DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

Pthreads

Sistemas Operativos
Primer Cuatrimestre de 2016

Integrante	LU	Correo electrónico
Franco Frizzo	013/14	francofrizzo@gmail.com
Iván Pondal	078/14	ivan.pondal@gmail.com
Maximiliano Paz	251/14	m4xileon@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. Introducción

En el presente informe, se relatan a grandes rasgos los pasos que fueron necesarios para adaptar la implementación del juego *Batalla Naval* y así permitir la conexión simultánea de varios jugadores. Con este fin se utilizó la biblioteca de POSIX Threads (`pthreads`), haciendo cambios en el servidor de *backend* que atiende los pedidos de los distintos jugadores.

El informe se divide en dos secciones. En primer lugar, se detalla la implementación de un *read-write lock*, que fue necesario para ordenar los accesos a los recursos compartidos del juego evitando que se produzcan situaciones de *deadlock* o inanición. En la segunda sección, se especifican las modificaciones que se hicieron en el servidor propiamente dicho y en la forma en que el mismo atiende cada una de las solicitudes de los usuarios.

2. Read-Write Lock

Para cumplir con el requisito de que no haya inanición se necesitaron las siguientes estructuras:

- Mutex `read_write_mutex`
- Variable condicional `turn_cv`
- Contador de lectores `read_count`
- Contador de escritores `writing_count`
- Flag `writing`

El problema de inanición está presente si al tener una secuencia no acotada de pedidos de lectura o escritura al intentar la operación contraria esta queda bloqueada. A continuación se explican las medidas necesarias para evitar que suceda esto.

rlock()

```
Pedir mutex
Si el número de escritores es mayor a 0
    Esperar turno
Mientras haya una escritura en proceso
    Esperar turno
Aumentar en uno número de lectores
Liberar mutex
```

A la hora de llamar a `rlock()` se pide el mutex `read_write_mutex` y se procede a consultar si es posible la lectura. Para esto se pide el valor de `writing_count` donde si el mismo es mayor a 0 entonces es necesario esperar ya que de otra forma no se podría frenar una secuencia no acotada de pedidos de lectura. Además se consulta si la bandera `writing` está alzada, ya que es aquí donde el thread se quedará esperando hasta poder efectuar su lectura.

runlock()

```
Pedir mutex
Disminuir en uno número de lectores
Si el número de lectores es cero
    Dar turno a todos los que estén esperando uno
Liberar mutex
```

Cuando se termina la lectura, con `runlock()` se disminuye el valor de `read_count` tal que en caso de llegar a 0 se despiertan todos los threads que estuvieran esperando la condición `turn_cv`. Esto permite despertar a escritores que hubieran llegado en medio de una lectura.

wlock()

```
Pedir mutex
Aumentar en uno número de escritores
Mientras haya una escritura en proceso o el número de lectores sea mayor a cero
    Esperar turno
Subir bandera de escritura en proceso
Liberar mutex
```

Con respecto a los pedidos de escritura mediante `wlock()`, lo primero que se hace es aumentar el valor de `writing_count` para que en caso de llegar una pedido de lectura el mismo deba esperar. Luego se espera a que se cumplan dos condiciones: que `writing = false` y `read_count = 0`. Esto es necesario para evitar que haya más de una escritura en simultáneo y que únicamente se escriba cuando no hay nadie leyendo. Esta condición se cumple en algún momento ya que al llevar `writing_count` a un valor mayor a 0 los nuevos pedidos de lectura se ven obligados a esperar, permitiendo así que `read_count` disminuya más rápido de lo que aumenta llegando finalmente a 0. En el caso de `writing` se sabe que a lo sumo hay un thread escribiendo, por lo tanto el mismo tendrá que llamar a `wunlock()` bajando la bandera.

wunlock()

```
Pedir mutex
Bajar bandera de escritura en proceso
Disminuir en uno número de escritores
Dar turno a todos los que estén esperando uno
Liberar mutex
```

Una vez finalizada la escritura, `wunlock()` baja la bandera `writing`, diminuye el número de escritores `writing_count` y despierta a todos los threads que estén esperando la variable condicional `turn_cv`.

