DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Sistemas Operativos

Integrante	LU	Correo electrónico
Salinas, Pablo	456/10	salinas.pablom@gmail.com
Oller, Luca	667/13	ollerrr@live.com
Ituarte, Joaquin	457/13	joaquinituarte@hotmail.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

Índice

1. Introducción

En este informe explicaremos el desarrollo del código realizado para resolver los problemas del Trabajo Práctico 2 HaSObro y los experimentos realizados para verificar el correcto funcionamiento.

El trabajo consiste en un juego de batalla naval multijugador, donde los jugadores estarán separados en dos equipos. En la primera fase (que llamaremos 'Fase de construcción de barcos') cada jugador colocará sus barcos en casilleros adyacentes del tablero de su equipo de acuerdo al reglamento de la batalla naval, en el cual las fichas que componen un mismo barco deben estar alineadas horizontal o verticalmente entre ellas. Al terminar de armar un barco, el jugador indicará esto enviando el mensaje 'BARCO TERMINADO', y esta acción permite que pueda armar otros barcos si así lo desea.

Al terminar de poner barcos, cada jugador deberá indicar que ha terminado enviando el mensaje 'LISTO'. Cuando todos los jugadores de los dos equipos hayan enviado este mensaje, comenzará la batalla propiamente dicha. Durante la batalla, cada jugador podrá tirar bombas, buscando acertar a los barcos puestos por el equipo rival en la fase de construcción de barcos. En caso de haberle acertado a un barco, el servidor de backend devolverá el mensaje 'GOLPE'; si el barco del casillero bombardeado ya había sido golpeado previamente, el backend devolverá el mensaje 'ESTABA_GOLPEADO, y si no había ningún barco en el casillero el backend devolverá el mensaje OK.

La arquitectura del juego consiste en un servidor de frontend (que fue provisto por la cátedra) que atiende a los jugadores cuando estos se conectan para jugar, y un servidor de backend que, por cada conexión al frontend (es decir, por cada jugador) lanzará un thread para atender las interacciones del jugador con el frontend. Para garantizar la sincronización correcta entre todos los jugadores, debimos implementar previamente un mecanismo de sincronización entre threads basado en mutex y variables de condición POSIX, que explicaremos en la siguiente sección.

2. Ejercicios

2.1. Read-Write Lock libre de inanición

En este ejercicio se pide implementar la clase RWLock, que exportará los métodos rlock(), wlock(), runlock() y wunlock().

Por requisito del enunciado, en la implementación de la clase utilizamos únicamente variables de condición POSIX, es decir, no está permitido utilizar semáforos para sincronizar threads. Además, se requiere que no haya ni deadlock ni inanición, tanto para los lectores como para los escritores.

Para la implementación de la clase, utilizamos las siguientes variables:

- `room_is_empty`: booleano que indica si la sección crítica está vacía. Se setea a `True` en el constructor de la clase.
- `room_empty`: variable de condición que indicará que la sección crítica está vacía. Se inicializa con la función `pthread_cond_init`, seteando los atributos por defecto para una variable de condición.
- `room_empty_mutex`: mutex que acompaña a la variable de condición `room_empty`. Se inicializa con la función `pthread_mutex_init`, seteando los atributos por defecto para un mutex.
- `readers`: lleva la cuenta de la cantidad de threads accediendo al recurso en modo lectura. Se inicializa en 0.
- `mutex_readers`: mutex que serializa los accesos al contador `readers`. Se inicializa con los atributos por defecto para un mutex.
- `turnstile`: mutex que utilizaremos como molinete para los lectores y como mutex para los escritores (detallaremos esta idea más adelante). Se inicializa con los atributos por defecto para un mutex.

La idea de nuestra implementación es la siguiente:

La variable `turnstile` funcionará como molinete para los lectores (apenas obtengan el lock, lo liberarán) y como mutex para los escritores (lo liberarán al terminar de escribir el recurso compartido). De esta manera, nos aseguramos de que varios lectores puedan acceder concurrentemente al recurso, pero que los escritores, al obtener ese lock (y no liberarlo inmediatamente como hacen los lectores) dejen bloqueados a todos los otros threads hasta haberlo liberado (adicionalmente, esto impide que los escritores tengan inanición en el caso en que, habiendo lectores presentes accediendo al recurso, sigan llegando lectores que puedan seguir accediendo, ya que ambos tipos de thread competirán por turnstile).

