Algoritmos de exclusión mutua

1 Exclusión mutua

Uno de los problemas a resolver cuando se implantan programas concurrentes es garantizar que las *secciones críticas* existentes en diferentes componentes del programa se ejecuten en *exclusión mutua*. Es decir, que únicamente las ejecute un solo proceso o hilo en cada momento.

Existen algunas soluciones clásicas a este problema. Están descritas en las próximas secciones.

1.1 Algoritmo centralizado

En la primera solución se utiliza un proceso coordinador. El algoritmo resultante sigue estos pasos:

1. Cuando un proceso deba acceder a su sección crítica enviará un *mensaje SOLICITAR* al proceso coordinador indicando que pretende acceder a la sección crítica.

2. Si el coordinador observa que la sección crítica está libre actualmente, responderá con un *mensaje CONCEDER*. De esa manera el proceso solicitante reanudará su ejecución y ejecutará su sección crítica.

   Por el contrario, si el coordinador sabe que la sección crítica está ocupada por otro proceso, se anotará la identidad del solicitante y no responderá. Como el proceso solicitante espera esa respuesta, de momento permanecerá suspendido.

3. Cuando un proceso finalice la ejecución de su sección crítica, enviará un *mensaje LIBERAR* al coordinador, reportando tal finalización. El coordinador comprobará si existe algún proceso suspendido esperando permiso para entrar en la sección. Si hubiera alguno, seleccionará alguno de ellos (por ejemplo, el que hubiese realizado la solicitud en primer lugar) y responderá a su solicitud con un mensaje CONCEDER.

   Si no hay ningún solicitante bloqueado, el coordinador se anotará que la sección crítica queda libre. De esta manera, el primer nuevo solicitante podrá acceder sin necesidad de suspenderse.

Este algoritmo centralizado presenta la ventaja de que requiere pocos mensajes para completar la gestión del acceso a una sección crítica. Basta con tres mensajes por cada proceso (SOLICITAR, CONCEDER y LIBERAR).

También se puede justificar de manera sencilla su buen funcionamiento. El coordinador sabe si la sección crítica está libre u ocupada y en función de ello sólo permite que un único proceso la ejecute en cada momento.

Desafortunadamente, también tiene algunos puntos débiles. La comunicación debe ser fiable, pues la pérdida de algún mensaje puede tener efectos graves. Por ejemplo, la pérdida de un mensaje SOLICITAR implicaría que el solicitante correspondiente permaneciera suspendido indefinidamente. A su vez, si se pierde un mensaje CONCEDER tendremos un efecto semejante para el proceso solicitante, pero además el proceso coordinador creería que la sección libre estaría ocupada y no permitiría que ningún otro proceso accediera a ella. Por tanto, esa sección quedaría bloqueada de manera indefinida. Si se perdiera el mensaje LIBERAR también mantendríamos el bloqueo sobre la sección, aunque en ese caso el proceso liberador continuaría su ejecución sin problemas.

Otro problema surge con la caída del proceso coordinador pues se perdería toda la información relacionada con el estado actual de la sección y la identidad de su proceso propietario, en caso de que hubiese alguno y los mensajes que deben esperarse para proseguir con una gestión adecuada. Por tanto, el proceso coordinador constituye un *punto único de fallo* y debería replicarse para garantizar la continuidad de este servicio.

1.2 Algoritmo distribuido

Aunque una primera versión de algoritmo distribuido de exclusión mutua ya se sugirió en [2], posteriormente fue optimizada en [3]. En ambos algoritmos se utilizaban los relojes lógicos de Lamport, complementados con los identificadores de los nodos (tal como se describe en [2]) para lograr un orden total en el etiquetado temporal de los eventos.

El algoritmo utilizado en [3] seguía estos pasos:

1. Cuando un proceso quiera acceder a la sección crítica difundirá un *mensaje TRY* a todos los procesos del sistema.

2. Cuando un proceso reciba un mensaje TRY, actuará de la siguiente forma:

   - Si no está en su sección crítica ni intentaba entrar, responderá con un *mensaje OK*.
   - Si está en su sección crítica, no contesta, pero encola el mensaje.
   - Si no está en su sección crítica, pero quiere entrar, compara el número de evento del mensaje entrante con el que él mismo envió al resto. Vence el número más bajo. Así, si el mensaje entrante es más bajo, el receptor responde OK. Si fuera más alto, no responde y encola el mensaje.

3. Un proceso entra en la sección crítica cuando recibe OK de todos.

   Cuando abandone la sección, enviará OK a todos los procesos que enviaron los mensajes que retuvo en su cola.

Es fácil justificar que el algoritmo funciona correctamente. Si sólo hubiese un proceso interesado en acceder a la sección crítica, todos los demás le responderían con el correspondiente OK y accedería sin problemas. En caso de que haya más de un peticionario, aquellos procesos que no hayan solicitado la entrada responderán a todos los solicitantes. El conflicto entre los solicitantes (un posible empate no resuelto bloquearía a todos ellos) se resuelve gracias a los relojes lógicos utilizados. Como estos están complementados con los identificadores de los procesos emisores, jamás podrá haber un empate. Aquel proceso que haya difundido un TRY con el valor de su reloj lógico más bajo será el único que podrá acceder a la sección.

Por motivos similares a los discutidos en el algoritmo centralizado, es básico que la comunicación sea fiable. En caso contrario podría bloquearse alguno de los procesos solicitantes y, a su vez, evitar que otros solicitantes posteriores accedieran alguna vez a la sección crítica.

1.3 Algoritmo para anillos

En el algoritmo para anillos, todos los procesos participantes se estructuran en un anillo lógico y cada uno de ellos conoce la identidad y dirección del siguiente proceso del anillo.

El algoritmo está basado en la existencia de un token que va circulando por el anillo de la siguiente manera:

1. Un proceso no puede iniciar la ejecución de su sección crítica mientras no reciba el token.

2. Al recibir el token, si se quería acceder a la sección ya se podrá ejecutar ésta. El token se mantendrá mientras no finalice la ejecución de la sección crítica. Cuanto termine, se pasará al siguiente proceso.

   Si el proceso no quería acceder a su sección cuando recibió el token, lo pasará inmediatamente al siguiente proceso en el anillo.

Este algoritmo funciona correctamente mientras no fallen los procesos que participan en él y no se duplique involuntariamente el token. Es obvio que garantizará exclusión mutua, pues solo debería existir un token y únicamente aquel proceso que lo mantenga podrá ejecutar su sección crítica.

Cuando un proceso falle habría que comprobar lo antes posible si tenía el token o no. No basta con esperar algún tiempo para que "aparezca" el token, pues resulta impredecible cuánto le costará a cada proceso ejecutar su correspondiente sección. Por ello, sería conveniente utilizar un algoritmo para obtener una imagen consistente del estado global [1].

References

[1] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[2] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[3] Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24(1):9–17, January 1981.