Sistemas Operativos

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

Trabajo Práctico Número 2

`pthreads` - "SOScrabel"

Integrante	LU	Correo electrónico
Ciruelos Rodríguez, Gonzalo	063/14	`gonzalo.ciruelos@gmail.com`
Maddonni, Axel Ezequiel	200/14	`axel.maddonni@gmail.com`
Thibeault, Gabriel	114/13	`gabriel.eric.thibeault@gmail.com`

Índice

1. Read-Write Lock

Nuestra implementación de RWLocks funciona de manera similar a la siguiente implementación propuesta en el libro *The Little Book of Semaphores* [1], que utiliza semáforos para solucionar el problema de lectores y escritores.

Listing 1: No-starve readers-writers initialization

```
readSwitch = Lightswitch()
roomEmpty = Semaphore(1)
turnstile = Semaphore(1)
```

Listing 2: No-starve writer solution

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

roomEmpty.signal()
```

Listing 3: No-starve reader solution

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

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

Esta solución utiliza un semáforo denominado `turnstile` o **molinete**, que permite a los lectores avanzar a su zona crítica hasta que un escritor haga lock del molinete. En ese caso, fuerza a los lectores entrantes a trabarse en el molinete y espera a que los que ya habían pasado el molinete terminen de leer, utilizando el semáforo `roomEmpty` para saber cuando no haya más lectores leyendo. Entonces, cuando el último lector deje la zona crítica, el escritor podrá escribir. Esto garantiza que además no pueda haber más de un escritor escribiendo, ya que también quedarán trabados en el molinete mientras haya alguien escribiendo. Este método evita inanición de escritores y lectores, ya que por ejemplo, en otra implementación podría suceder que un escritor espere indefinidamente mientras sigan llegando lectores o viceversa.

Para adaptar esta implementación usando variables de condición, utilizamos una variable de condición a la que llamamos `molinete`, asociada a un mutex `m`. Para reemplazar el mutex turnstile de la implementación anterior, utilizamos una variable booleana `mol` que

[1] http://www.greenteapress.com/semaphores/downey05semaphores.pdf

indica si el molinete está liberado o no, y para reemplazar el semáforo roomEmpty utilizamos un entero `lectores` que lleva la cuenta de la cantidad de lectores que se encuentran leyendo.

- El *rlock* consiste en hacer un lock del mutex para asegurar exclusividad de las variables y esperar a que el molinete se encuentre liberado (`mol` == True). Si está liberado se aumenta la cantidad de lectores en uno y se procede a hacer unlock del mutex para leer.
- El *runlock* consiste en hacer lock del mutex, disminuir la cantidad de lectores en uno y realizar un broadcast antes de hacer unlock del mutex, para despertar a los threads bloqueados en el molinete (es necesario realizar broadcast ya que los signal no son acumulables, a diferencia de los semáforos, y puede haber más de un lector esperando leer).
- El *wlock* consiste en hacer lock del mutex, esperar que el molinete se encuentre liberado, y una vez pasado el molinete bloquearlo (seteando `mol` = False) para evitar que pasen nuevos lectores o más de un escritor al mismo tiempo. Antes de escribir, espera que terminen de leer los lectores que se encontraban leyendo antes de trabar el molinete (es decir, `lectores` == 0), y luego hace unlock del mutex para escribir su sección crítica.
- El *wunlock* consiste en hacer lock del mutex, liberar el molinete seteando `mol` en True, y realizar el broadcast correspondiente antes de hacer unlock del mutex, para despertar a los demás threads.

La implementación no sufre de inanición ni deadlocks, como explicamos antes, dado que existen los molinetes. Los molinetes permiten que los escritores y lectores se vayan turnando, además de permitir, en el caso de los lectores, que varios entren a la zona crítica al mismo tiempo. El concepto de los turnos es el más importante para este problema, dado que pensarlo de esta manera clarifica en gran medida la solución.

1.1. Tests

Los tests implementados para corroborar el funcionamiento de los RWLocks consisten en la creación de una cierta cantidad de threads asociados a una función de lectura o escritura de una variable global. Cada thread luego imprime en pantalla, en caso de que esté escribiendo, el nuevo valor asignado a la variable global, o en caso de que esté leyendo, el valor leído. Para poder identificarlos, los threads escritores escriben en la variable su número de tid asociado que va de 0 a CANT_THREADS-1.

El programa `rwlocktest` toma por parámetro un entero para diferenciar tres tipos de test:

- `./rwlocktest 0` ejecuta un test con CANT_THREADS-1 lectores y 1 escritor, donde el orden en que se crea el thread escritor está dado pseudoaleatoriamente utilizando las funciones de C `rand()` y `srand()`.

- `./rwlocktest 1` ejecuta un test con CANT_THREADS-1 escritores y 1 lector, donde el orden en que se crea el thread lector está dado pseudoaleatoriamente utilizando las funciones de C `rand()` y `srand()`.
- `./rwlocktest 2` ejecuta un test con una cantidad aleatoria de escritores y lectores, utilizando `rand()` y `srand()` para determinar si el thread creado será lector o escritor.

Los tests 0 y 1 fueron implementados para corroborar que no se produzca inanición de escritores y/o lectores respectivamente.

El valor de CANT_THREADS se encuentra definido por default en 40, pero puede modificarse fácilmente en RWLockTest.cpp.

2. Backend Multiplayer

Para realizar la implementación del backend multithreaded se utilizó como base el código del monoplayer y el Read-Write Lock implementado por nosotros.

El primer problema que aparece es al atender las conexiones. Para eso, se debió delimitar un máximo de jugadores que pueden estar jugando al juego en simultaneo. Para eso, vamos a escuchar como máximo esa cantidad de conexiones.

Para el tablero de letras usamos un rwlock para cada casillero, permitiendo así que dos usuarios escriban simultáneamente 2 casilleros distintos. Por eso, debimos utilizar una matriz de rwlocks, en la que cada posición de la matriz se corresponde con la misma posición del tablero de letras.

Sin embargo, para el tablero de palabras la situación es distinta, y tomamos la decisión de tener un único lock para todo el tablero, dado que si tenemos un lock por casillero, pueden pasar cosas como que un jugador haga update mientras otro esta escribiendo una palabra y que al que hizo update le llegue solo media palabra.

Haciendo lock del tablero entero nos aseguramos que cuando un jugador escribe una palabra, estamos seguros de que lo va a terminar de hacer antes de que otro jugador lea los contenidos del tablero de palabras. Debido a esto, además debimos tener en cuenta que el lock se debe hacer durante toda la escritura de una palabra, es decir, que la escritura de una palabra entera debe verse como *atómica* desde afuera.

Otra modificación –muy menor– que hubo que realizar, fue la de la finalización del atendedor. Ahora, como cada jugador se atiende con threads distintos, la finalización de un thread debe tratarse de manera particular para cada thread, por lo que cuando un jugador cierra su conexión o termina su partida, se debe llamar a `pthread_exit`, en vez de a `exit` simplemente, como está en la versión monojugador.