Departamento de Computación,
Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires

TP2 - Pthreads

Sistemas Operativos

Segundo Cuatrimestre de 2015

Apellido y Nombre	LU	E-mail
Shaurli Tomas	671/10	zeratulzero@hotmail.com
Tripodi, Guido	843/10	guido.tripodi@hotmail.com

Contents

1 Desarrollo y Resultados

2 Parte I – Desarrollo de Read-Write Lock

2.1 Ejercicios

- **Ejercicio 1** En primer lugar, deberán implementar un Read-Write Lock libre de inanición utilizando únicamente Variables de Condición POSIX y respetando la interfaz provista en los archivos backend-multi/RWLock.h y backend-multi/RWLock.cpp

2.2 Resultados y Conclusiones

2.2.1 Ejercicio 1

Nuestra implementación del Read-Write Lock se baso en el pseudocodigo implementado en el libro *The Little Book of Semaphores* al resolver la inanición producida en el problema de $Readers - writers$.

Se utilizaron 3 Semaphores, los cuales son:

- roomEmpty
- turnstile
- readers_mutex

Y ademas, un entero denominado *readers*

Comenzando por la implementación de los lectores, el pseudocodigo del libro mencionado es el siguiente:

```
turnstile.wait()
turnstile.signal()

readSwitch.lock(roomEmpty)
    # critical section for readers
readSwitch.unlock(roomEmpty)
```

De aquí nuestro código implementado fue el siguiente:

READERS LOCK

```
pthread_mutex_lock(&turnstile);
pthread_mutex_unlock(&turnstile);

pthread_mutex_lock(&readers_mutex);
readers++;
pthread_mutex_unlock(&readers_mutex);
```

Como se puede observar en el codigo del lock del read, el lector realiza un lock (wait) y unlock (signal) del Semaphores *turnstile* para tener su turno y que ningún otro lo saque.
Por consiguiente, se realiza el lock del mutex que se encuentra vinculado al entero *readers*, ya que este será aumentará su cantidad en 1 para que de esta manera nadie pueda modificarlo, y luego es liberado dicho mutex (*readers_mutex*).

Luego, nuestro $READ\ UNLOCK$ fue el siguiente:

READERS UNLOCK

```
pthread_mutex_lock(&readers_mutex);
readers--;
if (readers == 0) {
    pthread_cond_signal(&room_empty);
}
pthread_mutex_unlock(&readers_mutex);
```

En esta implementación, primero se realiza un lock del Semaphore vinculado al entero *readers* ya que este disminuirá en 1.
Luego, se realiza una consulta chequeando el valor del entero, si este es 0 se le dara un signal al Semaphore *room_empty* para notificarle al escritor que ya no queda ningun lector y puede proceder a escribir.
Por último, se libera el mutex *readers_mutex*.

Continuando con el escritor, el pseudocodigo fue el siguiente:

```
turnstile.wait()
roomEmpty.wait()
    # critical section for writers
turnstile.signal()

roomEmpty.signal()
```

De aquí, nuestra implementación final fue:

WRITERS LOCK

```
pthread_mutex_lock(&turnstile);
pthread_mutex_lock(&readers_mutex);
while(readers != 0)
      pthread_cond_wait(&room_empty, &readers_mutex);
pthread_mutex_unlock(&readers_mutex);
```

Inicialmente en nuestra implementación del WRITE LOCK, se realiza un lock del Semaphore *turnstile* para que nadie pueda quitarle el turno, se realiza un lock del mutex vinculado al entero, y luego se ingresa a un ciclo siempre que *readers* sea distinto de 0, esto se realiza para luego poder ejecutar la funcion *pthread_cond_wait* para que esta misma tenga un funcionamiento correcto y seguro al chequear la condición sobre *room_empty*.
Una vez que se salga del ciclo o no se ingrese al mismo se libera el mutex *readers_mutex*.

Luego, la implementación del unlock fue:

WRITERS UNLOCK

```
pthread_mutex_unlock(&turnstile);
```

Para la implementación del unlock del writer solo se libera el Semaphore *turnstile*.

