Sistemas Operativos

Patrón Request

Pablo Jaramillo

Orden de ejecución

Comportamiento Mutex

Cuando un mutex es liberado TODOS los threads que esperaban este intentarán tomarlo, solo UNO va a lograrlo, el orden de esta adquisición NO está garantizado.

Comportamiento Mutex

Cuando un mutex es liberado TODOS los threads que esperaban este intentarán tomarlo, solo UNO va a lograrlo, el orden de esta adquisición NO está garantizado.

Para evitar situaciones como esta → vamos a necesitar un patrón de diseño que garantice orden de forma eficiente.

Comportamiento Mutex

Cuando un mutex es liberado TODOS los threads que esperaban este intentarán tomarlo, solo UNO va a lograrlo, el orden de esta adquisición NO está garantizado.

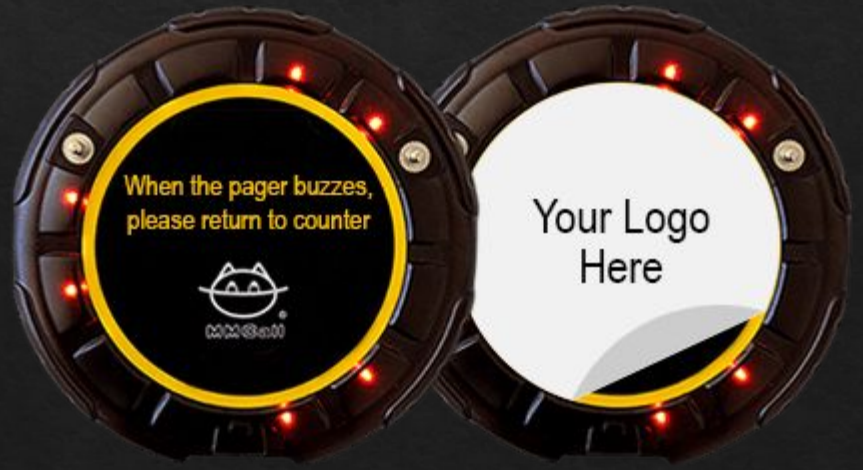
Para evitar situaciones como esta → vamos a necesitar un patrón de diseño que garantice orden de forma eficiente.

Patrón Request

Patrón Request

Podemos pensar sobre este como una forma de organizar “tickets” para poder entregar el núcleo a los threads de forma ordenada, según un orden que nosotros escojamos.

Similar a sistemas de ticket y beeper, pero de forma distribuida.



Un thread crea una request, y otro thread le indica cuando le toca y otros detalles que sean necesarios.

Patrón Request, how to?

Diseño

1. Diseñar estructura para hacer Requests
   a. Posee a lo menos: Indicador de cuándo está lista la petición y una herramienta para despertar el thread*
   b. Debe incluir toda la información necesaria para continuar.
2. Escoger estructura que permita acceder las Requests en el orden deseado.
3. Diseñar proceso para registrar una Request
4. Diseñar proceso para procesar Requests.

*Nota:
Esto para mutex implica un número que indique cuándo es correcto continuar y una condición respectivamente, pero para otras herramientas pueden ser otras cosas o ser combinadas.

Problema 1: Transporte de contenedores

Imaginemos una empresa que transporta contenedores. Sus clientes son threads a los cuales se les proveen camiones los cuales se maneja con la función **transportar**, esta se encarga de manejar el camión con el contenedor **cont**, desde la ciudad **orig** hasta la ciudad **dest**.

Problema 1: Transporte de Contenedores

Código antiguo

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
Camion *camion;
Ciudad *ubic= &Stgo;

void transportar(Contenedor *cont, Ciudad *orig, Ciudad *dest){
    pthread_muted_lock(&m);
    conducir(camion, ubic, orig);
    cargar(camion, cont);
    conducir(camion, orig, dest);
    descargar(camion, cont);
    ubic = dest;
    pthread_mutex_unlock(&m);
}
```

Problema 1: Transporte de Contenedores

- Las funciones: `cargar` , `conducir`, `descargar` existen y tardan mucho tiempo.
- `transportar` espera cuando el camión está ocupado por otro thread que llama a `transportar` .
- Antes de cargar un camión este debe llegar a la ciudad de origen.
- La función retorna después de que el camión haya sido cargado, viaje a la ciudad de destino y descargue el contenedor.

Problema 1: Transporte de Contenedores

Ahora la empresa tiene 8 camiones, por lo que todo el proceso anterior se puede paralelizar hasta para 8 clients a la vez. Para aquello debemos redefinir la función `transportar` para poder utilizar eficientemente los 8 camiones. Tenemos a nuestra disposición todo lo asumido hasta ahora y:

```
#define P 8
Camion *camiones[P];
Ciudad *ubicaciones[P];
double distancia(Ciudad * orig, Ciudad *dest);
```

Podemos asumir que inicialmente todos los camiones están en Santiago (o alguna otra ciudad).

Problema 1: Transporte de Contenedores

Restricciones:

- Un camión solo se puede usar por 1 thread a la vez.
- Un camión debe ser asignado una vez desocupado si hay peticiones pendientes. En particular se debe asignar a la petición cuyo origen sea más cercano al camión. Si no hay peticiones pendientes el camión se considera desocupado.
- Si al pedir un camión se encuentra varios desocupados se toma el más cercano.

* Se puede asumir un número máximo de peticiones y que este límite no será sobrepasado.

Problema 2: Subasta Aalsmeer

La subasta Aalsmeer es una tradición holandesa donde se subastan flores donde la oferta de uno está basada en el tiempo de la oferta. Partiendo desde un precio alto y bajando rápidamente hasta que alguien compre todo un lote.

Supongamos que nosotros vamos a inventar una versión de este sistema que permita vender el lote en partes, tal que los primeros postores son los que aseguran su parte del lote al mayor precio.

Problema 2: Subasta Aalsmeer

El problema de este sistema es que incentiva compras pequeñas, para cambiar esto implementaremos un mecanismo que motive mayores volúmenes de venta por descuentos sorpresa, tal que el precio unitario por flor sea el que diga el postor que llega después de uno mismo.

Cuando llegue el primer postor que no pueda comprar nada se considera cerrada la subasta y cada thread será responsable de despertar al que le precedió, con antes habiendo modificado el precio unitario a pagar.

Apoyándonos en la subasta original el precio unitario de cada thread será asignado de forma automática con la subida de la oferta, el precio asignado y la cantidad de unidades serán enviados a la función `double ofertar(Subasta *s, int *cantidad, double precio)`

Problema 2: Subasta Aalsmeer

Consideremos el siguiente caso ejemplo:
Hay en venta 20 flores.

Mateu llegó 1ro

Mateu quiere comprar
10 flores a $40 c/u

Lucas llegó 2do

Lucas quiere comprar
15 flores a $30 c/u

Pablo llegó 3ro

Pablo quiere comprar
5 flores a $25 c/u

Problema 2: Subasta Aalsmeer

Hay en venta 20 flores.

Pablo llegó 3ro

Pablo pudo comprar 0 flores, despertó al thread de Lucas

Lucas llegó 2do

Lucas pudo comprar 10 flores de las 15 que quiso y pago $25 c/u, despertó al thread de Mateu

Mateu llegó 1ro

Mateu compró sus 10 flores a $30 c/u