2.1. Inanición

Como se explicó en el funcionamiento de la estructura, la misma cuenta con mecanismos para evitar que queden pedidos ya sea de lectura o escritura sin atender por una secuencia no acotada de la operación contraria. Básicamente esto se reduce al hecho de que las lecturas deben esperar si el número de escritores es mayor a 0 y los escritores a que no queden threads leyendo ni escribiendo.

Cabe destacar que parte del motivo por el cual esto funciona es que `pthreads` asegura que si hay varios threads esperando a que se libere un mutex, al recibir un signal habrá *justicia* respecto quién lo recibe. Esto implica que no puede haber un thread esperando eternamente a que se le otorgue un mutex.

2.2. Deadlock

Para demostrar que no hay deadlock alcanza con probar que no se cumple la condición *hold and wait*. Esta condición necesaria pero no suficiente para que haya deadlock se cumple cuando hay un proceso que toma un recurso y no lo libera hasta no tener todo lo necesario para poder ejecutar su tarea.

En el read-write lock esto ocurriría si un thread en caso de no poder escribir o leer se quedará con el mutex, reteniéndolo hasta poder operar. Sin embargo, todos los mutex son liberados en cuanto el thread decide esperar a su respectiva variable de condición, por lo tanto nunca es retenido. A su vez, cuando un thread es despertado y su condición se cumple, el mutex asociado es liberado.

Por lo tanto no es posible que haya *hold and wait* en la implementación desarrollada, evitando así la existencia de un deadlock.

3. Servidor de backend

Para adaptar la implementación del servidor de *backend*, se utilizó `pthreads` para crear un *thread* por cada nuevo jugador que se conecta al servidor de *frontend*. La función que ejecutan estos *threads* es

`atendedor_de_jugador`, la cual fue adaptada tomando como base el trabajo realizado por los programadores *monoproceso-teístas*.

3.1. Variables globales

Los siguientes datos se almacenan de forma global y son accesibles en cualquier momento por todos los *threads*.

- El *socket* mediante el cual el servidor de *backend* se comunica con el servidor de *frontend*.
- Los nombres de los equipos en juego, que son dos como máximo: `nombre_equipo1` y `nombre_equipo2`. El acceso a los mismos se encuentra protegido por un *mutex*.
- Las dimensiones del tablero, `ancho` y `alto`, que se fijan por única vez al lanzar el servidor, y las posiciones en el mismo de las fichas de cada equipo, `tablero_equipo1` y `tablero_equipo2`, protegidas por un *read-write lock* diferente para cada equipo.
- Un contador, `cant_jugadores`, que permite conocer la cantidad de jugadores que ya se incorporaron al juego pero aún no terminaron de colocar su barcos (más adelante se detalla por qué esto es necesario). También se utiliza un *mutex* para proteger el acceso a esta variable.
- Una *flag* booleana, `peleando`, que indica si el juego está en la fase de batalla. Esta variable comienza con el valor `false` y es actualizada por única vez en forma conjunta con el contador recién mencionado. Por este motivo, no se utiliza un *mutex* particular para proteger el acceso a la misma.

3.2. Funcionalidad del juego

A continuación se detallan las modificaciones que fueron necesarias para adaptar cada una de las acciones que puede realizar un jugador, que se corresponden con los mensajes que acepta y a los cuales es capaz de responder la función `atendedor_de_jugador`.

Agregar un nuevo jugador

Cuando se conecta un nuevo jugador, se le solicita que ingrese un nombre de equipo. Si el nombre ingresado por el jugador corresponde a un equipo ya existente, se lo agrega a dicho equipo. En caso contrario, dado que hay un límite de dos equipos posibles, se verifica que no se haya alcanzado dicho límite. Si es posible, se crea un nuevo equipo con el nombre ingresado por el jugador y se lo agrega a este equipo; si no, se informa al jugador de que se produjo un error y se elimina el thread correspondiente (utilizando la función `pthread_exit`).

