Algoritmos libres de bloqueo.

Sistemas Operativos - 2022-1

Integrantes:

- Díaz Hernández Marcos Bryan
- Fuerte Martínez Nestor Enrique

INTRODUCCIÓN

- 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 se buscaba mejorar el rendimiento del procesador de la computadora a algo más aceptable. (Fig.1)

Fig. 1. Procesador IBM 370

- La instrucción atómica “CAS” se usaba comúnmente con las arquitectura “**IBM 370**” (Fig. 2) y las que le sucedieron.

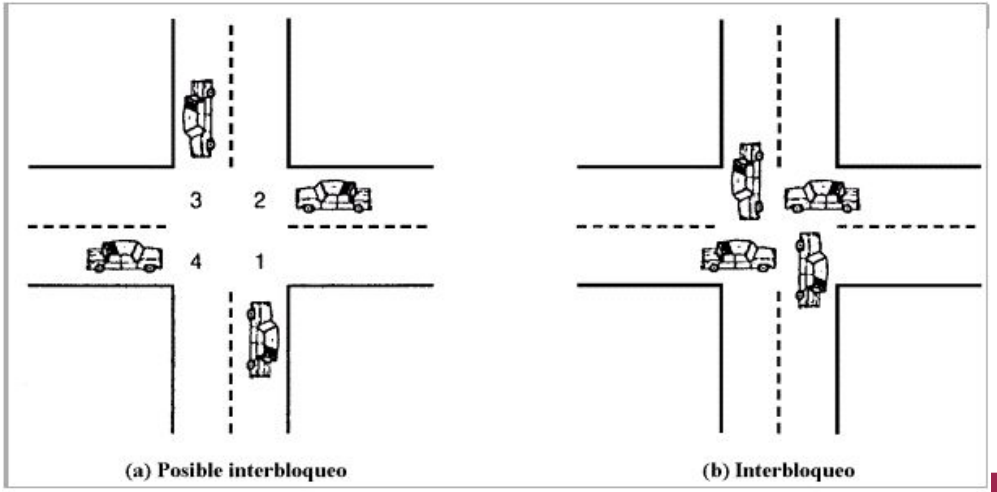
- Alrededor del año 2013 la mayoría de las arquitecturas de multiprocesador empezaron a admitir la instrucción CAS en su hardware.

Fig. 2. Sistema IBM 370

FUNCIONAMIENTO

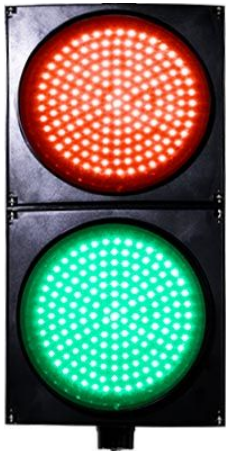
- El bloqueo mutuo se da cuando uno o más procesos están constantemente esperando un evento en específico que nunca ocurrirá.

- **¿Cómo logran quitar ese bloqueo?**
  - La parte donde se encuentra el bloqueo la reemplazan con instrucciones atómicas primitivas de lectura, modificación y escritura.
    * Test-and-set.
    * Fetch-and-add.
    * Compare-and-swap.



PRINCIPAL USO

- Dentro del sistema operativo existen primitivas de bloqueo como lo son los mutex, semáforos, barreras.
- Ordenar entre los procesos y asegurar que la información que manipulan y modifican esté segura.
- Normalmente genera un ordenamiento indirecto que aumenta el tiempo de espera de cada proceso.



PRIMITIVAS LIBRES DE BLOQUEO

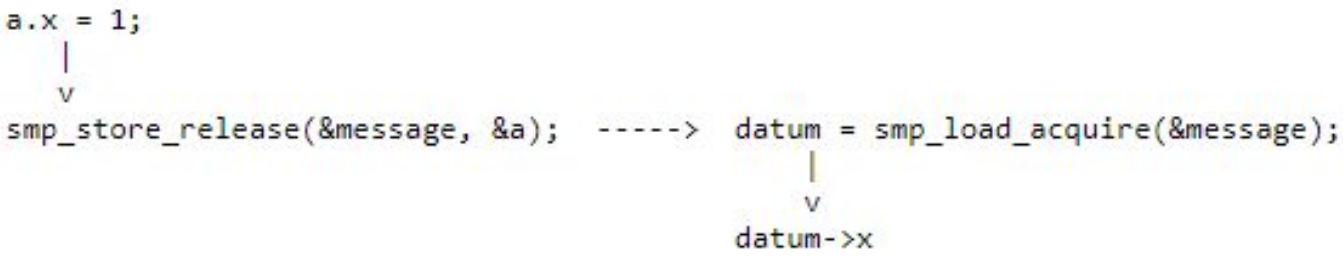
Acquire y Release.

- Si dos 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
--------------------------------
a.x = 1;
smp_store_release(&message, &a);
```

```
thread 2
------------------------

datum = smp_load_acquire(&message);
if (datum != NULL)
  printk("%x\n", 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.

- 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
```

```
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

- Se usa esta instrucción cuando nuestro algoritmo ocupe multiprocesos para poder lograr una sincronización entre ellos. 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 calcula con la información actualizada, en dado caso que la información se haya actualizado por un hilo distinto, esa sobreescritura fallaría.

- 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 ) );
}
```

LOCKLESS STACK - EJEMPLO

```
Node *top = NULL;     // top of stack

void push(Node *n) {
    do {
        Node *o = top;   // copy pointer
        n->next = o;
    } while (CAS(&top, o, n) == 0);
}
```

```
Node* pop() {
    Node *o, *n;
    do {
        o = top;          // copy pointer
        if (o == NULL)
            return NULL;
        n = o->next;
    } while (CAS(&top, o, n) == 0);
    return o;
}
```

CONCLUSIONES

- Analizando los diferentes algoritmos libres de bloqueos, podemos aprovechar el procesamiento concurrente, sin embargo es necesario que los procesos realicen comprobaciones constantemente para poder ejecutarse lo que nos provoca un aumento en el tiempo de ejecución del programa, debido a que se realizan comprobaciones constantemente.
- Independientemente a lo anterior, estos algoritmos nos garantizan un mejor rendimiento en el sistema ya que se evitan los bloqueos.

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=IwAR0dlZ8uVSLAAJC3R5sPjxUM0OmBvcgJerpFPBIfOowPUZOjjghwhj3WMD4