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

TP2 - Pthreads

Sistemas Operativos

Primer Cuatrimestre de 2016

Apellido y Nombre	LU	E-mail
Casanova, Manuel	355/05	morbous_panda@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 basó en el pseudocódigo 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 pseudocóodigo 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 código 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 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 a la variable de condición *room_empty* para notificarle al escritor que ya no queda ningún lector y puede proceder a escribir.
Por último, se libera el mutex *readers_mutex*.

Continuando con el escritor, el pseudocódigo 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 terminarán de leer y en caso de llegar nuevos lectores deberán esperar a que el escritor finalice su ejecución.

3 Parte II: RWLockTest

3.1 Ejercicios

- **Ejercicio 2** Deberán, a su vez, implementar un test para su implementación de Read-Write Locks (RWLockTest) que involucre la creación de varios threads lectores y escritores donde cada uno de ellos trate de hacer un lock sobre un mismo recurso y se vea que no haya deadlocks ni inanición (archivo backend-multi/RWLockTest.cpp).

3.2 Resultados y Conclusiones

3.2.1 Ejercicio 2

Para mostrar esto creamos una función que crea varios threads de escritura y lectura. Cada tipo de thread llama a una función distinta e imprime por pantalla su ID y el valor de una variable, en caso de los threads de escritura estos indican el nuevo valor de la variable mientras que los de lectura imprimen el valor actual de la misma.

Para ello se utilizan las siguientes variables y funciones:

- int NUM_THREADS: Indica la cantidad de threads a crear.
- int variable: Esta es el recurso que los threads van a manejar (ya sea para leerlo o modificarlo).
- RWLock para_variable: Este lock se usa cuando un thread quiere manejar el recurso.
- int array[NUM_THREADS]: Un array de tamaño igual a cantidad de threads, Se usa para darle a cada thread un ID. La posición del array va a corresponder al thread creado.
- pthread_t threads[NUM_THREADS]: Un array de tamaño igual a cantidad de threads a crear, se usa para crear los threads.
- void *soy_lector(void *p_numero): Función que va a correr el thread que lea el recurso.
- void *soy_escritor(void *p_numero): Función a correr por los threads que modifiquen el valor del recurso.

```
for (i = 0; i < NUM_THREADS; ++i){
  if(!(i % 2)){
    pthread_create(&threads[i], NULL, soy_escritor, &array[i]);
  }else{
    pthread_create(&threads[i], NULL, soy_lector, &array[i]);
  }
}
```

Este ciclo es el encargado de crear todos los threads, si el valor de i es impar se crea un thread que va a correr la función "soy_escritor", si es par ejecuta "soy_lector".
Para crear cada thread se le pasa a la función un thread dentro del array "threads" y la función a partir de donde va a correr(en este caso una de las 2 funciones ya mencionadas) y una posición del array "array" distinta a cada uno para poder identificarlos.

```
for (int j = 0; j < NUM_THREADS; ++j){
  pthread_join(threads[j], NULL);
}
pthread_exit(NULL);
return 0;
```

Luego se hace un ciclo para esperar a que terminen todos los threads creados antes de finalizar el programa.

```
void *soy_lector(void *p_numero){
  int mi_numero = *((int *) p_numero);

  for (int j = 0; j < 5; ++j){
    para_variable.rlock();
    printf("Lector numero: %d  ", mi_numero);
    printf(" Leo valor %d\n", variable );
    para_variable.runlock();
  }
  pthread_exit(NULL);
  return NULL;
}
```

La función lector corre un ciclo 5 veces. Dentro del ciclo lo primero que hace es usar el lock mientras trabaja con la variable a usar por todos los threads. Luego imprime su ID que se le paso por parámetro a la función y el valor actual de la variable y libera el lock. Luego de hacer esto 5 veces termina el thread.

```
para_variable.wlock();
variable++;
printf("Escritor numero: %d  ", mi_numero);
printf(" cambio valor %d\n", variable );
para_variable.wunlock();
```

La función "soy_escritor" es casi igual al de la otra función salvo porque lo primero que hace al hacer el lock es incrementar en 1 el valor de la variable.

3.2.2 Conclusiones

Con el código que implementamos nos aseguramos que no pueda haber deadlock ni inanición. Vamos a pasar a explicar que solución encontramos para cada uno.

