^etapas/2) entonces
          Waitfor [∀ j ∈ oponentes(i,k) : flag(j) < k] ó
                  [turn(comp(i,k))≠role(i,k)] (1)
flag(i)= etapas + 1
... Sección Crítica

Waitfor [∀ j: 1..n, lista(i,j)≠<..,en_cs>]

flag(i) = 0
```

Cada grupo está ocupado en una serie de competencias de $O(log(m))$ para obtener el recurso. Se considera que la competición está dispuesta en un árbol de competencia binario. Las hojas corresponden a los m grupos. Las definiciones de las funciones que se utilizan corresponden con las definiciones dadas en [4]. La característica que presenta es que en algunos niveles un actor grupo puede que no tenga oponentes en el mismo entonces el actor grupo avanza directamente al próximo nivel, esto puede ocurrir en el caso que m no sea una potencia de 2 entonces el árbol de competición no estaría completo.

El esquema base para cada uno de los actores *procesos* se asemeja al comportamiento que tiene un proceso que quiere acceder a la utilización de recurso o sección de código. A continuación se muestran el conjunto de pasos que realiza.

```
Proceso_l
... Sección Resto
Entrada_l
Selección del grupo en g
Si inactivo(g) entonces
    lista(g,l) = <2, espera>
sino
    lista(g,l) = <1, espera>
fin si
Waitfor (flag(g) = etapas + 1) ∧
        ((lista(g,l)=<2, espera>) ∨
        (∃ j: 1..n, lista(g,j)=<2, en_cs>))
lista(g,l)=<.., en_cs>
... Sección Crítica
Salida_i
lista(g,l) = <0, resto>
```

En el algoritmo se utiliza la variable *lista*, donde el primer índice especifica el grupo y el segundo índice el proceso. En la región de entrada, se selecciona el grupo en el cual va a participar. Luego verifica si el grupo está inactivo. Si la respuesta es afirmativa, es el primero que ingresa en el mismo, inicializa el nivel de la variable *lista(k,i)* en 2, de lo contrario lo hace en 1. Cuando se cumplen las condiciones indicadas en (1) significa que el grupo k está en la región crítica y que el proceso i pueda acceder a ella.

La idea es que el algoritmo cumpla las siguientes condiciones:

- Un único grupo está activo utilizando el recurso compartido (exclusión mutua)
- Si el recurso está disponible y un grupo quiere utilizarlo (está en espera) que acceda al mismo sin tener más demora.
- Si un grupo está activo y el primer proceso también, todo proceso que quiera trabajar en el mismo que lo pueda realizar, de esa manera se logra un mayor nivel de concurrencia.

Como se observa en la figura 3, el grupo g se encuentra en la región crítica, el proceso i es el primer proceso del grupo, luego le sigue el proceso k que se agrega al grupo, entran a la región crítica en forma concurrente, cuando el

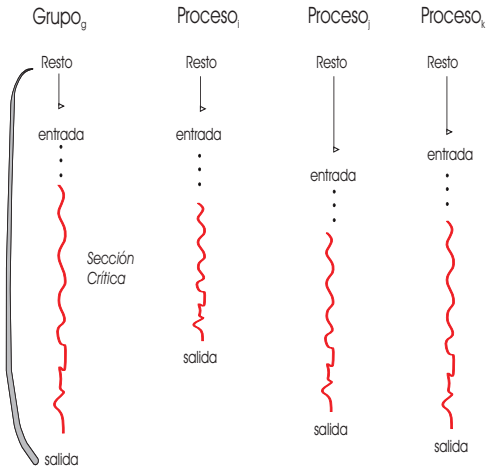


Figura 3: Concurrencia en el grupo g

proceso j selecciona el grupo g, como este ya se encuentra en la región crítica y el primer proceso (proceso i) sigue trabajando en la misma entonces puede acceder también, sin tener que esperar y trabajar concurrentemente.

