Sistemas Operativos
TP1

4 de noviembre de 2015

Integrante	LU	Correo electrónico
Martin Baigorria	575/14	martinbaigorria@gmail.com
Federico Beuter	827/13	federicobeuter@gmail.com
Mauro Cherubini	835/13	cheru.mf@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Read-Write Lock

Un *Read-Write Lock* es una primitiva de sincronización diseñada con el fin de proveer bajo paralelismo estabilidad a las estructuras de datos, motivada especialmente por los efectos de la concurrencia. Su objetivo consiste en facilitar una serie de funciones que permitan coordinar efectivamente el uso compartido de ciertos datos, permitiendo paralelismo unicamente para la lectura, y otorgando un solo permiso de escritura; para ello cada *thread* deberá invocar funciones de *lock* y *unlock* de lectura/escritura, que soliciten y liberen respectivamente el permiso de acceso. A diferencia de otros mecanismos como el *Spin Lock*, el *Read-Write Lock* evita las situaciones de *polling*, permitiendo que los *thread* que permanezcan esperando el permiso de acceso, tras invocar un *lock* se mantengan dormidos hasta que el *thread* correspondiente libere dicho permiso. Por este motivo suele implementarse mediante *mutexs* y *variables de condición*.

1.1. Variables de condición

Las *variables de condición* son primitivas de sincronización utilizadas en asociación a un mutex con el fin de sincronizar dos o mas *threads* en función de un predicado. La idea es que al realizar un *wait* bloqueante sobre la *variable de condición* (asociando cierto *mutex* en dicho proceso), automaticamente (y atomicamente) se libere el mutex asociado, de manera que otro *thread* a la espera de dicho *mutex* continue su ejecución hasta cumplir cierta condición y luego este despertiete nuestra *variable de condición* por medio de alguna operación (por ejemplo un *signal* o un *broadcast*); vale aclarar que a diferencia de los *semaforos* clásicos, en este caso el efecto de estas últimas operaciones no es acumulativo, sino instantaneo. Tras esto, el *mutex* se vuelve a bloquear y el/los *thread* que hayan estado esperando a la *variable de condición* se despiertan y continuan su ejecución. Es importante aclarar que mas de un *thread* podría estar esperando a cierta *variable de condición* pero esperando que se cumplan distintas condiciones, por lo que podrían llegar a despertarse indebidamente; a estos casos se los llama *Spurious wakeups*, y por ello recomendarse chequear la condición no solo antes sino después del *wait* a la *variable de condición*, tras lo cual volveremos a bloquearnos al detectar que no se cumpla nuestra condición o predicado.

1.2. Pseudo-código

Para implementar este mecanismo, hemos diseñado la clase RWLock", en ella tendremos los siguientes atributos:

- *readers*, un entero sin signo que registra la cantidad de lectores actuales (que hayan tomado el permiso de lectura y todavia no lo hayan liberado).
- *writer*, un *flag* que indica si alguien tomo el permiso de escritura.
- *condition* y *lock_mutex*, son la *variable de condición* y su *mutex* asociado.
- *lock_writer*, un *mutex* de seguridad, usado para preservar la estabilidad de la variable *writer*.
- *lock_reader*, un *mutex* de seguridad, usado para preservar la estabilidad de la variable *reader*.

Tendremos además un metodo para cada *lock* y *unlock* de lectura y escritura con el siguiente *pseudo-código*:

- **rlock()**:
  1. Espero a tomar el *lock_mutex*.
  2. Mientras haya alguien escribiendo (es decir mientras este arriba el flag de *writer*) me mantengo esperando a *condition*.
  3. Ya no hay nadie escribiendo, asi que puedo leer, espero a tomar el *lock_reader*
  4. Me sumo a los lectores actuales incrementando la variable *readers*.
  5. Libero el *lock_reader*.
  6. Libero el *lock_mutex*.
- **wlock()**:
  1. Espero a tomar el *lock_mutex*.
  2. Mientras haya alguien escribiendo (es decir mientras este arriba el flag de *writer*) me mantengo esperando a *condition*.
  3. Tomo el *lock_writer*.
  4. Indico que se esta escribiendo o se esta intentando escribir, levantando el *flag* de *writer*.
  5. Libero el *lock_writer*.

6. Mientras haya alguien leyendo, es decir *readers* sea mayor a 0, me mantengo esperando a *condition*.
7. Libero el *lock_mutex*.

