Sistemas Operativos

Trabajo Práctico 2

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

Segundo Cuatrimestre de 2012

Grupo AnitaLavaLaTina

Apellido y Nombre	LU	e-mail
Sebastian Garbi	***179/05***	***garbyseba@gmail.com***
Alejandro F Duran	***286/05***	***alejandrofduran@gmail.com***
Maximiliano Alvarez	***181/04***	***maxi1985.798@gmail.com***

Reservado para la cátedra

Instancia	**Docente que corrigió**	**Calificación**
Primera Entrega		
Recuperatorio		

Introducción

El presente trabajo práctico consta de 2 secciones: una en la que haremos una interface del patrón de semáforos lectores-escritores, y otra en la que usaremos esta interfaz para realizar el servidor de backend multithreading del juego de Scrabble.
En la primera sección explicaremos cada aspecto de la interfaz mediante pseudocódigos y además mencionaremos como correr un pequeño set de pruebas que se ha adjuntado al trabajo práctico.
En la segunda sección mostraremos el pseudocódigo de las decisiones que toma el servidor de backend frente a los mensajes que les envían los clientes

Desarrollo

Ejercicio 1:

Para este ejercicio vamos a escribir un pseudocódigo de muy alto nivel para explicar el funcionamiento de la interfaz que realizamos.

Primero describamos las estructuras que utilizamos para nuestra interfaz de lectores-escritores:

1. int lectores_corriendo: este entero es para contar la cantidad de lectores que han entrado en la seccion critica y todavia no salieron de ella.
2. int escritores_corriendo: este entero es para contar la cantidad de escritores que entraron en la seccion critica y todavia no salieron de ella (notar que siempre una de estas dos ultimas variables van a ser cero).
3. cola<Peticion> cola_peticiones: esta cola servira para encolar las peticiones de lectura y escritura.
4. Peticion = <mutex *m, int tipo_peticion>: esta tupla representa la peticion. En donde m es un puntero a un mutex para bloquearse y tipo_peticion es un entero que representa el tipo de peticion que se va a encolar (0 peticion de lectura, 1 peticion de escritura).
5. mutex mutex_cola_peticiones, mutex mutex_lectores_corriendo, mutex mutex_escritores_corriendo: estos 3 mutexes son para excluir el uso de las primeras 3 estructuras respectivamente.

Ahora describamos el funcionamiento de cada una de las funciones que nos provee nuestra interfaz

Pseudocódigo Función rlock(): Esta debe ser llamada cada vez que se va a entrar a la sección crítica para leer, si el lector-escritor está bloqueado debido a que hay alguien escribiendo o leyendo, entonces el thread que llame a esta función se bloqueará hasta que el lector-escritor esté disponible.
Una vez que el thread se desbloquea otros threads que hayan sido bloqueados por esta función pueden ser desbloqueados también.

```
rlock():
        pedir_exclusividad_sobre_recursos()

        if (!cola_peticiones.empty())
        {
                crear_peticion(p)
                cola_peticiones.encolar(p)
                devolver_exclusividad_sobre_recursos()
                wait(p.m)
        }
        else
```

```
{
        if (hay_alguien_ejecutando())
        {
                crear_peticion(p)
                cola_peticiones.encolar(p)
                devolver_exclusividad_sobre_recursos()
                wait(p.m)
        }
        else
        {
                lectores_corriendo++;
                devolver_exclusividad_sobre_recursos()
        }
}
```

Pseudocódigo Función wlock(): Esta función es llamada para pedir exclusividad para escribir, si el lector-escritor está bloqueado debido a que hay alguien escribiendo o leyendo, entonces el thread que llame a esta función se bloquea hasta que el recurso esté disponible. Una vez que se desbloquea solo el thread que llamo esta función estará en posesión del lector-escritor.

```
wlock():
        pedir_exclusividad_sobre_recursos()

        if (!cola_peticiones.empty())
        {
                crear_peticion(p)
                cola_peticiones.encolar(p)
                devolver_exclusividad_sobre_recursos()
                wait(p.m)
        }
        else
        {
                if (hay_alguien_ejecutando())
                {
                        crear_peticion(p)
                        cola_peticiones.encolar(p)
                        devolver_exclusividad_sobre_recursos()
                        wait(p.m)
                }
                else
                {
                        escritores_corriendo++;
                        devolver_exclusividad_sobre_recursos()
                }
        }
```

Pseudocódigo Función runlock(): Esta función se encarga de que un lector libere el lector-escritor. En el caso de que ya no quede ningún lector leyendo, esta función habilita a que un escritor o varios lectores tomen posesión del lector-escritor.

```
runlock():
        pedir_exclusividad_sobre_recursos()

        if (lectores_corriendo != 0)
                lectores_corriendo --

        if (!cola_peticiones.empty())
        {
                desencolar_hasta_la_primera_escritura()
                devolver_exclusividad_sobre_recursos()
        }
        else
        {
                devolver_exclusividad_sobre_recursos()
        }
```

Pseudocódigo Función wunlock(): Esta función se encarga de que un escritor libere el lector-escritor. Luego de esta función varios lectores o otro escritor pueden tomar posesión del lector-escritor.

```
wunlock():
        pedir_exclusividad_sobre_recursos()

        if (escritores_corriendo != 0)
                escritores_corriendo --

        if (!cola_peticiones.empty())
        {
                desencolar_hasta_la_primera_escritura()
                devolver_exclusividad_sobre_recursos()
        }
        else
        {
                devolver_exclusividad_sobre_recursos()
        }
```

Aclaraciones de funciones auxiliares:

- devolver_exclusividad_sobre_recursos() y pedir_exclusividad_sobre_recursos() son funciones que utilizan los mutexes: mutex_cola_peticiones, mutex_lectores_corriendo y mutex_escritores_corriendo para pedir o devolver la exclusión de los recursos necesarios para cada función.