4.- Consideraciones

Los algoritmos de exclusión mutua deben garantizar las condiciones para un buen algoritmo, y en este caso particular con la extensión para grupos de procesos es conveniente también que maximice la concurrencia de los mismos. Éstos requerimientos son condiciones necesarias para seleccionar un algoritmo, pero no son las únicas que se tienen en cuenta al momento de la implementación. Es importante poder definir funciones que limiten la cantidad de pasos que realizan y en este caso la cantidad de accesos a la memoria compartida, ya que estos consumen recursos y decrementan la performance del mismo. Considerando los diferentes tipos de algoritmos de exclusión mutua que quieren: minimizar la cantidad de accesos en el caso que no exista contención o que se consideren sólo los actores que realmente están en competición.

Analizando el algoritmo presentado en la

sección anterior se llegan a las siguientes consideraciones:

- Los pasos requeridos por cada actor para ingresar en la sección crítica está limitado por la cantidad de niveles. La espera limitada está garantizada porque cumple las condiciones de un buen algoritmo.
- Algunos de los pasos corresponden a una espera ocupada sobre variables compartidas. Si se quiere obtener una función límite de orden n, sobre la cantidad de accesos a memoria compartida (remota) no es posible.
- Para acceder el actor siempre compite contra todos los otros actores aunque estos no se encuentren activos.
- El algoritmo se puede implementar para que chequee primero a los oponentes y luego a la variable compartida *turn*, o modificar el orden de control obteniendo en algunos casos menor cantidad de accesos a la memoria compartida.

Se considera que un acceso a memoria compartida es equivalente a un mensaje.

5.- Algoritmo Adaptivo de Grupos

Una de las desventajas presentadas en la sección anterior es que el algoritmo propuesto controla todos los oponentes que tiene un *actor grupo* en cada uno de los niveles, una mejora es que chequee sólo los actores que están compitiendo en la sección de entrada. Teniendo en cuenta que la construcción del algoritmo se basa en un árbol binario, en donde los actores grupo se encuentran en las hojas y los ganadores de cada uno de los niveles en la raíz de cada subárbol, se puede considerar que en cada nivel de competencia cada uno de los actores

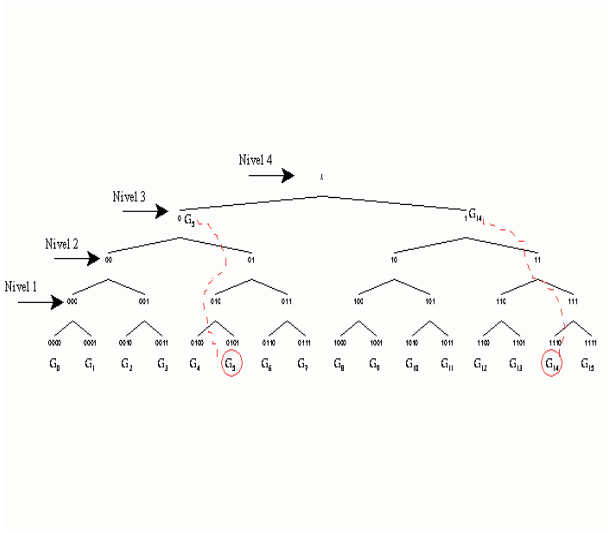


Figura 4: Ejemplo de Competición

tiene un sólo oponente activo. En la figura 4 se muestra un ejemplo teniendo 16 actores y en el nivel 4 están compitiendo los actores grupo g_5 y g_{14}. El grupo g_5 está en el nivel 4 de compentencia, en el caso inicial se chequearían todos los oponentes y estos serían $\{g_8, g_9, g_{10}, g_{11}, g_{12}, g_{13}, g_{14}, g_{15}\}$, que en realidad todos no pueden ser oponentes reales en ese nivel, sino solamente el ganador del nivel 3 que en este caso es el g_{14}.

Para resolver el problema de adaptabilidad se debe conocer el ganador de cada uno de los niveles, para esto es necesario contar con una estructura que mantenga esta información. Se incorpora una nueva variable compartida denominada *ganad*. Esta variable es escrita por aquellos grupos i para los cuales x es un prefijo de la representación binaria de i y leída por todos los grupos, ya que se utiliza para conocer los ganadores de cada uno de los niveles.

