



DEPARTAMENTO  
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# Trabajo Práctico II

“SOScrabel”

Sistemas Operativos  
Primer Cuatrimestre 2015

Integrante	LU	Correo electrónico
Aldasoro Agustina	86/13	agusaldasoro@gmail.com
More Ángel	931/12	angel_21_fer@hotmail.com
Zimenspitz Ezequiel	155/13	ezeqzim@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>

## 1. Read-Write Lock

Implementamos los algoritmos de **Read-Write Lock** libre de inanición utilizando únicamente *mutex* y respetando la interfaz provista en los archivos `backend-multi/RWLock.h` y `backend-multi/RWLock.cpp`.

Nuestro objetivo es restringir el acceso a la variable `tablero`, de modo que puedan leer los datos en él simultáneamente diversa cantidad de *lectores* pero sólo puedan *escribir* de a uno por vez y cuando nadie este leyendo.

La exclusión mutua que se lleva a cabo se comporta de modo que la ejecución de un thread en la sección crítica no excluye a otros threads que ingresen a la misma. Sin embargo, si determinado tipo de thread es quien se encuentra en la sección crítica prohíbe a las otras categorías de threads ingresar a ella.

Armamos la clase **Lightswitch**, la cual contiene un contador (`count`) el que llevará la cantidad de gente que se encuentra leyendo el tablero y un mutex (`mut`) para acceder al mismo.

Las funciones pertinentes a la clase serán: *lock* y *unlock*. Serán invocadas solamente por los *readers* para leer y dejar de leer el tablero respectivamente. Su comportamiento es simple: se toma el mutex para actualizar la variable `count` (+1 si es *lock*, -1 *unlock*) y luego sólo modificarán al mutex `m` pasado por parámetro si es el primero en leer o si es el último en dejar de leer.

Si ingresa un nuevo lector y no había nadie leyendo el tablero (`count == 1`), hará un wait del mutex `m`. Análogamente si es el lector que al retirarse, dejará al tablero vacío (`count == 0`) hará un signal liberando al mutex `m`.

Las variables con las que trabajarán las funciones Read-Write Lock serán tres: un **Lightswitch** llamado **readSwitch** y dos mutex **turnstile** y **roomEmpty**.

El `readSwitch` será el **Lightswitch** que se comportará acorde a lo explicado anteriormente únicamente para el uso del *reader*. De modo que permitirá a múltiples lectores acceder al tablero de manera simultánea.

Por otro lado, vamos a tener al mutex `roomEmpty`, el cual se va a habilitar únicamente cuando nadie este leyendo el tablero (sea leyendo o escribiendo). Es preciso aclarar que `roomEmpty` es el mutex que será pasado por parámetro a la hora de invocar a las funciones del `readSwitch`.

Por último, el mutex `turnstile` será el encargado de impedir la inanición. Los *escritores* son quienes podrían correr riesgo de *inanición* si sucede que llega una escritura, pero también lecturas y por no ejecutarse en orden siempre se bloquea el acceso al tablero por los *lectores*.

Tanto los *lectores*, como los *escritores* deberán pasar por el mutex `turnstile`, haciéndole wait. Pero los *lectores* le darán signal en la instrucción siguiente, mientras que los *escritores* lo harán después de esperar al mutex `roomEmpty`.

De esta manera, si un escritor no puede avanzar en el mutex `turnstile` obliga a los lectores que lleguen a “ponerse en espera” del `turnstile`. Y cuando el último lector de los que estaban en el tablero, lo abandone, permitirá al escritor tener acceso al tablero antes de cualquiera de los lectores que se ejecutaron después de él y estaban esperando.

## 2. Servidor Backend

Para crear el nuevo prototipo que permita a múltiples clientes jugar simultáneamente se usó como código base el provisto en *Backend\_mono.cpp*. Pero, dado que ahora hay más de un jugador se llevaron a cabo las modificaciones necesarias para evitar inanición, condición de carrera, o cualquier otro problema que intervenga en una utilización dinámica y correcta del juego.

Comenzamos agregando las siguientes variables globales (su uso se describirá más adelante):

```
mutex mutex_thread;
Matriz <RWLock> rwlockTablero;
mutex mutex_jugadores;
unsigned int jugadores_activos;
```

Se inicializa las dimensiones del tablero de palabras (que contendrá a las palabras formadas, *tablero\_palabras*), como así la del de letras (que posee letras que no conforman ninguna palabra, *tablero\_letras*) y el *rwlockTablero* con las mismas dimensiones. *rwlockTablero* será una matriz de mutex, con el objetivo de que cuando distintos jugadores quieran colocar una letra en una misma posición solo puedan hacerlo si se tiene habilitado el mismo, de esta forma solo uno será quien pueda colocar la letra pero, no así si varios quieren escribir en distintas posiciones, ya que podrán hacerlo simultáneamente si en esas posiciones el mutex está disponible; e inicializaremos el mutex *mutex\_thread*.

Se procede a establecer la conexión con el servidor mediante el uso de socket. Para permitir que varios jugadores se conecten, por cada jugador (o cliente) crearemos un thread y a cada uno de estos se le asociará la función *atendedor\_de\_jugador* que recibe como parámetro el file descriptor del socket asociado a cada jugador.

```
while (true) {
    if ((socketfd_cliente = accept(socket_servidor, &socket_remoto, &socket_size)) == -1)
        "Error al aceptar conexión";
    else {
        pthread_t thread;
        pthread_create(&thread, NULL, atendedor_de_jugador, socketfd_cliente);
    }
}
```

*atendedor\_de\_jugador* es similar al de *Backend\_mono.cpp* pero dado que ahora hay más de un jugador se utilizarán los mutex necesarios para evitar cualquier conflicto mencionado anteriormente. Al permitir varias conexiones simultáneamente y dado que al aceptar una conexión pasamos por referencia el socket, este se verá modificado cuando distintos clientes se conecten, para esto dentro de la función *atendedor\_jugador* nos guardamos el file descriptor del socket en *sockAUX* y además incrementamos la variable *jugadores\_activos*. Al momento de actualizar estas últimas variables si el scheduler asigna tiempo de ejecución a otros threads a ambas variables es posible que se le asigne un valor erróneo, por lo que serán protegidas por el mutex *mutex\_thread* y *mutex\_jugadores* respectivamente. De ahora en más las funciones que utilicen al socket harán uso de *sockAUX* que será distinto para cada thread.

En los casos de que se corte la comunicación, o se produjera un error al enviar las dimensiones, es necesario terminar el servidor del juego. A diferencia de la implementación para *Backend\_mono* ahora es necesario decrementar la variable cantidad de jugadores, lo mismo se realizará luego de hacer un wait a *mutex\_jugadores* y al reducirlo se hará un signal, en caso de no haber más jugadores se procede a terminar el programa (*exit(-1)*) sino solo se procede a terminar ese thread con *pthread\_exit*.

Finalmente se procede a esperar la colocación de una letra o confirmar una palabra. Cuando un jugador quiera escribir una letra en el tablero, primero se revisará si puede hacerlo mediante *es\_ficha\_valida\_en\_palabra*. *es\_ficha\_valida\_en\_palabra* chequeará que la posición en la que se quiere a la letra coincida con las dimensiones del tablero; que sea una posición libre; y en el caso de haya una palabra completa las letras a ubicar sean adyacentes a esta (horizontal o verticalmente). Dado que lo que se va a querer realizar es una escritura en esa posición (de ser posible), con el mutex correspondiente a dicha posición dentro de *rwlockTablero* se chequeará, de forma correcta, que la palabra exista y esté ubicada de forma correcta.

En caso de que *es\_ficha\_valida\_en\_palabra* de *true* mediante la matriz de mutex, procederemos a escribir de esta manera si simultáneamente varios jugadores quieren hacerlo en una misma posición solo uno será asignado (podrá hacer wait del mutex), mientras que todos los demás deberán esperar el signal correspondiente.

```
if (es_ficha_valida_en_palabra(ficha, palabra_actual)) {
    palabra_actual.push_back(ficha);
    rwlockTablero[ficha.fila][ficha.columna].wlock();
    tablero_letras[ficha.fila][ficha.columna] = ficha.letra;
    rwlockTablero[ficha.fila][ficha.columna].wunlock();
}
```

En caso de dar *false* es necesario quitar la letra, como ningún otro jugador debe escribir antes de que se retire, sino se eliminaría una letra errónea, lo haremos con el uso de los wlocks y wunlocks.

```
void quitar_letras(list palabra_actual) {
    for ( iterator casillero = palabra_actual.begin(); casillero != palabra_actual.end();
        casillero++) {
        rwlockTablero[casillero->fila][casillero->columna].wlock();
        tablero_letras[casillero->fila][casillero->columna] = VACIO;
        rwlockTablero[casillero->fila][casillero->columna].wunlock();
    }
    palabra_actual.clear();
}
```

Algo similar ocurre cuando se completa una palabra solo que ahora se hará uso del *tablero\_palabras*. En primer lugar es necesario actualizar el tablero por cada jugador, para esto es necesario leer el mismo. *enviar\_tablero* se encargará de actualizar el tablero, como ya se mencionó puede haber varios lectores pero mientras los haya, ningún escritor puede escribir, es por esto que usaremos las funciones rlock, runlock antes y después de leer cada letra del tablero respectivamente garantizando una correcta lectura y evitando que se escriba en el mismo hasta que todos los readers hayan leído. En segundo lugar dado que hay varios jugadores es necesario que se termine de escribir esa palabra completamente y que la asignación del scheduler a otro thread no altere las posiciones o la palabra que quedó sin completar para esto cuando se escribe la palabra completa se hará por medio del mutex wlock:

```
for (iterator casillero = palabra_actual.begin(); casillero != palabra_actual.end();
    casillero++) {
    rwlockTablero[casillero->fila][casillero->columna].wlock();
    tablero_palabras[casillero->fila][casillero->columna] = casillero->letra;
    rwlockTablero[casillero->fila][casillero->columna].wunlock();
}
```

Si para las funciones anteriores no se produjo ningún error entonces el o los casilleros habrán quedado actualizado correctamente sino se terminara el thread o todo el proceso, según corresponda con *terminar\_servidor\_de\_jugador*.