La variable `mutex_readers` se usará para proteger el contador `readers`, para impedir race conditions al accederla. Cuando llegue el primer lector, este deberá esperar a que la sección crítica esté libre de escritores (mediante la variable de condición) y luego actualizar la variable `room_empty` para indicar que la sección crítica ya no está vacía (el próximo lector que llegue no tendrá que hacer esta comprobación, ya que con uno o más lectores accediendo al recurso pueden obtener directamente el lock de lectura). En cambio, cuando llegue un escritor y solicite un lock de escritura, una vez que obtenga el lock sobre `turnstile`, únicamente deberá esperar a que la sección crítica esté vacía.

Cuando un lector desee liberar su lock de lectura, decrementará el contador `readers`. Adicionalmente, deberá considerar el caso borde en el cual él sea el último lector liberando el lock sobre el recurso, caso en el cual deberá avisar que la sección crítica está quedando vacía (mediante un `signal` sobre la variable de condición `room_empty`). En cambio, si es un escritor el que está liberando el lock de escritura, no deberá hacer esta validación ya que, al ser excluyente el acceso de los escritores, siempre estará garantizado que la sección crítica está quedando vacía cuando un escritor libera el lock sobre el recurso (con lo cual, siempre que libere el lock de escritura deberá indicar con un `signal` sobre `room_empty` que

dejó libre el recurso).

A continuación, exhibiremos un breve pseudocódigo por cada uno de los métodos de la clase `RWLock`:

- `Rlock()`: pide un lock de lectura sobre el recurso.

Algorithm 1 rlock

```
1:  procedure RLOCK( )                          ▷ paso turnstile y lo libero inmediatamente despues
2:      pthread_mutex_lock(&turnstile)
3:      pthread_mutex_unlock(&turnstile)        ▷ Pido el mutex para verificar cuantos lectores hay en la
    seccion critica
4:      pthread_mutex_lock(&mutex_readers)
5:      readers++          ▷ Si soy el primer lector, espero hasta que se vaya el escritor, si es que hay uno
6:      if readers == 1 then
7:          pthread_mutex_lock(&room_empty_mutex);
8:          while !room_is_empty do
9:              pthread_cond_wait(&room_empty, &room_empty_mutex)
10:             room_is_empty = false
11:             pthread_mutex_unlock(&room_empty_mutex)
12:     pthread_mutex_unlock(&mutex_readers);
```

- `Wlock()`: pide un lock de escritura sobre el recurso.

Algorithm 2 wlock

```
1: procedure WLOCK( )          ▷ trabo turnstile, para impedirle a futuros lectores matarme de inanicion mientras espero mi turno
2:     pthread_mutex_lock(&turnstile)          ▷ espero a que la seccion critica este vacia
3:     pthread_mutex_lock(&room_empty_mutex)
4:     while !room_is_empty do
5:         pthread_cond_wait(&room_empty, &room_empty_mutex);
6:     room_is_empty = false
7:     pthread_mutex_unlock(&room_empty_mutex)
```

- `Runlock()`: libera el lock de lectura sobre el recurso.

Algorithm 3 runlock

```
1: procedure RUNLOCK( )
2:     pthread_mutex_lock(&mutex_readers)
3:     readers –
4:     if readers == 0 then          ▷ si soy el ultimo lector saliendo de la sección critica, aviso que esta quedando vacía.
5:         pthread_mutex_lock(&room_empty_mutex)
6:         room_is_empty = true
7:         pthread_cond_signal(&room_empty)
8:         pthread_mutex_unlock(&room_empty_mutex);
9:     pthread_mutex_unlock(&mutex_readers)
```

- `Wunlock()` libera el lock de escritura sobre el recurso.

Algorithm 4 wunlock

```
1: procedure WUNLOCK( )          ▷ libero turnstile, para volver a dejar pasar a otros lectores/escritores.
2:     pthread_mutex_unlock(&turnstile)          ▷ aviso que la sección critica quedo libre.
3:     pthread_mutex_lock(&room_empty_mutex)
4:     room_is_empty = true
5:     pthread_cond_signal(&room_empty)
6:     pthread_mutex_unlock(&room_empty_mutex)
```

2.2. Read-Write Lock Test

En este ejercicio, se pide implementar tests para el punto anterior que utilicen threads lectores y escritores, para verificar que nuestra implementación del RWLock cumple con los requisitos de sincronización y que, además, está libre de inanición y deadlock.