En la sección de entrada ($Entrada_i$) se debe modificar para que se controle el oponente real, esto es, el ganador del nivel anterior y además se debe considerar que el nivel 1 es un caso especial, ya que tiene siempre un único oponente y es el mismo en todos los casos. Las modifi-

Entrada$_i$
waitfor [∃ j: 1..n, lista[k,j] = < .., espera>]
Bucar_lider(lista,i)
para k = 1 hasta etapas hacer

flag(i) = k {representa los diferentes niveles}

turn(comp(i,k)) = role(i,k)

Si (role(i,k)≠0) ó (i≤m-k) ó (i < 2^{etapas}/2)) entonces

Si (k = 1) entonces

Waitfor [∀ j ∈ oponentes(i,k) : flag(j) < k]
ó [turn(comp(i,k))≠role(i,k)]

sino

Waitfor [ChequearGanad(i,k)]
ó [turn(comp(i,k))≠role(i,k)]

ganad(comp(i,k)) = i;

flag(i)= etapas + 1 {para que el proceso sepa que está en la S.C.}

... *Sección Crítica*

Salida$_i$

Waitfor [∀ j: 1..n, lista(i,j)≠<..,en_cs>]
flag(i) = 0
para k = 1 hasta etapas hacer

ganad(comp(i,k)) = -1

ChequearGanad(i,k) = Si (ganad(comp(i,k)) ≠ -1) ó (ganad(comp(i,k)$\overline{role(i,k)}$) ≠ -1)
Entonces Falso
Sino Verdadero

Esquema 5

caciones realizadas se observan en el Esquema 5, aquí se realiza la llamada a una función denominada *ChequearGanad* la cual controlará si está activo alguno de los oponentes correspondientes a ese nivel y si ya ha ganado en el nivel. La función $\overline{role(i,k)}$ devuelve como resultado el complemento con respecto a la función *role*, esto es si *role* devuelve 1 entonces $\overline{role}$ devuelve 0 y viceversa, el resultado de esta función es concatenado con el resultado de aplicar *comp(k,i)* que permite obtener el ganador oponente correspondiente.

La sección de salida también se la debe adaptar para mantener los nuevos requerimientos, se libera para cada uno de los niveles el ganador del mismo para que los competidores puedan avanzar. Este algoritmo cumple con las condiciones de un buen algoritmo de exclusión mutua para grupos de procesos y además es adaptivo, esto es, sólo intervienen en la sección de entrada los grupos que están en competición. Si se quiere obtener una función límite de orden n, sobre la cantidad de accesos a memoria compartida (remota) no es posible, ya que está diseñado a través de una espera ocupada sobre las variables compartidas *ganad*, *flag* y *turn*.

Para eliminar las esperas ocupadas sobre variables compartidas (esto significa acceder a las mismas en forma remota) se puede testear inicialmente el estado de estas variables y luego esperar hasta que las mismas hayan cambiado su estado avisándole al actor. Considerando que el mayor costo es de comunicación, se puede incorporar una espera ocupada en forma local (local spin).

Se requiere incorporar variables para que cada uno de los actores grupo accedan localmente mientras esperan que cambie el estado respectivo permitiendo que avance en el nivel de competencia. Para eliminar la espera ocupada sobre las variables compartidadas *ganad*, *flag* se agrega una variable denominada *oponente_nivel*.

∀ oponente_nivel(i): 0 ≤ i ≤ (m-1), inicialmente oponente_nivel (i) = 0, oponente_nivel (i) ∈ {0, 1, 2}, leída por i y escrita por todos.

Para cada uno de los *actores grupo* el valor de *oponente_nivel* está inicializado en 0, esto representa que es la primera vez que va a chequear la variable compartida *ganad* o *flag*, si en el primer control encuentra que debe esperar asigna el valor 1 a *oponente_nivel* y en los siguientes controles espera hasta que cambie este valor, el mismo es modificado por el *actor*

grupo oponente ganador cuando se encuentra en la sección de salida.

Para eliminar la espera ocupada sobre la variable *turn* se agrega una variable denominada *turno*.

> ∀ turno(i): 0 ≤ i ≤ (m-1), inicialmente vacío, turno (i) ∈ {0, 1}, leída por i y escrita por todos.

En cada iteración de la espera se controla por el estado de la variable *turno(i)* que se encuentra localmente, las modificaciones se deben realizar en la sección de entrada para que actualice el valor de la variable *turno(j)* correspondiente con el competidor del nivel.

