Universidad Nacional Autónoma de México

Facultad de Ingeniería

Sistemas Operativos
2022-1

Algoritmos libres de bloqueo.

Nombres de los integrantes:
Díaz Hernández Marcos Bryan
Fuerte Martínez Nestor Enrique

Fecha de realización:
30/11/2021

Nombre del profesor:
ING. GUNNAR EYAL WOLF ISZAEVICH

Índice

Algoritmos libres de bloqueo.

Introducción.

En el ámbito de la informática y de los sistemas operativos a los algoritmos se les puede denominar “no bloqueo” si la falla de un hilo no puede ocasionar la falla de otro. En los algoritmos sin bloqueo podemos encontrar un progreso en todo el sistema, y a diferencia de los algoritmos con bloqueo el sistema no tiene que esperar si en un subproceso hay progreso.

Alrededor de la década de 1990 a este tipo de algoritmos se tenían que codificar de una manera más “nativa” ya que con este tipo de algoritmos se buscaba mejorar el rendimiento del procesador de la computadora a algo más aceptable. Al hablar de este tipo de algoritmos se usan las instrucciones atómicas, una de las que se usa más es la llamada “CAS” ya que esta se usaba junto con las arquitectura “**IBM 370**” y las que le sucedieron, para poder facilitar el procesador y por ende el proceso.

Alrededor del año 2013 la mayoría de las arquitecturas de multiprocesador empezaron a admitir la instrucción CAS en su hardware, otorgando más popularidad a este para implementarlo en estructuras de datos concurrentes.

Funcionamiento.

Para entender qué son los algoritmos libres de bloqueo primero debemos empezar con los algoritmos con bloqueo, a estos últimos se le llaman así porque contienen un “bloqueo mutuo”.

El bloqueo mutuo se da cuando uno o más procesos están constantemente esperando un evento en específico que nunca ocurrirá, entonces se dice que están bloqueados porque cada proceso retiene un poco de los recursos que se necesitan para que puedan finalizar, estos esperan a que se liberen los recursos retenidos por los procesos que están en el mismo grupo que ellos, pero como nunca lo liberarán, no podrán finalizar.

Un ejemplo de la vida real son los acumulamientos por tráfico, donde cada uno de los automóviles se puede ver como un proceso que está intentando llegar a su destino, pero como se obstruyen entre sí, ninguno de los procesos puede cumplir con su objetivo.

Ya con eso en mente, los algoritmos libres de bloqueo no tiene ese tipo de interrupciones, pero **¿Cómo logran quitar ese bloqueo?** pues la parte donde se encuentra el bloqueo la reemplazan con instrucciones atómicas primitivas de lectura, modificación y escritura; estas pueden ser “probar y configurar”, “buscar y agregar” y la que se ve más en este tipo de algoritmos “comparar e intercambiar”.

Principal uso.

Dentro del sistema operativo existen primitivas de bloqueo como lo son los mutex, semáforos, barreras, las cuales permiten establecer un orden entre los procesos y asegurar que la información que manipulan y modifican esté segura, pero la eficiencia de estos es limitada, debido a que normalmente estos bloquean el acceso a los datos por lo que se genera un ordenamiento indirecto que aumenta el tiempo de espera de cada proceso, lo que se traduce en que los procesos que esperan no hagan nada, y en un vistazo más global el sistema sea lento.

Primitivas libres de bloqueo.

Para poder comprender el funcionamiento de los algoritmos libres de bloqueos, es necesario comenzar por los elementos más fundamentales que nos permiten establecer un estado libre de bloqueo.

Acquire y Release.

Para la implementación de múltiples hilos en un sistema es necesario el sincronizar de alguna manera la ejecución, para esto se debe de conocer el orden en que se ejecutan los hilos.

Dentro de un sistema la cantidad de hilos que se ejecutan en un proceso permite establecer una relación entre estos hilos, sabiendo si un evento sucedió antes que otro. Cómo es posible conocer esta relación, bueno se puede saber mediante las primitivas de sincronización acquire y release. Para conocer el orden de ejecución entre dos procesos A y B, es clave saber cual envió el mensaje y cuál es el mensaje recibido.

Si estos hilos A y B, se ejecutan y A envía un mensaje de tal manera que B es el mensaje recibido, se puede asegurar que el hilo A se ejecutó primero y el hilo B se ejecutó después, esto es debido a que el hilo A tuvo que emitir un release para emitir el mensaje y el hilo B un acquire para poder recibir el mensaje.

```
thread 1                                thread 2
--------------------------------        ------------------------
a.x = 1;
smp_store_release(&message, &a);        datum = smp_load_acquire(&message);
                                        if (datum != NULL)
                                          printk("%x\n", datum->x);
```

En el ejemplo de arriba se puede ver como el hilo 1 modifica el valor de a.x=1 y posteriormente emite el release en forma de mensaje para que el hilo 2 pueda hacer el acquire, y hacer la impresión del valor.

```
a.x = 1;
   |
   v
smp_store_release(&message, &a);  ----->  datum = smp_load_acquire(&message);
                                                  |
                                                  v
                                          datum->x
```

Cómo es posible ver en el diagrama, se puede ver el flujo de la comunicación entre los hilos y la ejecución que tienen dentro del proceso.

Dekker's algorithm.

Si dos procesos intentan acceder a una sección crítica de información al mismo tiempo, el algoritmo previene que ambos procesos se excluyan, de tal manera que se elige a uno de los procesos para que pueda entrar a la sección mientras el otro proceso espera su entrada.