Para verificar esto, creamos el archivo RWLockTest.cpp, con el siguiente contenido:

Una clase ThreadParameters, que contendrá los siguientes parámetros de entrada para los threads: id del thread (numérico, autoincremental) y un file descriptor, que representará el recurso por el que los threads competirán para leer/escribir. Adicionalmente, habrá una instancia de la clase RWLock (visible para todos los threads), que será la que los lectores y escritores soliciten para acceder al recurso.

Luego, implementamos tres casos de prueba, que simularán distintos escenarios de uso para la clase RWLock:

- Un escenario donde todos los threads lanzados sean lectores.
- Un escenario donde todos los threads lanzados sean escritores.
- Un escenario mixto, donde algunos threads serán lectores y otros escritores (qué tipo de thread será cada uno se tomará en función de su número de id).

A continuación, mostramos un pseudocódigo que describe los tres casos de prueba implementados:

Algorithm 5 entry_function

```
1:  procedure VOID* ENTRY_FUNCTION(void* params)        ▷ Obtengo los parámetros del thread.
2:      ThreadParameters* parameters = (ThreadParameters*) params
3:      int my_id = parameters->thread_id
4:      FILE* f = parameters->the_file
5:      if soy_escritor then
                                                                  ▷ pido el lock de escritura.
6:          the_lock.wlock();
                                                                  ▷ imprimo la salida
7:          fprintf(f, "Soy el thread %d entrando a escribir ", my_id)
8:          sleep(1)
9:          fprintf(f, "Soy el thread %d saliendo de escribir ", my_id)
                                                                  ▷ libero el lock de escritura.
10:         the_lock.wunlock()
11:     else
                                                                  ▷ pido el lock de lectura
12:         the_lock.rlock()                                      ▷ imprimo la salida
13:         fprintf(f, "Soy el thread %d entrando a leer", my_id);
14:         sleep(1);
15:         fprintf(f, "Soy el thread %d saliendo de leer", my_id);   ▷ libero el lock de lectura
16:         the_lock.runlock()
17:     return NULL
```

Para cualquiera de los tres tests implementados, el orden de las lineas en el archivo de salida debe indicar que:

- Los threads lectores pueden acceder concurrentemente al recurso.
- Los threads escritores tienen acceso excluyente al recurso.

Después de ejecutar nuestras pruebas, pudimos observar los siguientes resultados:

- En el test de solo lectores, todos obtienen el lock de lectura sobre el recurso y luego lo liberan, pero no necesariamente en el mismo orden en el que accedieron.
- En el test de escritores, se puede ver que el acceso está totalmente serializado: un thread obtiene el lock de escritura y, hasta no haberlo liberado, ningún otro thread lo puede obtener.
- En el test mixto, observamos que los lectores tienen acceso compartido entre ellos, pero una vez que un escritor obtiene el lock de escritura, nadie más obtiene el lock hasta que el escritor no lo haya liberado. Adicionalmente, observamos que no hay riesgo de inanición para ninguno de los dos tipos de acceso: no se observa que los lectores ni los escritores obtengan todos juntos el acceso al recurso una vez que todos los threads del otro tipo lo liberaron (es decir, no se ve una sucesión de líneas al final del archivo de salida que indique que todos los lectores tuvieron que esperar a que terminaran todos los escritores, ni tampoco el caso reverso).

2.3. Servidor de Backend Multithreaded

Para la implementación del servidor de backend, se lanza un thread por cada jugador, para que atienda los mensajes que el jugador envía a través del frontend. El servidor de backend guardará las siguientes variables globales (visibles para todos los threads jugadores) el estado actual del juego:

- Dos tableros que almacenarán los barcos terminados de cada equipo.
- Dos tableros auxiliares con los barcos que están a medio armar en cada equipo.
- Dos tableros, donde cada posición contiene una variable de tipo RWLock, para sincronizar las lecturas y las escrituras a los tableros de cada equipo.