En el algoritmo se incorporan las siguientes funciones:

- ActualizarTurno(grupo,k) esta función tiene como parámetros el grupo y el nivel correspondiente. Controla si existe un oponente activo, si es así entonces actualiza el valor de la variable *turno* local con el valor de la variable *turn(comp(grupo,k))*.
- ChequearFlag(grupo,k) esta función tiene como parámetros el grupo y el nivel correspondiente. La primera vez que es invocada en el primer nivel, controla el estado de la variable *flag* del oponente, si está activo entonces en las próximas iteraciones controlará el estado de la variable *oponente_nivel(grupo)* que se encuentra localmente. Si no es el primer nivel entonces la primera vez que es invocada controla el estado de la variable *ganad(comp(grupo,k)* para poder determinar si ya hay un grupo ganador en el nivel, o si hay un oponente activo en el mismo; si está activo el oponente en las próximas iteraciones controlará el estado de la variable *oponente_nivel(grupo)*

En la figura 5 se muestran todas las variables compartidas utilizadas en el algoritmo y

---

∀ flag(i): 0 ≤ i ≤ (m-1), flag(i) = 0 inicialmente
para cada cadena binaria x de a lo sumo longitud etapas-1

turn(x) inicialmente arbitraria, escrito y leído por exactamente aquellos grupos i para los cuales x es un prefijo de la representación binaria de i.

ganad(x), ganad(x) = -1 inicialmente

∀ turno(i): 0 ≤ i ≤ (m-1), inicialmente vacío, turno(i) ∈ {0,1}, leído por i y escrita por todos
∀ oponente_nivel(i): 0 ≤ i ≤ (m-1), inicialmente oponente_nivel(i) = 0, oponente_nivel(i) ∈ {0,1,2}, leído por i y escrita por todos
∀ lista(i,j): 0 ≤ i ≤ (m-1), 0 ≤ j ≤ (n-1), lista(i,j) = <0, resto> inicialmente
∀ espera(j): 0 ≤ j ≤ (n-1), espera(,j) = resto inicialmente
etapas = (Si truncar(log(m)) = log(m)
entonces
= truncar(log(m))
sino
= truncar(log(m)) + 1 fin si)

Figura 5: Variables Compartidas

en el Esquema 6 se muestra el algoritmo en un formato tradicional.

Para considerar que es un buen algoritmo debe estar bien formado, garantizar las condiciones de exclusión mutua, progreso y que no espere indefinidamente para ingresar en la sección crítica.

¿El algoritmo presentado garantiza la exclusión para los actores grupo? Si consideramos que tenemos $m = 16$, cantidad de *actores grupo*. Supongamos que los grupos G_4 y G_{12} ingresan simultáneamente a la sección crítica. Se tendría un total de 4 niveles de competencia. Como cada uno de los grupos pertenece a una rama diferente de árbol construido, ya que la representación binaria para G_4 es 0100 y para el G_{12} es 1100. Estos grupos van a ser competidores en el último nivel, esto es en el nivel 4. Ambos actores grupo tendrían valor 4 en sus correspondientes *flag*, considerando que primero actualiza la variable *turn* el grupo G_4 se tendría la siguiente situación:

1. (G_4) turn(comp(4,4)) = role(4,4) = 0
2. (G_4) turno(4) = turn(comp(4,4)) = 0

Entrada$_i$
waitfor [∃ j: 1..n, lista[k,j] = < .., espera>]
Bucar_lider(lista,i)
para k = 1 hasta etapas hacer
    oponente_nivel(i) = 0
    flag(i) = k
    turn(comp(i,k)) = role(i,k)
    turno(i) = turn(comp(i,k))
    ActualizarTurno(i,k)
    Si (role(i,k)≠0) ó (i≤m-k) ó (i < (2^{etapas}/2)) entonces
        Waitfor [ChequearFlag(i,k)] ó [turno(i)≠role(i,k)]
    ganad(comp(i,k)) = i
flag(i)= etapas + 1
para j = 0 hasta n-1 hacer