- desencolar_hasta_la_primera_escritura() puede realizar una de las siguientes acciones:
    - desencolar los primeros k lectores hasta el primer escritor o hasta el final de la cola.
    - desencolar un escritor si se encuentra al principio de la cola.

Set de prueba:

En el presente trabajo práctico se ha adjuntado una carpeta ./backend-mono/rw para realizar pruebas de esta interfaz. Dentro de esta carpeta se encuentra un archivo main.c, un makefile y una copia exacta de los archivos RWLock.cpp y RWLock.h para compilar y generar un binario rw que lanzará 80 threads con lectores y escritores sobre una variable global. También, en esta carpeta, se encuentra un script hecho en bash para correr el binario rw unas 50 veces.

Ejercicio 2:

En el presente ejercicio hemos implementado el servidor de backend multithread utilizando la interfaz de lectores-escritores del ejercicio anterior.
Para que cada cliente se pueda conectar simultáneamente, se ha creado un thread para cada conexión que acepte el servidor de backend. Luego cada uno de estos threads tendrá un socket único para poder enviar y recibir mensajes hacia los clientes.
Veremos un pseudocódigo que ejecuta cada thread frente a cada petición de los clientes del servidor de backend.

Primero veamos algunas variables globales:
RWLock** rw_tablero_letras;
RWLock rw_tablero_palabras;
vector<vector<char> > tablero_letras;
vector<vector<char> > tablero_palabras;
Donde:

- rw_tablero_letras es una matriz de lectores escritores para cada uno de los casilleros del juego.
- rw_tablero_palabras es una variable global de lectores-escritores para bloquear toda la matriz de palabras.
- tablero_letras es la matriz donde tiene las letras que aún no son palabras.
- tablero_palabras es la matriz donde tiene las letras que son palabras.

Ahora veamos el pseudocódigo de cada thread:

```
while(true)
{
	mensaje = recibir_mensaje(s);
	if (mensaje == MSG_LETRA)
	{
		Letra l = dame_letra(s);
		rw_tablero_letras[l.fila][l.columna].wlock();
		rw_tablero_palabras .rlock();
		if (es_ficha_valida())
		{
			asignar_letra_al_tablero();
			rw_tablero_palabras .runlock();
			rw_tablero_letras[l.fila][l.columna].wunlock();
		}
		else
		{
			rw_tablero_palabras .runlock();
			rw_tablero_letras[l.fila][l.columna].wunlock();
			quitar_letras_jugador();
```

```
                }

        }
        else if (mensaje == MSG_PALABRA)
        {
                rw_tablero_palabras .wlock();
                asignar_palabra();
                rw_tablero_palabras .wunlock();
        }
        else if (mensaje == MSG_UPDATE)
        {
                rw_tablero_palabras.rlock();
                if (enviar_tablero(s) == ERROR)
                {
                        rw_tablero_palabras.runlock();
                        terminar_servidor_jugador(s);
                }
                rw_tablero_palabras.runlock();
        }
        else if (mensaje == MSG_INVALID)
        {
                continue;
        }
}
```

Donde:

- s es el socket único para cada cliente.
- las funciones es_ficha_valida() y asignar_letra_al_tablero() hacen uso de las variables globales tablero_letras (para escribir) y tablero_palabras (para leer). La primera pregunta si es correcto poder asignar esa letra al tablero de letras, y la segunda asigna efectivamente en el tablero de letras.
- asignar_palabra() hace uso de la matriz tablero_palabras (escribe las palabras en esta). Realiza la escritura sobre el tablero de palabras para que efectivamente quede marcado en el tablero que palabra formó el jugador.
- enviar_tablero() hace uso de la matriz tablero_palabras (únicamente lee de esta).

Debemos aclarar que hay funciones como enviar_tablero() que lo que hacen es enviar datos a través del socket hacia los clientes. En este tipo de funciones se tiene que hacer la comprobación de error en caso de que se sufra de algún tipo de desconexión o algo similar. Ante cualquier error lo que hacemos es retirar las palabras no formadas del jugador y dejar las palabras formadas para que los demás jugadores las continúen utilizando. Toda esta lógica la tenemos implementada en terminar_servidor_jugador().
A continuación mostraremos un pseudocodigo de la funcion terminar_servidor_jugador():

```
terminar_servidor_jugador(socket s):
        close(s);
        quitar_letras_jugador();
        exit();
```

Como podemos ver en el pseudocódigo de la función terminar_servidor_jugador() llamamos a la funcion quitar_letras_jugador(). Esta función retira una a una las letras del jugador que aún no son palabras. Entonces debemos solicitar, en rw_tablero_letras, el casillero correspondiente cada vez que retiramos una letra.

Bibliografía

- Tutorial del LLNL. https://computing.llnl.gov/tutorials/pthreads/
- David R. Butehof, Programming with POSIX threads. Addison-Wesley Professional Computing series
- IEEE Online Standards: POSIX. http://standards.ieee.org/catalog/olis/arch_posix.html http://www.unix.org/version3/ieee_std.html
- Edward A. Lee., The Problem with Threads. Technical report, EECS Dept., University of California, Berkeley http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf