DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II - Threads

Scrabble Hasobro

Sistemas Operativos
Segundo Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Arístides Catalano	279/10	dilbert_cata@hotmail.com
Laura Muiño	399/11	mmuino@dc.uba.ar
Jorge Quintana	344/11	jorge.quintana.81@gmail.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

Contents

1 Servidor Multi

Para crear un servidor que soporte mútiples jugadores, se adaptó el backend-mono provisto por la cátedra. El primer cambio es permitir que atienda mas de una conexión. Para esto se utilizan *threads*. La idea es que cada vez que un cliente se conecte al servidor, se cree un thread nuevo que va a atenderlo hasta que termine de jugar, y luego de eso terminará exitosamente. Otro problema a solucionar es la sincronización entre threads, ya que ahora van a estar compartiendo el tablero, y "compitiendo" por cada casilla para poder leer y escribir.Por otro lado, se desea no perder demasiada concurrencia en el juego, es decir, que deberian poder modificar simultáneamente dos posiciones distintas de las tableros simultáneamente, sin tener problemas.

A continuación se muestran brevemente los cambios realizados. Las variables globales utilizadas son:

```
int socket_servidor = -1;

//matriz de locks para proteger cada casilla del tablero_palabras
vector<vector<RWLock> > lock_casilla_posta;

//matriz de locks para proteger cada casilla del tablero_letras
vector<vector<RWLock> > lock_casilla_temp;

// tiene letras que aun no son palabras validas
vector<vector<char> > tablero_letras;
// solamente tiene las palabras validas
vector<vector<char> > tablero_palabras;

unsigned int ancho = -1;
unsigned int alto = -1;
```

Estas se inicializan en la función main, una vez que se obtienen los valores de alto, y ancho del tablero.

Aqui un fragmento de código de backend-multi.cpp:

```
while (true) {
    if ((socketfd_cliente = accept(socket_servidor, (struct sockaddr*) &remoto,
                                   (socklen_t*) &socket_size)) == -1)
        cerr << "Error al aceptar conexion" << endl;
    else {
                int* nuevo_socket = new int;
                *nuevo_socket = socketfd_cliente;
                pthread_t thread_id;

//Aca se crea un nuevo thread que va a atender la conexion nueva(el jugador nuevo)
                pthread_create(&thread_id, NULL, atendedor_de_jugador,
                                   (void*) nuevo_socket);
    }
}
```

La función *pthread_create* se encarga de crear un thread nuevo, el cual va empezar su ejecución en la función *atendedor_de_jugador*, y se le pasa el argumento *nuevo_socket*, que contiene el file descriptor del socket del nuevo cliente.

El thread principal, que es el servidor, se va a quedar escuchando conexiones entrantes de nuevos jugadores y se va a encargar de crear los threads.
Una vez creado el thread, comienza a ejecutar en la función *atendedor_de_jugador*. Lo primero que ejecuta es la función $pthread\_detach(pthread\_self())$, Esto hace que cuando el thread termine y haga exit, devuelva todos los recursos ultilizados, sin necesidad de que el thread principal haga join con el que

esta terminando. Luego se obtiene el numero de socket del cliente que esta atendiendo ese thread para poder realizar el envío de mensajes con el servidor. Finalmente, cuando el cliente elija terminar el juego, se llamará a la función *terminar_servidor_de_jugador*, la cual cierra el socket del cliente terminado, libera los casilleros del *tablero_letras* que pudiera estar utilizando, y llama a la función *pthread_exit*, que termina el thread de manera exitosa.

NOTA: cuando el servidor termina, tambien cierra su socket y hace exit. Sería mas adecuado que cierre los sockets de todos los clientes que estan corriendo y ver que terminen exitosamente. Esto no afecta el funcionamiento del juego.

2 Sincronización

Para poder permitir la concurrencia entre jugadores, se utilizó la clase RWLock, que permite hacer lecturas simultáneas y escrituras exclusivas. La idea de crear una matriz de estas estructuras, es protejer cada casilla por separado, permitiendo asi, modificar o leer, mas de una a la vez por distintos threads. Entonces, cada vez que potencialmente se vaya a modificar algun casillero de algun tablero, se llama a la funcion *wlock*, que bloquea ese casillero, sin permitir que ningun otro thread pueda ni siquiera leerlo. La funcion *rlock* si va a permitir lecturas concurrentes.