- Deadlock: Sabemos que para que haya deadlock hay 4 cosas fundamentales que tienen que cumplirse y aún así no siempre hay. Se observa en el código explicado mas arriba que solo se usa "RWLock para_variable" para proteger la sección crítica del código por lo que no hay "Hold and wait" evitando así posible deadlock.
  Ahora vamos a ver que nuestro código implementado para Read-Write Lock tampoco puede sufrir Deadlock. Igual que en el caso anterior el código implementado no puede sufrir de "Hold and wait". La única función que pareciera poder sufrir de esto es "wlock". Esta hace lock de "pthread_mutex_t turnstyle" y luego hace lock de "pthread_mutex_lock(&readers_mutex)". Si se da el caso de que puede hacer lock de ambos en el paso siguiente hace unlock de "readers_mutex" que luego combinado y usado correctamente con la función "wulock" se libera el otro mutex (turnstile) evitando el riesgo de "Hold and wait".
  En el otro caso que se puede dar, "wlock" se queda esperando el mutex "readers_mutex". Si vemos cuales son las funciones que usan el mutex "readers_mutex" vemos que estas una vez tomado el mutex pueden correr su código sin problemas finalizando y liberando dicho mutex. Esto permite que "wlock" pueda más adelante seguir con su ejecución sin problemas. Luego de analizar ambos casos y sus posibles desenlaces podemos concluir que no hay riesgo de Deadlock usando Read-Write Lock. Luego de analizar ambas implementaciones podemos confirmar que nuestro código está libre de Deadlock.
- Como vimos en el punto 1 nuestro código esta libre de inanición.

4 Parte III: Desarrollo de Backend Multithreaded

4.1 Ejercicios

- **Ejercicio 3** En segundo lugar, deberán implementar el servidor de backend multithreaded inspirándose en el código provisto y lo desarrollado en el punto anterior.

4.2 Resultados y Conclusiones

4.2.1 Ejercicio 3

En este Ejercicio, se solicitó la implementación de un backend mutithreaded, para el desarrollo del mismo, utilizamos la base del backend mono para la conexión con el servidor, el parseo de las fichas, tanto para la validez de las mismas y también el envío 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, &socketfd_cliente);
```

En donde, threads es un arreglo de thread_t y se le asigna uno a cada jugador y socketfd_cliente es el atributo de cada thread. En este ejercicio como no vamos a necesitar el atributo todos los threads usan el mismo.

A continuación, mostraremos esta sección de código:

```
\*creacion de arreglo de threads *\

                    pthread_t threads[NUM_THREADS];
```

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 parte del barco, el barco terminado, una bomba o update.
En caso de ser una parte de barco, luego de parsear el casillero, y al chequear la validez de la ficha utilizamos nuestro read_write_lock y realizamos la funcion *wlock*(). En caso de ser una ficha valida, realizamos un write lock y procedemos a escribir de la siguiente manera:

```
(*rwlocks_tablero)[ficha.fila][ficha.columna].wlock();
(*tablero_jugador)[ficha.fila][ficha.columna] = ficha.contenido;
(*rwlocks_tablero)[ficha.fila][ficha.columna].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.

Por consiguiente, en caso de que el mensaje sea una barco terminado se realiza un wlock para escribir la palabra y luego un wunlock.

El mismo se detalla a continuación:

```
for (list<Casillero>::const_iterator casillero = barco_actual.begin(); casillero != barco_actual.end
(*rwlocks_tablero)[casillero->fila][casillero->columna].wlock();
```

```
(*tablero_jugador)[casillero->fila][casillero->columna] = casillero->contenido;
(*rwlocks_tablero)[casillero->fila][casillero->columna].wunlock();
}
barco_actual.clear();
```

Luego, en caso de ser una bomba chequeamos si el casillero donde se coloco la bomba tiene un barco o una bomba, en caso de tener una bomba se envia el mensaje de que ya estaba golpeado como en el backend mono. Si había un barco, hacemos un wlock para escribir en el casillero bomba y liberamos el lock.

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

5 Bibliografía

- Cátedra de Sistemas Operativos - Clases teóricas y prácticas (1º Cuatrimestre 2016)
- The Little Book of Semaphores