DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Nº 2

Primer cuatrimestre de 2016 — Sistemas Operativos

Integrante	LU	Correo electrónico
Martínez, Manuela	160/14	martinez.manuela.22@gmail.com
Rabinowicz, Lucía	105/14	lu.rabinowicz@gmail.com
Goldstein, Brian	027/14	brai.goldstein@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

1. Read-Write Lock

1.1. RWLock

Un read-write lock permite resolver problemas de sincronización cuando se tiene un recurso compartido que se quiere leer y escribir. En este caso, el problema de sincronización sucede entre threads.
El read-write lock permite que varios threads puedan leer el mismo recurso al mismo tiempo, pero a la hora de escribir, se permite uno solo.
En un primera implementación utilizamos 2 variables principales:

- 'escritor': cuando está prendida no se puede ni leer ni escribir(es booleana). Representando cuando alguien está escribiendo(ningún otro thread puede ni leer ni escribir).
- 'lectores': que contiene la cantidad de threads que están leyendo (cada vez que se realiza un rLock esta variable se incrementa, y cuando se realiza un rUnlock, se decrementa).

Con estas variables se nos posibilitó hacer un RWL ya que permitiríamos las lecturas siempre y cuando no se este escribiendo y permitiríamos la escritura cuando no se este leyendo ni escribiendo.
Pero para evitar inanición fue necesario también agregar un mutex al que llamamos 'pedidoDeLectura' que se 'tome' (lock) al comienzo del codigo del wlock() y se libere (unlock) recién en el wunlock. Este mutex se solicita también al principio de rlock() y se libera después al final del mismo.
El resultado entonces es que una vez que se realiza un pedido de escritura, los pedidos tantos de escritura como lectura futuros pasan a un estado de 'esperar a que se libere el mutex' y no se prosiguen hasta que el thread que pidió el wlock() lo libere usando wunlock(). Una vez liberado tanto los de lectura como los de escritura que estaban esperando podrán intentar escribir o leer.

1.2. Test RWLock

Para testear el RWLock implementado realizamos 3 test que testeen independientemente:

- El no-deadlock: Para verificar esto, creamos muchos threads tanto de lectura como de escritura intercalados y esperamos a que terminen. Como terminan podemos decir que no hubo deadlock.
- La no-inanición: para verificar la no inanición corremos varios threads de lectura y eventualmente cada tanto creamos uno de escritura y observamos que el orden en el que se crean sea aproximadamente el orden en el que se ejecutan, es decir que tanto una lectura como una escritura creada en el i-esimo thread, se ejecute aproximadamente i-esimo en el orden de ejecución y no suceda que por ejemplo una escritura creada cerca de comienzo se ejecute cerca del final, como sucedería de haber inanición.
- La no superposición de lecturas con escrituras: Que no se permita escribir durante una o varias lectura. Para esto ultimo creamos un thread que realiza una lectura muy larga(de mucha duración) y tratamos de realizar una escritura antes de que termine de leerse. Observamos que se espera a que termine la lectura y recién luego se procede a escribir.

2. Batalla Naval

En la implementación del servidor backend multithreaded, A cada cliente del juego La Batalla Naval, se lo atiende en un thread distinto. Estos threads se crean a partir de la función "*atendedorDeJugador*" que toma por parámetro el socket por el que se comunican el cliente con el servidor. Es en esta función donde se escribió el algorimo para asignar a cada cliente a su equipo y además es en esta función y en todas las demás con acceso a las variables globales en las que se protegieron estas variables con el uso de los RWL implementados previamente.

Las siguientes variables se dejaron en memoria global, ya que todos los threads necesitan accederlas:

- *tablero_equipo*1 y *tablero_equipo*2
- *peleando*
- *ancho* y *alto*
- *nombre_equipo*1 y *nombre_equipo*2
- *primero_del_equipo*1 y *primero_del_equipo*2
- *cant_jugadores*
- *cant_jugadores_listos*

Para mantener la sincronización se utilizaron los siguientes read-write lock:

- RWLock tablero1RWL
- RWLock tablero2RWL
- RWLock nombresRWL
- RWLock peleandoRWL

En el juego de la Batalla Naval solo se pueden tener dos equipos pero no esta restringida la cantidad de jugadores que pertenecen a cada uno de ellos. Cuando un cliente envía el nombre del equipo al que desea pertenecer primero se deberá decidir si el mismo es un nombre válido. En caso de que ya haya dos equipos inicializados se comparará el nombre ingresado con cada uno de ellos. Si no coincide con ninguno a este cliente no se le permitirá acceso al juego. En caso de que sea igual a alguno entonces se lo incluirá en ese equipo. Si solo se había inicializado un equipo, si el nombre ingresado coincide con éste entonces se lo incluirá en ese equipo. En caso contrario se inicializará un equipo con el nombre elegido por el jugador. Si no había ningún equipo inicializado entonces se creará un equipo con este nombre.
Es necesario saber en todo momento que tablero le corresponde al cliente y cual es el tablero rival ya que en caso de que el mismo pida actualizar se le deberá pasar su tablero en caso de estar en la parte BARCOS o el tablero enemigo en caso de que se encuentre en la BATALLA. Para ello se mantienen dos referencias, una a cada tablero (tablero_jugador y tablero_rival) con una referencia a sus respectivos RWLock (rwlRival y rwlJugador).
Luego cada vez que se necesite leer o escribir en el tablero será necesario poner un lock antes de accederlo y un unlock luego de ser accedido. Para que pueda comenzar la batalla cuando todos

pulsen la tecla "todos los barcos terminados"se mantendrá un contador que se inicializa en cero y se incrementará cada vez que un jugador presione esa tecla. Cuando este llegue a la cantidad de jugadores totales se cambiará la variable pelea indicando así que la pelea comenzó.

A continuación mostraremos un pseudo-código orientativo general y resumido del funcionamiento de el servidor de back-end:

BACKENDMULTI()

```
1 resolverSockets()
2 mientras true hacer
3 |   aceptar(cliente)
4 |   crearThread(atendedor_de_jugador, socket_cliente)
5 fin mientras
```

ATENDEDOR_DE_JUGADOR()

```
1 recibirNombre()
2 resolverEquipo()
3 mientras true hacer
4 |   comando ← recibirComando()
5 |   atender(comando)
6 fin mientras
```

Este pseudo código ilustra la idea global de la estructura: se atiende un nuevo usuario, se le asigna un thread y a partir de ese momento se pasa a recibir comandos. En cada ocación que un thread quiera usar la memoria compartida, utilizara el RWLock correspondiente a dicha memoria para mantener la integridad.