DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

Pthreads

Sistemas Operativos

Segundo Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Confalonieri, Gisela Belén	511/11	gise_5291@yahoo.com.ar
Mignanelli, Alejandro Rubén	609/11	minga_titere@hotmail.com
Sabarros, Ian	661/11	iansden@live.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. Read Write Lock **3**
1.1. Implementación 3
1.2. Testing 3

2. Backend Multithreaded **5**
2.1. Introduccion 5
2.2. Implementacion 5

1. Read Write Lock

Se implementó un *Read-Write Lock* **libre de inanición** utilizando sólo Variables de Condición POSIX. La interfaz del lock es la siguiente:

- **Constructor:** Inicializa el lock para su utilización.
- **rlock():** Pide el lock para lectura.
- **wlock():** Pide el lock para escritura.
- **runlock():** Libera el lock de lectura.
- **wunlock():** Libera el lock de escritura.

1.1. Implementación

Este *Read-Write Lock* permite lecturas concurrentes y asegura que, mientras se realiza una escritura, ningún otro thread se encuentra escribiendo ó leyendo al mismo tiempo.

Para evitar inanición de escritores, este lock les da cierta prioridad. Es decir, al momento en que un escritor solicita el lock, se espera a que los threads que están leyendo (de haberlos) terminen de hacerlo y se otorga el lock al escritor, dejando en espera a los lectores que hayan llegado después que él. En caso de llegar un escritor mientras se está realizando una escritura, éste obtendrá el lock al terminarse dicha escritura, aún cuando haya lectores en espera.

Sin embargo, si llegasen sucesivos escritores podrían provocar inanición de lectores. Así que para evitar esto, luego de cierta cantidad de escrituras consecutivas, se otorgará el lock a los lectores que estén esperando en ese momento, y luego se continuará con los escritores restantes (de existir), operando de la misma manera.

Para implementar este *Read-Write Lock* se mantienen ciertas variables que permitirán la sincronización adecuada. Éstas incluyen 4 enteros, 2 que representarán la cantidad total de lectores y escritores (esperando y operando) y 2 que representarán la cantidad de lectores y escritores que efectivamente están operando. Se utiliza también un mutex y dos variables de condición: una para quienes esperan leer y una para quienes esperan escribir. Todos los enteros se inicializan en 0 (Figura 1).

```
1  pthread_mutex mutex(1)
2  int lectores = 0
3  int escritores = 0
4  int leyendo = 0
5  int escribiendo = 0
6  int contador = 0
7  int contador_auxiliar = 0
8  pthread_cond cond_esperoleer
9  pthread_cond cond_esperoescribir
```

Figura 1: Pseudocódigo variables

Las Figuras 2, 3, 4 y 5 muestran el funcionamiento interno de RLOCK(), WLOCK(), RUNLOCK() y WUNLOCK() respectivamente. La cantidad de escrituras consecutivas está indicada por la constante LIMITE.

1.2. Testing

Para testear esta implementación de *Read-Write Lock* se corrieron 3 sets de pruebas. Para todos ellos se consideró el LIMITE = 1 y se crearon sucesivos threads de lectura y escritura para evaluar el funcionamiento. Entre ellos se comparte una variable entera que comienza en 0, los lectores muestran por pantalla el valor de esta variable al momento de realizar la lectura y los escritores la incrementan al momento de

```
RLOCK()
1  mutex.lock
2  lectores++
3  while (escritores > 0 ∧ contador < LIMITE) ∨ escribiendo > 0
4        cond_wait(cond_esperoleer,mutex)
5  leyendo++
6  contador_auxiliar- -
7  if contador_auxiliar == 0
8        contador_auxiliar = 0
9  mutex.unlock
```

Figura 2: Pseudocódigo rlock

```
WLOCK()
1  mutex.lock
2  escritores++
3  while leyendo > 0 ∨ escribiendo > 0 ∨ (lectores > 0 ∧ contador >= LIMITE)
4        cond_wait(cond_esperoescribir,mutex)
5  escribiendo++
6  mutex.unlock
```

Figura 3: Pseudocódigo wlock

```
RUNLOCK()
1  mutex.lock
2  lectores- -
3  leyendo- -
4  if leyendo == 0
5        cond_signal(cond_esperoescribir)
6  mutex.unlock
```

Figura 4: Pseudocódigo runlock

```
WUNLOCK()
1  mutex.lock
2  escribiendo- -
3  contador++
4  if (escritores == 0 ∨ contador >= 0) ∧ lectores > 0
5        contador_auxiliar = lectores
6        cond_broadcast(cond_esperoleer)
7  else cond_signal(cond_esperoescribir)
8  mutex.unlock
```

Figura 5: Pseudocódigo wunlock

realizar la escritura. Para notar la concurrencia de lectores y la exclusividad de escritores, se esperan 100

ns en medio de la operación.