De esta manera, con dicha implementación, siempre que llegue un escritor el mismo tendrá su turno sin producirse inanición. Ya que, en caso de haber lectores y llegar un escritor, estos terminaran de leer y en caso de llegar nuevos lectores deberán esperar a que el escritor finalice su ejecución.

3 Parte II: Desarrollo de Backend Multithreaded

3.1 Ejercicios

- **Ejercicio 2** En segundo lugar, deberán implementar el servidor de backend multithreaded inspirándose en el código provisto y lo desarrollado en el punto anterior.

3.2 Resultados y Conclusiones

3.2.1 Ejercicio 3

En este Ejercicio, se solicito la implementación de un backend mutithreaded, para el desarrollo del mismo, utilizamos la base del backend mono para la conexion con el servidor, el parseo de las fichas, tanto para la validez de las mismas y tambien el envio de dimensiones del tablero de juego.

Utilizamos la función *atendedor_de_jugador* la cual la convertimos en un thread para cada jugador. Esta función es llamada desde la función main cuando las conexiones del socket entre el cliente-servidor son correctas.
Convertimos la función enunciada de la siguiente manera:

```
pthread_create(&threads[i], NULL, &atendedor_de_jugador, (void *)&td[i]);
```

En donde, threads es un arreglo de thread_t y se le asigna uno a cada jugador y td es un arreglo de una estructura creada por nosotros la cual contiene:

- socket_cliente_struct Este es el socket correspondiente a cada jugador
- rw_lock Esto es un read-write-lock que tendra cada jugador

Con esta estructura, la cual es creada fuera del ciclo en el que se cargan todos los jugadores, y luego por cada iteracion del ciclo se va guardando el socket y rw_lock correspondiente.
De esta forma, vamos creando cada thread con sus respectivos socket y rw_lock.

A continuación, mostraremos esta sección de código:

```
\*creacion de arreglo de threads y arreglo de estructuras thread_data *\

                    pthread_t threads[NUM_THREADS];
                    struct thread_data td[NUM_THREADS];

\* carga por cada iteracion de ciclo y creacion del thread *\

                    td[i].socket_cliente_struct = socketfd_cliente;
                    td[i].rw_lock = read_write_lock;
                    pthread_create(&threads[i], NULL, &atendedor_de_jugador, (void *)&td[i]);
                    i++;
```

Luego, en la función *atendedor_de_jugador* la cual recibe un thread_data lo guardamos en un puntero a thread_data llamado my_data y creamos un entero llamado socket_fd el socket que nos viene como parametro.

Luego, basandonos en el backend mono, realizamos una implementación similar con la particularidad que, en el if donde se consulta si el mensaje del jugador es una ficha, palabra o update.
En caso de ser una ficha, luego de parsear el casillero, antes de chequear la validez de la ficha utilizamos nuestro read_write_lock y realizamos la funcion *rlock*(). En caso de ser una ficha valida, realizamos un read unlock y procedemos a escribir de la siguiente manera:

```
                    my_data->rw_lock.wlock();
                    palabra_actual.push_back(ficha);
                    tablero_letras[ficha.fila][ficha.columna] = ficha.letra;
                    my_data->rw_lock.wunlock();
```

Para luego enviar la misma y terminar la jugada. En caso de que la validez de la ficha no sea correcta, dejamos de leer y procedemos a escribir para quitar fichas y asi dejar de escribir, como mostramos a continuación:

```
my_data->rw_lock.runlock();
my_data->rw_lock.wlock();
quitar_letras(palabra_actual);
my_data->rw_lock.wunlock();
```

Por consiguiente, en caso de que el mensaje sea una palabra se realiza un wlock para escribir la palabra y luego un wunlock.
Finalmente, en caso de ser update se realiza un rlock para actualizar la pantalla y se finaliza la jugada.

Fuera de este IF, se finaliza el thread creado con la funcion $pthread_exit(NULL)$.

De esta manera, queda implementado nuestro backend multithreaded como fue solicitado.

4 Bibliografía

- Cátedra de Sistemas Operativos - Clases teóricas y prácticas (2º Cuatrimestre 2015)
- The Little Book of Semaphores