En la función *atendedor_de_jugador*, se recibe la ficha, y la poscicion donde se quiere ubicar a la misma. A partir de ahí comienza a chequear, si la posicón es válida, si está libre, etc. Antes de realizar la consulta *es_ficha_valida_en_palabra* se bloquea dicho casillero con *wlock*, ya que de ser válida, se podrá escribir en ese lugar, y recién ahí se podrá desbloquear para garantizar el acceso exclusivo a esa pos.

Dentro de la función $es\_ficha\_valida\_en\_palabra$, se chequea que la poscición en la que se desa ubicar la letra sea válida, es decir, que esté dentro del tablero, que no está ocupada, que si no es la primera letra de la *palabra_actual* sea solo vertical o solo horizontal, y que, si hay espacio entre la letra anterior y la que estoy poniendo, ese espacio esta ocupado por otras letras en el *tablero_palabras*. Es decir, esta usando letras de otra palabra ya escrita en el tablero. En cualquier caso que no cumpla las condiciones, se debe desbloquear esa posicion del *tablero_letras*, osea hacer *wunlock* en la matriz $lock\_casilla\_temp[f][c]$. Si se cumplieran todas la condiciones, entonces la letra puede escribirse en esa pos, por lo que se libera el lock, despues de modificarla. Cuando se realizan los chequeos de los casilleros del *tablero_palabras*, se realizan *rlock*, ya que como no se vana modficar, esto permita que varios threads puedan leer ese valor. Una vez que se termina de acceder, se hace *runlock*, para desbloquearlo. Básicamente se bloquean las casillas cada vez se quiere hacer una escritura o una lectura. En la funcione *enviar_tablero*, se hace *rlock* sobre lock_casilla_temp ya que varios jugadores pueden estar actualizando el tablero a la vez. En *quitar_letras*, se modifica el *tablero_letras* entoces se hace *wlock* sobre lock_casilla_temp y en *atendedor_de_jugador* cuando se recibe el commando MSG_PALABRA, copia la *palabra_actual* en el *tablero_palabras*, para esto hacemos *wrlock* sobre lock_casilla_posta. Luego de hacer la lectura o escritura, de libera la casilla haciendo el unlock correspondiente.

3 Implementación RWLock

Se pidió realizar la implementación de la clase RWLock, que provee funciones lectura y escritura bloqueantes. Para hacerlo, se utilizaron Varibles de Condición POSIX. para poder garantizar con ellas el acceso exclusivo y la no inanición. Se pasará a describir brevemente cada función.

- **RWLock:** Se encarga de inicializar las variables globales que van a utilizar todas la funciones de la clase.

```
cant_lectores = 0;
hay_escribiendo = false;
hay_escritor_esperando = false;

pthread_mutex_init(&mtx_RWL, NULL);
pthread_cond_init(&barrera_lectores, NULL);
pthread_cond_init(&primera_barrera_lectores, NULL);
pthread_cond_init(&sem_escritores, NULL);
```

- **rlock:** Cuando un thread llame a la función *rlock*, lo primero que va a hacer es

```
pthread_mutex_lock(&mtx_RWL);

while(hay_escritor_esperando){
        pthread_cond_wait(&primera_barrera_lectores, &mtx_RWL);
}
```

Toma el mutex, y se queda esperando en la *primera_barrera_lectores*, si es que hay algún escritor esperando. Esto es para que se respete le orden de llegada de los pedidos de lectura y escritura. Una vez que el thread que estaba esperando para escribir logra obtener el lock, manda un signal a todos los lectores que estaba en la *primera_barrera_lectores*. Luego los lectores deben chequear si hay alguien escribiendo en ese momento.

```
while(hay_escribiendo){
        pthread_cond_wait(&barrera_lectores, &mtx_RWL);
}
```

Si habia alguno el thread hace wait, sino no entra en el *while*, y aumenta en uno la *cant_lectores*. Finalmente desbloquea el mutex.