- **runock()**:

1. Espero a tomar el *lock_mutex*.
2. Espero a tomar el *lock_reader*
3. Como ya no voy a leer decremento *readers*.
4. Libero el *lock_reader*.
5. Si era el último lector (*readers* vale cero), lo hago saber mediante un *broadcast* sobre *condition*.
6. Libero el *lock_mutex*.

- **wunlock()**:

1. Tomo el *lock_writer*.
2. Indico que ya no hay nadie escribiendo, levantando el *flag* de *writer*.
3. Libero el *lock_writer*.
4. Hago saber que ya no hay nadie escribiendo despertando a los que esperen a *condition*, mediante un *broadcast*.
5. Libero el *lock_mutex*.

1.3. Implementacion

```
1  class RWLock {
2      public:
3          RWLock();
4          void rlock();
5          void wlock();
6          void runlock();
7          void wunlock();
8
9      private:
10         pthread_mutex_t lock_mutex;
11         pthread_mutex_t lock_writer;
12         pthread_mutex_t lock_reader;
13         pthread_cond_t condition;
14         bool writer;
15         unsigned int readers;
16
17 };
18
19 RWLock :: RWLock() {
20
21     pthread_mutex_init(&(this->lock_mutex), NULL);
22     pthread_mutex_init(&(this->lock_writer), NULL);
23     pthread_mutex_init(&(this->lock_reader), NULL);
24     pthread_cond_init (&(this->condition), NULL);
25     writer = false;
26     readers = 0;
27 }
28
29 void RWLock :: rlock() {
30
31     pthread_mutex_lock(&(this->lock_mutex));
32
33     while (writer)
34         pthread_cond_wait(&(this->condition), &(this->lock_mutex));
```

```
35
36      pthread_mutex_lock(&(this->lock_reader));
37      readers++;
38      pthread_mutex_unlock(&(this->lock_reader));
39
40      pthread_mutex_unlock(&(this->lock_mutex));
41  }
42
43  void RWLock :: wlock() {
44
45      pthread_mutex_lock(&(this->lock_mutex));
46
47      while (writer)
48          pthread_cond_wait(&(this->condition), &(this->lock_mutex));
49
50      pthread_mutex_lock(&(this->lock_writer));
51      writer = true;
52      pthread_mutex_unlock(&(this->lock_writer));
53
54      while (readers > 0)
55          pthread_cond_wait(&(this->condition), &(this->lock_mutex));
56
57
58      pthread_mutex_unlock(&(this->lock_mutex));
59  }
60
61  void RWLock :: runlock() {
62
63      pthread_mutex_lock(&(this->lock_mutex));
64
65      pthread_mutex_lock(&(this->lock_reader));
66      readers--;
67      pthread_mutex_unlock(&(this->lock_reader));
68
69      if (readers == 0)
70          pthread_cond_signal(&(this->condition));
71
72      pthread_mutex_unlock(&(this->lock_mutex));
73  }
74
75  void RWLock :: wunlock() {
76
77      pthread_mutex_lock(&(this->lock_writer));
78      writer = false;
79      pthread_mutex_unlock(&(this->lock_writer));
80
81      pthread_cond_signal(&(this->condition));
82  }
```

2. Backend multijugador

2.1. Loop de conexiones entrantes

```
1  // aceptar conexiones entrantes.
2  socket_size = sizeof(remoto);
3  int cant_users_act = 0;
4  pthread_t threads[CANT_USERS];
5  int socketfd_cliente[CANT_USERS];
6
7  while (true) {
8      if(cant_users_act < CANT_USERS) {
9          if ((socketfd_cliente[cant_users_act] = accept(socket_servidor, (struct
               sockaddr*) &remoto, (socklen_t*) &socket_size)) == -1)
10             cerr << "Error al aceptar conexion" << endl;
11         else {
12             pthread_create(&threads[cant_users_act], NULL, atendedor_de_jugador, &
                   socketfd_cliente[cant_users_act]);
13             cant_users_act++;
14         }
15     }
16 }
```

En este caso se tomo el codigo original y se lo modifico para que pueda recibir multiples conexiones por el mismo socket, para lograr esto se utilizo *threading* mediante el uso de la libreria *pthreads*. La idea de usar *threads* es poder tener varios hilos de ejecucion bajo el mismo marco de un proceso, compratiendo recursos entre ellos, en este caso tenemos el mismo tablero en memoria para varios jugadores haciendolo un caso ideal para utilizar *threading*.

Para crear cada uno de los hilos de ejecucion, primero se establecio la cantidad de maxima de usuarios que pueden estar conectados simultaneamente mediante la constante `CANT_USERS`, esto se hizo de esta forma para poder mantener un arreglo de *threads* y otro para guardar el *socket* por cada hilo de ejecucion, lo cual nos permite a la hora de crear un *thread* no entrar en conflictos por las posiciones de memoria, ya que todos serian independientes y no perderiamos ningun *socket*. Para poder indexar estos arreglo se utilizo una variable llamada `cant_users_act` incializada en `0`, la cual es aumentada luego de cada llamado a la funcion `pthread_create`.

2.2. Loop de jugador

```
1  while (true) {
2      // espera una letra o una confirmacion de palabra
3    char mensaje[MENSAJE_MAXIMO+1];
4    int comando = recibir_comando(socket_fd, mensaje);
5    if (comando == MSG_LETRA) {
6          Casillero ficha;
7          if (parsear_casillero(mensaje, ficha) != 0) {
8              // no es un mensaje LETRA bien formado, hacer de cuenta que nunca llego
9              continue;
10         }
11         // ficha contiene la nueva letra a colocar
12         // verificar si es una posicion valida del tablero
13         lock_juego.wlock();
14         if (es_ficha_valida_en_palabra(ficha, palabra_actual)) {
15             palabra_actual.push_back(ficha);
16             tablero_letras[ficha.fila][ficha.columna] = ficha.letra;
17       // OK
18       if (enviar_ok(socket_fd) != 0) {
19                 // se produjo un error al enviar. Cerramos todo.
20         terminar_servidor_de_jugador(socket_fd, palabra_actual);
21             }
22         } else {
23             quitar_letras(palabra_actual);
```

```
24          // ERROR
25          if (enviar_error(socket_fd) != 0) {
26                      // se produjo un error al enviar. Cerramos todo.
27            terminar_servidor_de_jugador(socket_fd, palabra_actual);
28                }
29            }
30            lock_juego.wunlock();
31        } else if (comando == MSG_PALABRA) {
32            // las letras acumuladas conforman una palabra completa, escribirlas en el tablero de palabras y borrar
                  las letras temporales
33        lock_juego.wlock();
34        for (list<Casillero>::const_iterator casillero = palabra_actual.begin();
              casillero != palabra_actual.end(); casillero++) {
35                tablero_palabras[casillero->fila][casillero->columna] = casillero->letra;
36            }
37            palabra_actual.clear();
38
39        if (enviar_ok(socket_fd) != 0) {
40                // se produjo un error al enviar. Cerramos todo.
41          terminar_servidor_de_jugador(socket_fd, palabra_actual);
42            }
43            lock_juego.wunlock();
44        } else if (comando == MSG_UPDATE) {
45            lock_juego.rlock();
46            if (enviar_tablero(socket_fd) != 0) {
47                // se produjo un error al enviar. Cerramos todo.
48                terminar_servidor_de_jugador(socket_fd, palabra_actual);
49            }
50            lock_juego.runlock();
51        } else if (comando == MSG_INVALID) {
52            // no es un mensaje valido, hacer de cuenta que nunca llego
53        continue;
54        } else {
55            // se produjo un error al recibir. Cerramos todo.
56            terminar_servidor_de_jugador(socket_fd, palabra_actual);
57        }
58    }
```

En lo que respecta al loop de atencion de jugador, no hubo cambios muy significativos, el mas importante de ellos fue la adicion de un *RWLock* (`lock_juego`) para poder controlar la concurrencia de manera correcta. Debido a que cada jugador corre en un *thread* individual, hay que tener en cuenta los casos de concurrencia que se pueden dar sobre el tablero, los comandos que fueron modificados son los siguientes:

- `MSG_LETRA` y `MSG_PALABRA`: Se agrego un lock de escritura en ambos, ya que estos dos comandos modifican variables globales del juego
- `MSG_UPDATE`: Se agrego un lock de lectura, ya que este comando realiza una lectura sobre la variable del tablero para poder actualizar el contenido del *frontend*

Con estas modificaciones es posible ejecutar el juego de manera correcta en un entorno multijugador.