Cuando el juego se encuentra en la fase de armado de barcos, cada jugador que se incorpora incrementa en uno el contador de juegadores global. Dicho contador será luego decrementado cuando el jugador indique que sus barcos están listos, permitiendo conocer cuándo ya están listos todos los participantes, para así pasar a la fase de batalla. Una vez en esta fase, se permite agregar nuevos jugadores, pero los mismos ya no pueden sumar barcos al tablero.

Para acceder tanto a los nombres de los equipos como al contador de jugadores deben solicitarse los respectivos *mutex* que protegen estos recursos.

Agregar piezas a un barco

En primer lugar, un jugador solo puede agregar fichas al tablero si el juego aún no pasó a la fase de batalla; para verificar esto se chequea la *flag* global que lo indica, sin solicitar ningún *mutex* ya que no se alterará su contenido. Más aún, sabemos que mientras se esté procesando este pedido el juego no cambiará de fase, ya que para esto es requisito que todos los jugadores indiquen que están listos, lo cual incluye al jugador actual. Por otro lado debe verificarse que el propio jugador no haya indicado que ya está listo para la batalla, lo cual se conoce mediante una variable booleana local.

Luego debe verificarse que la ficha a colocar sea válida; dado que para esto es necesario acceder al tablero del jugador actual, se solicita momentáneamente acceso como lector al *read-write lock* correspondiente.

Si la ficha es correcta, se solicita el *read-write lock* de escritura. Aún falta verificar que la posición del tablero donde quiere colocarse una ficha no esté ocupada. Este chequeo debe hacerse de forma atómica con el agregado de la ficha, para evitar que dos usuarios intenten colocar simultáneamente una ficha en la misma posición. Si el casillero todavía esta libre, se agrega la ficha al tablero, guardando un registro de la misma como parte del barco actual por si debe se retirada, y luego se libera el *read-write lock*. Si el casillero está ocupado, se libera el *read-write lock* y se llama a la función auxiliar `quitar_partes_barco`, que elimina del tablero todas las piezas del barco actualmente en construcción. Esta función vuelve a solicitar el *read-write lock* para escritura, ya que necesita modificar el estado del tablero.

En el caso de que intente colocarse una ficha inválida, se utiliza también la función `quitar_partes_barco`.

Marcar un barco como terminado

En primer lugar, se verifica que el juego no haya pasado a la fase de batalla; como se explicó en el caso anterior, para esto no hace falta solicitar *mutex* alguno. Luego se solicita para escritura el *read-write lock* del tablero del jugador actual, se agregan al tablero todas las fichas del barco recién creado, y se vuelve a liberar el *lock* de escritura. Por último, se vacía el vector donde se mantenía el registro de las fichas de dicho barco.

Indicar que todos los barcos están listos

Cuando el jugador notifica que ya no agregará nuevos barcos y está listo para la batalla, el *thread* correspondiente debe, en primer lugar, actualizar su variable local `listo` con el valor `true`. Luego, para notificar de esta situación a los demás *threads* se pide el *mutex* que protege al contador de jugadores que aún no están listos. Este contador se decrementa en uno y, en caso de que haya llegado a cero (es decir, de que todos los jugadores hayan comunicado que están listos) se actualiza el valor de la *flag* global `peleando` a `true`, con lo cual el juego pasa a estar en la fase de batalla.

Tirar una bomba

La lógica de la implementación realizada por los programadores *monoproceso-teístas* para tirar una bomba ya era compatible con el modo multijugador, por lo que no hubo que hacer grandes modificaciones. Solo se necesitó pedir el *read-write lock* del tablero del rival como lector para conocer el contenido de la casilla deseada, y en caso de haber un barco, también solicitar el *read-write lock* como escritor para poder colocar la bomba en dicha casilla.

Actualizar el tablero

Para poder enviar a un jugador el estado actual del tablero, fue necesario solicitar como lector el *read-write lock* correspondiente. Para esto, se agregó una sencilla lógica que permite decidir, según la fase del juego y el equipo del jugador actual, cuál es el tablero a enviar, como así también solicitar el *read-write lock* apropiado.