    Si lista(i,j) = <.., espera> entonces espera(j) = en_cs

... *Sección Crítica*

Salida$_i$

Waitfor [∀ j: 1..n, lista(i,j)≠<..,en_cs>]
flag(i) = 0
para k = etapas hasta 1 hacer
    ganad(comp(i,k)) = -1
    Si ((j=OponenteActivo(i,k)) ≠ -1) entonces
        oponente_nivel(j) = 2

ChequearFlag(i,k) = Si (oponente_nivel(i) = 0)
Entonces (Si (k=1) Entonces
(Si (para el grupo oponente flag(j) ≥ k)
Entonces oponente_nivel(i) = 1, Falso Sino Verdadero)
Sino (Si (ganad(comp(i,k)) ≠ -1) ó
        (ganad(comp(i,k)$\overline{role(i,k)}$) ≠ -1)
      Entonces Falso Sino Verdadero))
Sino (Si (oponente_nivel(i) =1)
    Entonces Falso Sino Verdadero)

ActualizarTurno(i,k) =
Si ((j= OponenteActivo(i,k)) ≠ -1) Entonces
    Si (flag(j) = k) Entonces turno(j) = turn(comp(i,k))

OponenteActivo(i,k) =
Si (k=1) Entonces j=Oponentes(i,k)
Sino (Si (ganad(comp(i,k)) ≠ -1) ó
        (ganad(comp(i,k)$\overline{role(i,k)}$) ≠ -1)
      Entonces j = ganad(comp(i,k)$\overline{role(i,k)}$)
      Sino j = -1)

Esquema 6

3. (G_{12}) turn(comp(12,4))= role(12,4) = 1
4. (G_4) ActualizarTurno(4,4), como está activo el oponente correspondiente al nivel, turno(12)=turn(comp(4,4))=1
5. (G_{12}) turno(12)= turn(comp(12,4)) = 1
6. (G_{12}) ActualizarTurno(12,4), como está activo el oponente correspondiente al nivel, turno(4)=turn(comp(12,4))=1
7. (G_4) Waitfor [ChequearFlag(4,4)] ó [turno(4) ≠ role(4,4)]
8. (G_{12}) Waitfor [ChequearFlag(12,4)] ó [turno(12) ≠ role(12,4)]

Cuando el grupo G_4 se encuentre en el Waitfor (paso 7) el control sobre el *ChequearFlag* será falso ya que el oponente también está en el mismo nivel, lo mismo le ocurrirá al grupo G_{12} en el Waitfor (paso 8). Para que ambos ingresen entonces tendrá que ser verdadera la otra condición, esto es, el valor de la variable *turno* deberá ser diferente al *role*, pero esta variable está relacionada con la variable *turn* que en un instante de tiempo tiene el valor 0 ó 1, por lo tanto uno de los 2 grupos tendrá falso en la condición. En este caso ingresaría el grupo G_4 a la sección crítica y el otro grupo deberá esperar que el grupo libere la sección crítica. Si el grupo G_{12} hubiera actualizado primero la variable *turn* entonces sería este grupo el que ingresaría a la sección crítica y el otro debería esperar. Por lo tanto se contradice la suposición inicial en la cual los 2 grupos puede ingresar al mismo tiempo a la sección crítica. En general, considerando la situación que se observa en la figura 6, donde se tiene 2 grupos G_i y G_j que están compitiendo en el mismo nivel k entonces supongamos que los 2 grupos pueden ganar al mismo tiempo en el nivel k. Ambos grupos tendrían en la variable *flag* correspondiente el valor k, que es el nivel de competencia. Considerando que primero actualiza la variable *turn* G_j entonces se tiene la siguiente situación:

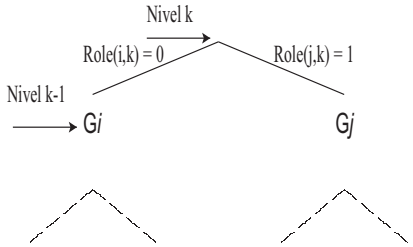


Figura 6: Ejemplo de Competición en el Nivel k

1. (G_j) turn(comp(j,k)) = role(j,k) = 1
2. (G_j) turno(j) = turn(comp(j,k)) = 1
3. (G_i) turn(comp(i,k))= role(i,k) = 0
4. (G_j) ActualizarTurno(j,k), como está activo el oponente correspondiente al nivel, turno(i)=turn(comp(j,k))= 0
5. (G_i) turno(i) = turn(comp(i,k)) = 0
6. (G_i) ActualizarTurno(i,k), como está activo el oponente correspondiente al nivel, turno(j)=turn(comp(i,k))= 0
7. (G_j) Waitfor [ChequearFlag(j,k)] ó [turno(j) $\neq$ role(j,k)]
