PROGRAMACION CONCURRENTE

Freddy L. Abad Leon

—————————— ◆ ——————————

**PROBLEMA DEL PRODUCTOR-CONSUMIDOR**

El problema del productor-consumidor es un ejemplo clásico de problema de sincronización de multiprocesos (orden de ejecución). Está presente en casi todos los mecanismos de comunicación, las tuberías entre procesos y comandos, la E/S a dispositivos, comunicaciones por red, etc. Hay dos procesos en el programa: productor y consumidor, los cuales comparten un buffer de tamaño finito. La tarea del productor es generar un producto, almacenarlo y comenzar nuevamente; mientras que el consumidor toma (simultáneamente) productos uno a uno. El problema consiste en que el productor no añada más productos que la capacidad del buffer y que el consumidor no intente tomar un producto si el buffer está vacío. A su vez pueden ser síncronos, el cual quiere decir que el productor se encarga de Producer un elemento si es consumido el anterior producto para asi continuar el proceso. O también pueden ser asíncronos donde el canal de comunicación tiene una capacidad de almacenamiento o buffer limitada, el cual hace más fácil que los productores generen más productos, ponerlos en cola y dejar que los consumidores tomen los productos. El caso de ser asíncrono es mas común, ya que los procesos pueden avanzar a su propio ritmo, pero necesitan de sincronización para hacer determinadas tareas, como cuando el buffer esta vacío, esperar los consumidores, o si el buffer está lleno, esperar los productores. Estos procesos son llamados cíclicos, ya que están en constante producción o consume del buffer.

*PRODUCTOR*



*CONSUMIDOR*



Si hay productores en espera se soluciona haciendo que ambos procesos (productor y consumidor) se ejecutan simultáneamente y se “despiertan” o “duermen” según el estado del buffer. Concretamente, el productor agrega productos mientras quede espacio y en el momento en que se llene el buffer se pone a “dormir”. Cuando el consumidor toma un producto notifica al productor que puede comenzar a trabajar nuevamente. En caso contrario si el buffer se vacía, el consumidor se pone a dormir y en el momento en que el productor agrega un producto crea una señal para despertarlo. Se puede encontrar una solución usando mecanismos de comunicación interprocesos, generalmente se usan semáforos. Una inadecuada implementación del problema puede terminar en un deadlock donde ambos procesos queden en espera de ser despertados. Este problema pude ser generalizado para múltiples consumidores y productores.

**MECANISMO: CERROJO**

Es un mecanismo de sincronización en los sistemas operativos de corto plazo basados en la espera por ocupado (como inhibición de interrupciones y cerrojos de espera activa).

En multiprocesadores es necesario utilizar cerrojos de espera activa. El cual es un mecanismo más general que la inhibición de interrupciones es la utilización de un variable cerrojo para proteger la sección crítica. El proceso que quiere entrar a la sección crítica consulta el cerrojo. Si está libre (cerrojo==0), el proceso arroja (cerrojo=1) y entra a la sección crítica. Si está echado, ejecuta una espera activa consultando su valor hasta que esté libre. Cuando un proceso deja la sección crítica, libera el cerrojo (cerrojo=0). Este esquema sencillo presenta importantes problemas de implementación. Como se puede comprobar, la operación de consulta y modificación del cerrojo constituye a su vez una sección crítica que hay que resolver previamente; si no dos procesos podrían leer simultáneamente un valor cero y ambos entrar a la sección crítica, violando la condición exclusión mutua.

Los procesadores modernos cuentan con instrucciones máquina análogas a Test & Set que permiten implementar la espera activa más eficientemente, reduciendo la contención en el acceso al bus de memoria. Algunos sistemas operativos utilizan cerrojos condicionales, proporcionando una primitiva de echar el cerrojo condicionalmente (cond\_lock ()) que, en vez de dejar al proceso esperando, devuelve un código de estado si el cerrojo está ocupado, lo que es útil para tratar de evitar interbloqueos, como se verá más adelante. La espera activa es un mecanismo adecuado para multiprocesadores. En monoprocesadores una espera activa por entrar a la sección crítica podría impedir, dependiendo de la política de planificación, que el proceso que ocupa la sección crítica acceda al procesador para liberarla, y en el mejor de los casos retardaría su entrada. En cualquier caso, aún en multiprocesadores, es adecuado combinar el mecanismo de espera activa con una planificación que incremente la prioridad del proceso que ejecuta la sección crítica.

Un cerrojo primitivo es una primitiva de sincronización que no pertenece a ningún hilo en particular cuando está bloqueada. En Python & Java, actualmente es las primitiva e sincronización de más bajo nivel disponible, implementada directamente por el módulo de extensión thread. Un cerrojo primitivo está en uno de dos estados, ``bloqueado'' o ``desbloqueado''. Se crea en el estado desbloqueado. Tiene dos métodos básicos, acquire() (adquirir) y release() (liberar). Cuando el estado es desbloqueado, acquire() cambia el estado ha bloqueado y retorna inmediatamente. Cuando el estado es bloqueado, acquire() se queda bloqueado hasta que por una llamada a release() desde otro hilo cambia el estado ha desbloqueado. El hilo que había llamado a acquire() vuelve a ponerlo a bloqueado y retorna de inmediato. Cuando más de un hilo esté bloqueado en acquire() esperando a que se desbloquee el cerrojo, sólo uno de ellos sigue su ejecución cuando una llamada a release() retorna el estado ha desbloqueado. Cuál de ellos prosigue no está definido y depende de la implementación.

Todos los métodos se ejecutan atómicamente.

**acquire** ([blocking *= 1*])

Adquirir un cerrojo, bloqueante o no bloqueante. Si se invoca sin argumentos, bloquear la ejecución hasta que se desbloquee el cerrojo, bloquearlo y retornar. No hay valor de retorno en este caso. Si se invoca con el argumento blocking a cierto, hacer lo mismo que si se llama sin argumentos y devolver cierto. Si se invoca con el argumento blocking a falso, no bloquear la ejecución. Si llamándolo sin argumento se hubiera bloqueado, devolver falso de inmediato. De otro modo, hacer los mismo que al llamar sin argumentos y devolver verdadero.

**release** ()

Libera un bloqueo. Si el cerrojo está bloqueado, devolverlo al estado de desbloqueado y retornar. Si hay otros hilos bloqueados a la espera de que el cerrojo se desbloquee, permitir exactamente a uno que prosiga. No se debe llamar a este método si el cerrojo está desbloqueado. No hay valor de retorno.

**MECANISMO: LECTORES Y ESCRITORES**

El problema de los lectores y escritorio es un problema de sección crítica *("La sección crítica es la parte que debe protegerse de interferencias de otros procesos. También se define como a la porción de código de un programa de computador el cual accede a un recurso compartido (estructura de datos o dispositivo) que no debe de ser accedido por más de un hilo en ejecución (thread). La sección crítica por lo general termina en un tiempo determinado y el hilo, proceso o tarea solo tendrá que esperar un período determinado de tiempo para entrar."*), donde es posible encontrar otros tipos de problemas de control de concurrencia en la sincronización de procesos. Los problemas como sección crítica y los que se mencionan a continuación son usados actualmente para poner a prueba todos los nuevos esquemas de sincronización propuestos. El problema trata de dos perfiles de procesos que interactúan simultáneamente, es decir, imaginando una Base de Datos donde se tiene un proceso que desea leer los datos y otro proceso que desea actualizar los datos, a los cuales le llamaremos lector y escritor. Si los lectores acceden simultáneamente no habrá problema alguno, pero el problema surge cuando quiere interactuar con la base de datos el lector y escritor al mismo tiempo, generando así el problema.

Para asegurar que no ocurra este tipo de problemas, se debe asegurar que los procesos escritores tengan acceso exclusivo a la base de datos.

**MECANISMO: SEMAFOROS**

Un semáforo es un mecanismo de comunicación con el cual no se mueven datos, puesto que solo se puede consultar y modificar su valor al tener un carácter puramente informativo.

Son componentes de muy bajo nivel de abstracción, de fácil comprensión y con una gran capacidad funcional. Un semáforo es un tipo abstracto de dato, y como tal, su definición requiere especificar sus dos atributos básicos:

\* Conjunto de valores que puede tomar.

\* Conjunto de operaciones que admite.

Un semáforo tiene también asociada una lista de procesos, en la que se incluyen todos los procesos que se encuentra suspendidos a la espera de acceder al mismo. Un semáforo puede tomar valores enteros no negativos (esto es, el valor 0 o un valor entero positivo). La semántica de estos valores es: 0 semáforo cerrado, y >0 semáforo abierto. En función del rango de valores positivos que admiten, los semáforos se pueden clasificar en:

\*Semáforos binarios: Son aquellos que solo pueden tomar los valores 0 y 1.

\*Semáforos generales: Son aquellos que pueden tomar cualquier valor no negativos.

Frecuentemente, el que un semáforo sea binario o general, no es función de su estructura interna sino de cómo el programador lo maneja.

Los semáforos admiten dos operaciones:

\* Operación Wait (P): Si el valor del semáforo no es nulo, esta operación decrementa en uno el valor del semáforo. En el caso de que su valor sea nulo, la operación suspende el proceso que lo ejecuta y lo ubica en la lista del semáforo a la espera de que deje de ser nulo el valor.