Adicionalmente, el servidor de backend guardará otras variables para implementar la lógica del juego (contadores para el total de jugadores por equipo, el total de jugadores que terminaron de poner sus barcos, además de un flag que indicará si el servidor seguirá aceptando conexiones de nuevos jugadores) que no explicaremos en detalle en este informe, ya que nos concentraremos en los mecanismos usados para sincronizar a los threads.

Para la fase de construcción de barcos, cada jugador irá haciendo las escrituras en el tablero auxiliar. Al ser un recurso compartido, deberán solicitar el lock de lectura/escritura correspondiente al casillero antes de poder leer/escribir el tablero auxiliar. Cuando un jugador indique que terminó de construir el barco actual, el thread que atiende al jugador se encargará de escribir el barco armado por el jugador al tablero de barcos terminados, solicitando también el uso del recurso compartido mediante un lock de escritura sobre los casilleros en los que el jugador ubicó el barco. Para este punto del juego, tomamos la decisión de que las casillas que contienen un barco incompleto de algún jugador del equipo no puedan ser utilizadas por otro jugador del equipo para construir sus barcos. Tomamos esta decisión ya que consideramos que es lo que más se asemeja a cómo se jugaría este juego con un tablero físico: al no haber turnos para poner barcos entre los jugadores de un mismo equipo, consideramos que un casillero con un barco incompleto ya está 'reservado' para el jugador que está ubicando su barco allí. Otra decisión tomada que vale la pena aclarar es la siguiente: una vez que ambos equipos tengan jugadores y que uno de ellos haya terminado de poner todos sus barcos, no se permitirán más conexiones al servidor. Tomamos esta decisión porque, sin un criterio para dejar de aceptar jugadores, la etapa de construcción de barcos podría extenderse infinitamente, ya que podría interpretarse que los jugadores recién ingresados no terminaron de poner sus barcos, con lo cual no se podría avanzar a la fase de batalla.

Una vez que todos los jugadores de ambos equipos terminaron de ubicar sus barcos, se procede a la batalla entre los equipos. En esta etapa, cada jugador puede tirar bombas sobre el tablero rival, buscando acertarle a los barcos construidos por el equipo rival en la fase anterior. Como el enunciado especifica que un mismo casillero ocupado con la parte de un barco no puede ser golpeado dos veces (es decir, en el segundo golpe el servidor debe informar que esa posición ya había sido golpeada), el lanzamiento de bombas debe ser sincronizado con el mismo mecanismo de la fase previa: cuando un jugador lance una bomba en una posición, el thread que atiende sus pedidos solicitará un lock de escritura sobre el casillero, verificará que la posición no haya sido previamente golpeada (en cuyo caso devolverá el mensaje 'ESTABA_GOLPEADO'), escribirá una bomba en la casilla y liberará el lock de escritura.

Para probar el correcto funcionamiento de la sincronización entre threads, al tener ya testeado el funcionamiento del lock de lectura/escritura sobre un recurso, hicimos pruebas manuales de integración sobre el juego, levantando tanto el servidor de frontend como el de backend. En estas pruebas, verificamos los casos que podrían llegar a ser problemáticos por la sincronización entre los jugadores:

- Escrituras concurrentes sobre los mismos casilleros, para verificar que el que llegue primero pueda ubicar allí su barco, mientras que el segundo no pueda usar el casillero disputado para hacerlo.
- Lanzamientos de bombas concurrentes a los mismos casilleros, para verificar que el que lance primero la bomba reciba el mensaje 'GOLPE', mientras que segundo reciba el mensaje 'ESTABA_GOLPEADO'.

Finalmente, verificamos también que los aspectos relacionados a la lógica del juego respeten el reglamento de la batalla naval especificada por la cátedra:

- La construcción de cada barco debe respetar una única orientación (horizontal o vertical)
- Los jugadores que ya indicaron que terminaron de ubicar sus barcos no pueden lanzar bombas al tablero del equipo rival hasta que todos los jugadores del equipo rival hayan terminado de ubicar sus barcos.
- Al actualizar el estado del tablero, se debe poder ver los barcos que estén terminados, mientras que los barcos de otros jugadores que estén sin terminar no deben estar visibles para los otros jugadores.
- Al terminar la conexión de un jugador, se deben descartar cualquier barco que dicho jugador tuviera sin terminar, pero los que estuvieran terminados deben preservarse en el tablero.