8. (G_i) Waitfor [ChequearFlag(i,k)] ó [turno(i) $\neq$ role(i,k)]

Cuando el grupo G_j se encuentra en el Waitfor (paso 7) el control sobre el *ChequearFlag* será falso ya que el oponente también está en el mismo nivel, lo mismo le ocurrirá al grupo G_i en el Waitfor (paso 8). Para que ambos sean los ganadores entonces tendrá que ser verdadera la otra condición, esto es, el valor de la variable *turno* deberá ser diferente del *role* en ambos casos, pero esta variable está relacionada con la variable *turn* que en un instante de tiempo tiene el valor 0 ó 1, cuando se actualiza *turn* se invoca a la función *ActualizarTurno* y actualiza el valor del oponente si este está activo, por lo tanto un sólo grupo va a ser el ganador y en este caso sería G_j, lo cual contradice la suposición inicial. Por lo tanto, en cada uno de los niveles del subárbol hay un único ganador y en el último nivel el ganador accede a la sección crítica.

Si en una ejecución a, hay por lo menos un grupo en la sección de entrada y no hay ningún grupo en la sección crítica, supongamos que no entra ningún grupo. Si el grupo, denominado i, está en la sección de entrada entonces está esperando en el estado de *ChequearFlag* o chequear el *turno*, sino hay nadie en la sección crítica y es el único competidor entonces el ChequearFlag verificará que no hay ganador y podrá avanzar de nivel y así hasta el último nivel. Si hay 2 grupos compitiendo en el mismo nivel, por lo mostrado anteriormente uno será el ganador y por lo tanto podrá acceder a la sección crítica. Esto contradice la suposición inicial.

Si un algoritmo está bien formado y es libre de bloqueo (para todos los procesos) entonces garantiza la propiedad de progreso. Si mantiene la propiedad de vivacidad entonces no puede ocurrir inanición. ¿Es posible que un actor proceso/grupo espere indefinidamente para acceder a la sección crítica? Cuando un grupo i ingresa en cada uno de los niveles de la sección de entrada, inicializa la variable *turn* para ese nivel e invoca a la función ActualizarTurno que controla si hay un grupo oponente activo, y actualiza el valor de *turno*, por lo tanto un grupo j que accede después a la sección de entrada, deberá esperar que el grupo i tenga su acceso a la sección crítica o al próximo nivel antes de poder avanzar.

La idea del algoritmo es poder obtener una

función de orden n que limite la cantidad de accesos a memoria compartida. El modelo tiene *etapas* niveles donde $etapas \equiv log(m)$ ó $etapas \equiv \mid log(m) \mid +1$, dependiendo si m es una potencia de 2. ¿Cuántos accesos se requieren para acceder a la sección crítica?, analizando el algoritmo y para responder a la pregunta se divide el problema en dos casos, cuando se encuentra en el primer nivel ($k = 1$) y en el resto de los niveles ($k \neq 1$), considerando al actor *grupo i*

- Para k = 1
  - 1 acceso a variable *turn* (es la que mantiene el turno, para evitar la espera indefinida)
  - 3 accesos en la función ActualizarTurno, se accede a la variable *flag(j)* y a la variable *turno(j)* asignándole el valor de *turn*, en el caso que el oponente se encuentre activo en el nivel; sino sería un solo acceso.
  - 1 acceso en la función ChequearFlag, se accede a la varibale *flag(j)*
  - 1 acceso a la variable *ganad*

  La cantidad de accesos son 6 en el peor de los casos.