Para esto se utilizan dos banderas de estado, donde cada proceso las modifica en base a si está dentro de la sección crítica o está esperando entrar, la primer bandera es *wants_to_enter[0]* and *wants to enter[1]*, donde cada uno indica en su respectiva localidad el estado en que se encuentra y la bandera de estado *turn* que indica la prioridad para acceder a la sección crítica.

```
variables
    wants_to_enter : array of 2 booleans
    turn : integer

wants_to_enter[0] ← false
wants_to_enter[1] ← false
turn ← 0   // or 1
```

El pseudocódigo se muestra en la imagen anterior, donde se crea un arreglo de booleanos que indica el estado de cada uno de los procesos, y un entero que indica el turno en que se encuentra el acceso a la sección crítica, posteriormente se indica el estado de acceso como falso, y se indica el turno.

```
p0:
   wants_to_enter[0] ← true
   while wants_to_enter[1] {
      if turn ≠ 0 {
         wants_to_enter[0] ← false
         while turn ≠ 0 {
           // busy wait
         }
         wants_to_enter[0] ← true
      }
   }

   // critical section
   ...
   turn ← 1
   wants_to_enter[0] ← false
   // remainder section
```

```
p1:
   wants_to_enter[1] ← true
   while wants_to_enter[0] {
      if turn ≠ 1 {
         wants_to_enter[1] ← false
         while turn ≠ 1 {
           // busy wait
         }
         wants_to_enter[1] ← true
      }
   }

   // critical section
   ...
   turn ← 0
   wants_to_enter[1] ← false
   // remainder section
```

El algoritmo indica para cada proceso que ambos tienen la intención de entrar pero el criterio del turno da la prioridad para acceder a la sección crítica, en caso de que

no sea el turno del proceso este entra a un estado de espera y coloca como falsa su intención de entrar lo que libera al otro proceso del while y permite su acceso a la sección crítica.

En este caso existe exclusión debido a que uno de los procesos entrará a la sección pero el otro tendrá que esperar, aún así la exclusión no es tan grave debido a que se garantiza el acceso a la sección crítica. Pero solo es aplicable a dos procesos.

Compare and Swap (CAS).

Otro algoritmo que se puede implementar es compare-and-swap el funciona en la actualidad y que permite el manejo de los procesos de formal similar al algoritmo de Dekker's, pero con la ventaja de que este es mas general, mas fácil y veloz.

El prototipo general es el siguiente:

```
T cmpxchg(T *ptr, T old, T new);
```

¿Qué se supone que haga esta instrucción para que no haya bloqueo? bueno la principal dificultad a resolver es que los procesos se sincronicen y así no se bloqueen mutuamente, entonces lo que hace "comparar e intercambiar" (CAS) es justamente esto se usa esta instrucción cuando nuestro algoritmo ocupe multiprocesos para poder lograr esta sincronización entre ellos, lo que hace es checa el contenido que tengamos en una ubicación de memoria, la compara con un valor dado y únicamente si estos dos valores son iguales modifica el contenido de esa ubicación de memoria por un nuevo valor dado.

Todo esto se hace como una operación atómica, esto nos garantiza que el nuevo valor dado se calcule con la información actualizada, en dado caso que la información se haya actualizado por un hilo distinto, esa sobreescritura fallaría. Mediante un booleano y con el resultado de la operación se indica si hubo una sustitución.

Se puede decir que gracias a esto en los algoritmos libres de bloqueo la falla o suspensión de un hilo no puede causar la falla o suspensión de otro. En estos algoritmos si hay un progreso garantizado en el sistema.

Este algoritmo se ha implementado en las pilas y listas libres de bloqueos, pero se presenta el problema ABA al momento de que varios hilos intenten modificar las estructuras de datos.

```
int CAS( int *ptr, int oldvalue, int newvalue ){
    int temp = *ptr;
    if( *ptr == oldvalue ){
        *ptr = newvalue;
    }
    return temp;
}

void lock( int *mutex ){
    while( !CAS( mutex, 0, 1 ) );
}
```

Referencias

- tok.wiki, (2011). “Algoritmo de no bloqueo”. Hmong. Recuperado Noviembre 27, 2021, de https://hmong.es/wiki/Non-blocking_synchronization

- tok.wiki, (2013). “Comparar e intercambiar”. Hmong. Recuperado noviembre 27, 2021, de https://hmong.es/wiki/Compare-and-swap

- tok.wiki, (2013). “Leer-modificar-escribir”. Hmong. Recuperado noviembre 27, 2021, de https://hmong.es/wiki/Read-modify-write

- Desconocido,(2011).“7.Interbloqueo,bloqueo mutuo" .Microsoft word. Recuperado noviembre 27, 2021, de http://cs.uns.edu.ar/~jechaiz/sosd/clases/slides/05-Deadlocks-extra.pdf

- Wolf, G. (2017). “ Administración de procesos: Bloqueos mutuos y políticas” . Sistop. Recuperado noviembre 27, 2021, de http://gwolf.sistop.org/laminas/07-bloqueos-mutuos.pdf

- Kerrek, S. (2014). “Cómo funciona Comparar e Intercambiar”. it-swarm. Recuperado noviembre 27, 2021, de https://www.it-swarm-es.com/es/c/como-funciona-comparar-e-intercambiar/1044947618/

- LWN.net. (2021). “Lockless patterns: an introduction to compare-and-swap”. Lwn.net website. Recuperado Noviembre 28, 2021, de https://lwn.net/Articles/847973/?fbclid=IwAR0dlZ8uVSLAAJC3R5sPjxUM0OmBvcgJerpFPBlfOowPUZOjjghwhj3WMD4