Los sets de prueba son los siguientes:

1. Se crearon 41 threads, 40 de lectura y 1 de escritura. El escritor fue creado luego de 20 lectores, para observar que efectivamente tiene prioridad y no muere de inanición, que las lecturas son concurrentes, y que la escritura es exclusiva. Se espera que el escritor no sea el último thread en correr, ya que esto significaría que no tuvo prioridad y en un caso real significaría inanición. A su vez, los lectores previos a la escritura deberán mostrar un 0 y los posteriores un 1, y ningún lector debería acceder a la variable durante la escritura.

2. Se crearon 41 threads, 40 de escritura y 1 de lectura. El lector fue creado luego de 20 escritores, para observar que efectivamente se cumplía el LIMITE y no moría de inanición y que las escrituras son exclusivas. Se espera que el lector no sea el último thread en correr, ya que esto significaría que no se respetó el LIMITE y en un caso real significaría inanición. A su vez, las escrituras deberán ser consecutivas y excluyentes, y la lectura no debería ocurrir durante una escritura.

3. Se crearon 41 threads y se les asignó el rol de escritor ó lector de manera aleatoria, para chequear un funcionamiento más general. Se espera que las lecturas sean excluyentes y las lecturas sean concurrentes.

Se realizaron 10 ejecuciones de cada test, y en todos los casos el comportamiento fue el esperado.

Para ejecutar estos tests: `./rwlocktest <i>` (sin <>) donde i es 1,2 ó 3 para indicar el test correspondiente.

Queda pendiente la prueba con distintos valores de LIMITE[1].

2. Backend Multithreaded

2.1. Introduccion

Debemos permitir que los jugadores se conecten en simultáneo al Scrabble, para eso utilizaremos distintos threads para atender a cada cliente que se conecte al Backend.

2.2. Implementacion

Ante el problema de implementar varios threads que atiendan a los clientes en simultáneo decidimos utilizar un tablero de palabras que es compartida por todos ellos, los cuales serán sincronizados utilizando locks.

Al tener sólo un tablero compartido debemos protegerlo de escrituras y lecturas en simultáneo. Utilizamos el modelo de sincronización clásico de escritores con acceso exclusivo y lectores con acceso compartido.

Nuestra implementación permite lecturas en simultáneo al tablero de palabras y elimina los tableros de letras utilizados en la implementación original.

Los principales cambios que se hicieron al código original son:

- Permitir que el servidor reciba múltiples conexiones, una por cada jugador. La cantidad de jugadores está seteada en una constante.
- Creación de Thread al aceptar la conexión del cliente, inicializado en la función ATENDEDOR_DE_JUGADOR para soportar la atención de múltiples clientes.
- Pedidos de ReadLock y liberación del mismo en caso de que el cliente solicite un *update*, ya que esta operación necesita una lectura del tablero.

[1] Para probar esto se debe modificar el valor de la constante LIMITE en `RWLock.cpp`

- Pedido de ReadLock y liberación del mismo en la lectura del Tablero de Palabras cuando el cliente coloca una *letra* en su palabra actual, ya que se debe chequear que la letra esté en una posición adyacente a las demás de la misma palabra, y que no está siendo colocada en una posición que ya contiene una letra en el tablero.
- Pedido de WriteLock y liberación del mismo cuando el cliente envía la operación *palabra* al servidor, ya que esta operación necesita modificar el tablero, colocando las letras correspondientes.
- Chequeo al insertar una *letra* o enviar una *palabra* si alguna de las casillas correspondientes a las letras de la palabra actual fue ocupada luego de la escritura de otro jugador, para evitar la continuación o escritura de palabras que ya se tornaron inválidas.