- Para k $\neq$ 1
  - 1 acceso a variable *turn* (es la que mantiene el turno, para evitar la espera indefinida)
  - 4 accesos en la función ActualizarTurno, se accede a la variable *ganad* para buscar el oponente, a la varible *flag(j)* y a la variable *turno(j)* asignándole el valor de *turn*, en el caso en que el oponente se encuentre activo en el nivel. Si no hay oponente activo entonces se tendría un sólo acceso que es a la variable *ganad*. Si hay oponente pero se encuentra en un nivel superior entonces se tendrían sólo los 2 primeros accesos a las variables *ganad* y *flag*.
  - 2 accesos en la función ChequearGanad, se accede a la varibale *ganad* para saber si hay un ganador en el nivel o si el oponente se encuentra compitiendo en el nivel.
  - 1 acceso a la variable *ganad*

  La cantidad de accesos son 8 en el peor de los casos para cada una de los niveles. Considerando que se tienen (*etapas -1*) niveles entonces la cantidad de accesos que se requiere para acceder a la sección son $8(etapa - 1)$.

Considerando los 2 casos y que etapas está definido en función de m, entonces se puede determinar que se requiere

$$6 + 8(log(m) - 1) \tag{1}$$

en el peor de los casos para acceder a la sección crítica, considerando como un único paso las esperas ocupadas localmente.

Un actor proceso requiere (4+n)accesos a memoria. En un total un actor proceso que va a participar sólo en un grupo tendrá como función que limite la cantidad de accesos:

$$(4 + n) + 6 + 8(log(m) - 1) + 1 \tag{2}$$

En el caso en que en el grupo participen l procesos concurrentemente entonces se tendría la siguiente función que limite la cantidad de accesos a:

$$(4 + n)l + 6 + 8(log(m) - 1) + l \tag{3}$$

6.- Conclusiones

En arquitecturas distribuidas, existen aplicaciones que conviven en el ambiente que no comparten recursos ni tareas en común, pero también existen aplicaciones que colaboran para resolver un problema, o se dividen en procesos para distribuir el trabajo entre los distintos nodos y obtener un mejor rendimiento (como

por ejemplo, en cálculo numérico para resolver problemas basados en matrices).

Cuando se considera la implementación de un modelo de exclusión y concurrencia se tiene en cuenta: (1) las condiciones necesarias, esto es, exclusión mutua, progreso, concurrencia; (2) el costo del mismo, está relacionado con la cantidad de pasos necesarios y los accesos a memoria realizados. Por esto, es necesario que el modelo seleccionado pueda definirse funciones que estimen el peor de los casos. En este trabajo se realiza una adaptación del modelo presentado en [5] transformando las esperas ocupadas en esperas locales, y para el caso en que l procesos estén concurrentemente en la sección crítica se requiere en el peor de los casos $(4+n)l+6+8(log(m)-1)+l$ accesos. La desventaja que presenta es que se tienen esperas locales, la idea es continuar trabajando en esta línea.

Referencias

[1] Yuh-Jzer Joung *Asynchronous Group Mutual Exclusion (extended abstract).* In Proc. 17 th. ACM PODC, Junio 1998.

[2] Patrick Keane, Mark Moir *A Simple Local-Spin Group Mutual Exclusion Algorithm.*

[3] J. Anderson, Mark Moir *Using local-spin k-exclusion algorithms to improve wait-free object implementations.* Distributed Computing, 11:1-20, 1997. Una versión preliminar apareción en Proceedings of the 13th Annual ACM Symposium on Pronciples of Distributed Computing, 1994, pp. 141-150.

[4] Karina Cenci, Jorge Ardenghi *Exclusión Mutua para Coordinación de Sistemas Distribuidos.* CACIC 2001

[5] Karina Cenci, Jorge Ardenghi *Algoritmo para Coordinar Exclusión Mutua y Concurrencia de Grupos de Procesos* CACIC 2002

[6] K. Vidyasankar *A simple group mutual* l-*exclusion algorithm* Information Processing Letters, 2003.

[7] L. Lamport *A Fast Mutual Exclusion Algorithm* ACM on Transactions on Computer Systems, Volume 5, Number 1, Febrero 1987

[8] Gary L. Peterson. *Myths about the mutual exclusion problem.* Information Processing Letters, Junio 1981.

[9] G. L. Peterson, M. J. Fischer *Economical Solutions for the Critical Section Problem in a Distributed System* ACM Annual proceedings of Theory of Computing, pp.91-97, 1977.