\* Operación Signal (V): Incrementa el valor del semáforo, y en caso de que haya procesos en la lista de espera del semáforo, se activa uno de ellos para que concluya su operación Wait.

Lo importante del semáforo es que se garantiza que la operación de chequeo del valor del semáforo, y posterior actualización según proceda, es siempre segura respecto a otros accesos concurrentes.

Dijsktra define un semáforo como una variable entera positiva o nula sobre la que sólo se pueden realizar dos operaciones: wait (también denominada P) y signal (también denominada V). La operación wait decrementa el valor del semáforo siempre que éste tenga un valor mayor que 0; por lo tanto, esta operación se utiliza para adquirir el semáforo o para bloquearlo en el caso de que valga 0. La operación signal incrementa el valor del semáforo y por tanto se utiliza para liberarlo o inicializarlo.

Ambas operaciones deben ser atómicas para que funcionen correctamente; es decir que una operación wait no puede ser interrumpida por otra operación wait o signal sobre el mismo semáforo, y lo mismo ocurre para la operación signal. Este hecho garantiza que cuando varios procesos compitan por la adquisición de un semáforo, sólo uno de ellos va a poder realizar la operación.

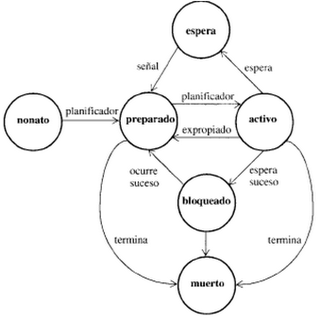
Además, se ha de indicar que este mecanismo memoriza las peticiones de operaciones wait no satisfechas y despierta por tanto a los procesos en espera.

Es una solución sencilla y elegante del problema de la exclusión mutua. Esta técnica permite solucionar la mayoría de los problemas de sincronización entre procesos. Un semáforo binario es un indicador de condición (s) que registra si un registro está disponible o no. Un semáforo sólo puede tomar dos valores; 0 y 1. Si para un semáforo, S=1 el recurso está disponible y la tarea lo puede utilizar; si S=0 el recurso no está disponible y el proceso debe esperar.

Los semáforos se implementan mediante una cola de tareas a la que se añaden los procesos que están en espera del recurso. Solo se permiten tres operaciones sobre un semáforo:

1. Inicializa (s: Semáforo\_Binario; v: integer) -- > poner el valor del semáforo s al valor de v (0,1).
2. Espera (wait)(s) if s = 1 then s: = 0 else Suspender la tarea que hace la llamada y ponerla en la cola de tareas.
3. Señal (signal)(s) if cola de tareas vacía then s: = 1 else Reanudar la primera tarea de la cola tareas.

Estas operaciones son procedimientos que se implementan como acciones indivisibles. En sistemas con un único procesador bastará simplemente con inhibir las interrupciones durante la ejecución de las operaciones del semáforo. Al introducir el semáforo se crea un nuevo estado en el diagrama de transiciones, el de espera.



**BIBLIOGRAFIA**

1. Tanenbaum A.S., Van Steen Maarten , “Sistemas Distribuidos: Principios y Paradigmas”, <http://climate.cs.buap.mx/CondeJC/cursos/Material/Distribuido_/Lecturas/300_ARQ_MIDDLEWARE_AUTOADMIN.pdf>
2. Colaboradores Pyspanishdoc, “Objetos cerrojo”, <http://pyspanishdoc.sourceforge.net/lib/lock-objects.html>
3. Anónimo, “Gestión de hilos de ejecución” , <http://bibing.us.es/proyectos/abreproy/11320/fichero/Capitulos%252F13.pdf>
4. Pita, Miguel Ángel, “Sincronización entre procesos” , <http://www.monografias.com/trabajos51/sincro-comunicacion/sincro-comunicacion2.shtml>
5. Colaboradores Wikipedia, “Problema sección crítica y posibles soluciones” <http://wiki.inf.utfsm.cl/index.php?title=Problema_secci%C3%B3n_cr%C3%ADtica_y_posibles_soluciones#Otros_problemas_de_Sincronizaci.C3.B3n>
6. J.M. Drake, “Sincronización basada en memoria compartida: Semáforos” <https://www.ctr.unican.es/asignaturas/procodis_3_II/Doc/Procodis_2_03.pdf>
7. Anónimo, “Mecanismo de Semáforo”

<http://eq2-sistemasoperativos.blogspot.com/2012/04/2421-mecanismo-de-semaforos.html>