DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico: Pthreads

26 de noviembre de 2015 — Sistemas Operativos

Integrante	LU	Correo electrónico
Lebrero, Ignacio	751/13	ignaciolebrero@gmail.com
Vázquez, Jésica	318/13	jesis_93@hotmail.com
Arribas, Joaquín	702/13	joacoarribas@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. Problema

Fuimos encargados con la tarea de implementar un juego que permite múltiples usuarios a la vez. Nuestro trabajo empezó donde lo dejaron otros, un sistema para que un solo jugador pueda divertirse. En este sentido, adaptamos el juego de un usuario a múltiples posibles jugadores.

1.1. Cambios al código del servidor Backend

Inicialmente tenemos un único servidor de *frontend* que puede atender a varios *browsers* (usuarios) que se quieren conectar para jugar. El servidor inicia una conexión TCP por cada uno de los *browsers* que se conectan. El problema es que el servidor de backend recibe una única conexión TCP, permitiendo así un sólo jugador a la vez.

Para solucionar dicho problema utilizamos *Pthreads*. Inicialmente el servidor creaba un socket y permitía una conexión de un único usuario, lo cual modificamos para que pueda realizar más conexiones (el valor está seteado en 50, pero es modificable dependiendo de cuántas sean necesarias).

Luego por cada conexión aceptada mediante la acción *accept* se crea un nuevo *thread* el cual se encaga de atender dicha conexión. De esta manera podemos atender varias conexiones en simultáneo.

Como tenemos varios *threads* (usuarios) interactuando con un único tablero, fue necesaria la implementación de *mutexes* para salvaguardar los problemas de concurrencia que surgen cuando se quiere lograr paralelismo en el programa. Por cada aparición de las variables globales dentro del código, dependiendo de si se escribía en ellas, o simplemente se deseaba leer, se utiliza su correspondiente *mutex*.

1.2. implementación de multiples conexiones

Para este punto se modificaron dos partes:

- La sección encargada de levantar la conexión entrante:

```
while (true) {
    if ((socketfd_cliente = accept(...)) {
      ...
    } else {
        close(socket_servidor);
        atendedor_de_jugador(socketfd_cliente);
    }
}
```

Fue reemplazada por:

```
while (true) {
    if ((socketfd_cliente = accept(...)) {
        ...
    } else {
        pthread_t t;
        pthread_create(
            &t,
            NULL,(void* (*)(void*)) &atendedor_de_jugador,
            (void*) (long) socketfd_cliente
        );
    }
}
```

De esta manera el servidor sigue escuchando nuevas conexiones y por cada una que ingrese se lanza un nuevo thread para que ésta sea atendida.

- $listen(socket\_servidor, 1)$ fue cambiado por $listen(socket\_servidor, 50)$ de manera que ahora escucha hasta 50 conexiones.

1.3. Implementación de mutexes

Para la implementación de los mutexes utilizamos las siguientes variables:

- $\_cantLecturas$ que cuenta la cantidad de *threads* que están esperando a que se liberen las variables globales para poder leer su valor.
- $\_cantLecturasHasta$ que cuenta la cantidad de lectores que llegaron antes de un *thread* que deseaba escribir.
- $\_mCantLecturas$ es el *mutex* utilizado para proteger las variables que cuentan la cantidad de lectores.
- $\_mEscribiendo$ es el *mutex* utilizado para determinar si hay alguien en la sección crítica del código escribiendo.
- $\_cond\_cantLecturas$ es la variable de condición utilizada para que los *threads* que desean escribir sepan que pueden proceder (su funcionamiento está explicado más abajo).

A continuación exponemos un pseudocódigo de nuestra implementación

Pseudocodigo 1 **void** Wlock()

Pedir *_mEscribiendo* para ser único en la sección crítica al momento de escribir
Pedir *_mCantLecturas* para leer las variables *_cantLecturas* y *_cantLecturasHasta*
Asignar a *_cantLecturasHasta* el valor de *_cantLecturas*
while La cantidad de lectores que estaban cuando leí *_cantLecturas* sea mayor a cero **do**
  Esperar a que sea cero (que no haya lectores en la sección crítica)
end while
Liberar el mutex *_cantLecturas* que retomé cuando me desperté de la variable de condición

Como un sólo *thread* puede escribir en las variables globales a la vez, manteniendo el mutex *_mEscribiendo* desde el principio de la rutina puedo asegurar que no va a haber problemas de concurrencia en cuanto a los escritores.

Pseudocodigo 2 **void** Wunlock()

Liberar el mutex *_mEstanEscribiendo* y *_mEscribiendo* para avisar que terminó la escritura

Recién cuando termina de escribir el *thread* le avisa al próximo escritor que puede proceder a fijarse si puede escribir.

Pseudocodigo 3 **void** Rlock()

Pedir *_mEscribiendo* para verificar que no haya ningún escritor en la sección crítica
Pedir *_mCantLecturas* para leer la variable *_cantLecturas* y aumentar en 1 (ya que voy a leer)
Liberar *_mCantLecturas*
Liberar el mutex *_cantLecturas*
Liberar *_mEscribiendo* sabiendo que aumente de manera válida cantLecturas y que el próximo escritor me tendrá en cuenta

Como las lecturas pueden ser concurrentes (no hay ningún peligro de alteración de datos) no se restringue a que un sólo *thread* pueda leer las variables globales, es decir, pueden haber múltiples *threads* consultando los valores de las variables.

Pseudocodigo 4 **void** Runlock()

Pedir *_mCantLecturas* para leer la variable *_cantLecturas* y *_cantLecturasHasta*
if *_mCantLecturasHasta* es mayor a cero **then**
  Decrementar en 1 porque ya leí
end if
if *_mCantLecturasHasta* es igual a cero **then**
  Avisar al escritor que estaba esperando que ya puede escribir
end if
Decrementar en 1 *_cantLecturas*
Liberar el mutex *_mCantLecturas*

Cada vez que un lector termina de leer decrementa en uno la cantidad de lectores que estaban deseando acceder a la sección crítica. Si alguien estaba esperando a que *_cantLecturasHasta* lectores terminen de leer, le mando una señal.

1.4. Programa libre de *DeadLock*

Evaluando el código podemos observar que en ningún momento sucede que varios *threads* esperan un recurso que tiene otro y viceversa. Se pueden correr los tests en el archivo *RWLockTests.cpp* que muestran que no ocurre dicho evento.

1.5. Programa libre de inanición

Las lecturas son concurrentes así que siempre que algún *thread* quiera leer, excepto que haya alguien escribiendo, podrá hacerlo. Para que un *thread* pueda escribir necesitará esperar a que determinada cantidad de lectores haya terminado de leer. Esto se ve reflejado en la asignación a $\_cantLecturasHasta$ que contabiliza la cantidad de lectores que había al momento de poder proceder a esperar para escribir. Analizamos los siguientes casos:

- Hay varios lectores en la sección crítica y llegan varios escritores. En este caso podemos asegurar que el escritor esperará un tiempo acotado para poder escribir ya que instanciará la variable $\_cantLecturasHasta$ en la cantidad de lectores que hay en el momento que llegó, de manera tal que, una vez que esa cantidad finalice, el podrá ingresar a la sección crítica. Una vez que termine, liberara el mutex $\_mEscribiendo$ permitiendo así que el proximo escritor lleve cuenta de cuantos lectores debe dejar pasar.
- Hay varios escritores esperando a escribir y llega un lector. El lector quedará "trabado" en el mutex $\_mEscribiendo$ hasta que uno o más escritores terminen de escribir. Como las implementaciones del mutex tienen un cierto $fairness$ en cuanto a quien lo recibe, podemos asegurar que eventualmente el lector obtendrá el recurso, podrá aumentar $\_cantLecturas$ y entrar a la sección crítica. El próximo que quiera escribir tendrá que esperar a que el lector finalice.

1.6. Casos de prueba

Los *tests* que usamos para corroborar que no hubiera inanición se usaron funciones de escritura y lectura que realizan 300 veces su operación cada vez que son llamadas. Las lecturas simplente imprimen el valor de la variable mientras que las escrituras aumentan en uno su valor. Los *test* son:

- Lanzar 100 threads de escritura(haciendo que en total haya 3000) y luego hacer una lectura y para ver que 3000 sea el valor de la variable.
- Lanzar 76 threads de escritura y la misma cantidad de lectura de a pares de a 4 a la vez y hacer $join$ de los mismos, de manera de ver que estos terminen.
- Idem al *test* anterior pero haciendo $join$ luego de haber lanzado todos los threads.