- **wlock:** Cuando un thread quiere escribir, va a llamar a esta función.

```
pthread_mutex_lock(&mtx_RWL);

while(cant_lectores > 0 || hay_escribiendo){
        hay_escritor_esperando = true;  //YO PASO A SER UN ESCRITOR ESPERANDO
        pthread_cond_wait(&sem_escritores, &mtx_RWL);
}
```

Lo que hace es esperar a que no haya nadie leyendo ni escribiendo para poder tener el acceso exclusivo. Una vez que logra obetenerlo, actualiza las variables, suelta el mutex y hace broadcast a todos los lectores que estaba esperando en la *primera_barrera_lectores*.

```
hay_escritor_esperando = false;
hay_escribiendo = true;

pthread_mutex_unlock(&mtx_RWL);
pthread_cond_broadcast(&primera_barrera_lectores);
```

- **runlock:** Resta uno en la cant_lectores y si es igual a 0, hace un signal al escritor que este esperando en &sem_escritores.
- **wunlock:** Pone $hay\_escribiendo = false$ y hace broadcast a todos los lectores esperando en $barrera\_lectores$ y hace signal a &sem_escritores.

Cada vez que se modifica una variable global, se pide el mutex.

4 Test RWLock

Para comprobar el correcto funcionamiento de las funciones de lectura/escritura bloqueantes para threads, se realizó un programa con el obejtivo de simular varios pedidos de lectura y escritura en forma concurrente sobre un mismo recurso. La idea era ver que la implementación se enctontrara libre de inanición y que ningún thread pueda compartir la sección crítica. Entonces, la función *main* se encarga de la creación de todos los threads lectores y escritores. A su vez cada thread creado va a esperar en una barrera hasta que terminen de crearse todos los restantes, esto es para evitar que bien se creen ya pida el acceso exclusivo a la variable. Una vez creados, el thread principal hace broadcast de esa barrera para que pueden empezar a disputarse el control sobre RWlock. Luego de esto el thread principal se queda esperando a que todos los threads terminen, esto se hace mediante la función *pthread_join*.
Cada escritor va a, una vez que obtenga el acceso exclusivo, incrementar en uno la variable protegida. y va a imprimir por pantalla que realizó efectivamente el cambio, y su *thread_id*.
Cada lector va a impirmir el contenido de esa variable, y también su *thread_id*.
Para corroborar el comportamiento, se espera que:

- la cantidad de lectores y escritores que se pasen por parámetros sean igual a la cantidad de threads que escribieron y leyeron la variable.
- que no haya mas de un thread en la sección crítica, es decir, o se modifica o se lee y una vez que se modifica el valor es uno mas que el anterior.

Se realizó un ejemplo pequeño, creando 20 lectores y 5 escritores. El resultado obtenido es el siguiente:

```
ya estan todos creados

mi tid---->0 Cambio valor
mi tid---->1 Cambio valor
mi tid---->5 leo valor:2
mi tid---->6 leo valor:2
mi tid---->3 Cambio valor
mi tid---->13 leo valor:3
mi tid---->8 leo valor:3
mi tid---->15 leo valor:3
mi tid---->7 leo valor:3
mi tid---->16 leo valor:3
mi tid---->17 leo valor:3
mi tid---->18 leo valor:3
mi tid---->24 leo valor:3
mi tid---->14 leo valor:3
mi tid---->9 leo valor:3
mi tid---->11 leo valor:3
mi tid---->10 leo valor:3
mi tid---->12 leo valor:3
mi tid---->2 Cambio valor
mi tid---->19 leo valor:4
mi tid---->21 leo valor:4
mi tid---->22 leo valor:4
mi tid---->20 leo valor:4
mi tid---->23 leo valor:4
mi tid---->4 Cambio valor
```

Vemos que la cantidad de threads creados es la esperada y que no hay mas de uno en la sección crítica. Se incrementa en uno cada vez que se ejecuta un escritor, cuando le toca a un lector imprime por pantalla el valor correctamente. También se realizaron muchas iteraciones del test, con 10.000 lectores y 4.000 escritores con el fin de ver si existía inanición.