



DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo práctico II

PTHREADS

Sistemas Operativos

| Integrante | LU | Correo electrónico |
|---------------------------|--------|---------------------------|
| Cámara, Joel Esteban | 257/14 | joel.e.camera@gmail.com |
| Lavia, Alejandro Norberto | 43/11 | lavia.alejandro@gmail.com |
| Guttman, Martín David | 686/14 | mdg_92@yahoo.com.ar |

Nota: I. Reentregar. Fundamentalmente corregir el tema de la variable `i` y el test 3

- Falta inicializar las variables de `rwlock`.

- En `rlock`, tendrías `writing` o nada... solo puede escribir uno a la vez.

```
if(this->writers > 0)
```

```
pthread_cond_wait(&this->turn, &this->m);
```

- En el test `testMitadEscritoresMitadLectores`, pasás mal la variable `i`, ya que puede terminar llegando la de otro caso !

- Tu test tira

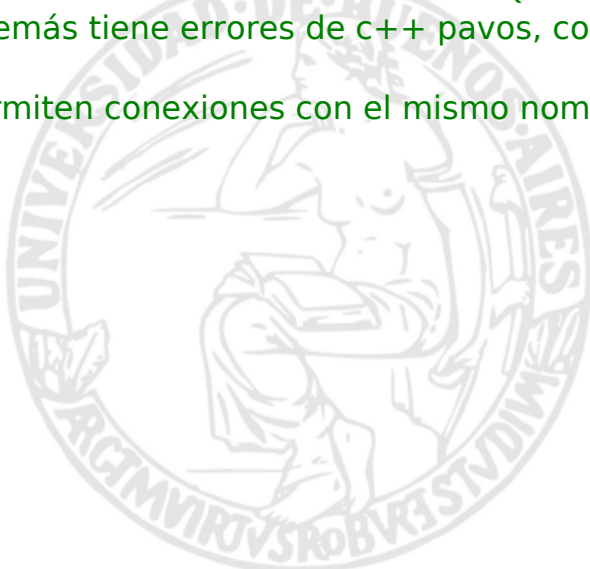
Corriendo el test 3.

ERROR! Inanición en test 3.

ERROR! en test: `testMasEscritoresQueLectores`.

Además tiene errores de `c++` pavos, como chequear un `int` que siempre vale `CANT_THREADS`, etc

Permiten conexiones con el mismo nombre



Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2610 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (+54 11) 4576-3300

<http://www.exactas.uba.ar>

Índice

| | |
|---|----------|
| 1. Introducción | 2 |
| 2. Implementación <i>Read-Write Lock</i> | 2 |
| 2.1. Estructura | 2 |
| 2.2. Función <i>rlock()</i> | 2 |
| 2.3. Función <i>wlock()</i> | 3 |
| 2.4. Función <i>runlock()</i> | 3 |
| 2.5. Función <i>wunlock()</i> | 3 |
| 2.6. Inanición | 3 |
| 2.7. Deadlock | 4 |
| 2.8. Tests | 4 |
| 3. Servidor de backend | 5 |
| 3.1. Backend mono cliente | 5 |
| 3.2. Backend múltiples clientes | 5 |
| 3.2.1. Función Atender Jugador | 6 |
| 3.2.2. Función Enviar Tablero | 6 |
| 3.2.3. Función Quitar Cartas | 6 |
| 3.2.4. Función Es Ficha Valida en Jugada | 6 |
| 3.2.5. Función Terminar Servidor de Jugador | 6 |

1. Introducción

En el presente informe reseñamos las mejoras realizadas al servidor del juego de *HaSObro* para que tener múltiples jugadores a la vez. Para esto, utilizamos la biblioteca **pthread**s de POSIX para realizar los cambios en el servidor de *backend* así pueda atender a múltiples jugadores.

Como primera parte del mismo, exponemos la implementación de la estructura *Read-Write lock*, donde se utilizan sólo variables de condición de la librería **pthread**s, y los respectivos test que demuestran que la estructura está libre de inanición y posee el comportamiento esperado. Como segunda parte, presentamos una implementación del servidor de *backend multiusuario* con los detalles implementativos y las decisiones de diseño del mismo que hacen que permita múltiples clientes jugando simultáneamente.

2. Implementación *Read-Write Lock*

Aquí presentamos nuestra implementación del *Read-Write Lock* donde exponemos su estructura y su funcionamiento. En ésta se podría generar el problema de inanición si se presentan los casos en donde haya siempre pedidos de lectura o escritura, al intentar realizar un pedido contrario a estos según el caso éste queda bloqueado. Por ello, también presentamos las medidas tomadas para evitar estos casos.

2.1. Estructura

Para poder implementar el *Read-Write Lock* utilizamos una estructura basada en una variable de condición y un mutex, además de dos enteros que cuentan la cantidad de lectores y escritores y un flag.

La estructura es la siguiente:

- Mutex: **m**
- Variable de condición: **turn**
- Contador de lectores: **reading**
- Contador de escritores: **writers**
- Flag: **writing**

2.2. Función *rlock()*

Pido el mutex

Si la cantidad de escritores es mayor que cero
Espero el proximo turno

Mientras haya alguien escribiendo
Esperar el proximo turno

Aumento en uno el contador de la cantidad de lectores

Libero el mutex

Es lo mismo. Sólo puede haber 1 escritor en la región crítica

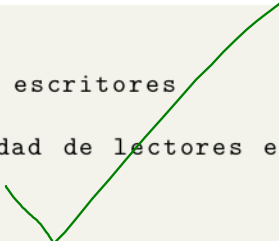
Lo primero realiza esta función es pedir el mutex. Una vez obtenido el mismo, se fija si la cantidad de escritores es mayor que cero y, si lo es, espera al proximo turno. Esto se hace para parar una posible seguidilla de pedidos de lectura que produzcan inanición a los threads que pidan una lectura.

Luego, mientras haya algún thread escribiendo se espera hasta el proximo turno. Esto se hace para garantizar el acceso exclusivo a la sección crítica.

Por último, una vez que no haya algún thread escribiendo se aumenta en uno el contador de la cantidad de lectores y se libera el mutex.

2.3. Función *wlock()*

```
Pido el mutex  
Aumento en uno el contador de la cantidad de escritores  
Mientras haya alguien escribiendo o la cantidad de lectores es mayor que cero  
    Espero el proximo turno  
Levanto el flag de escritura  
Libero el mutex
```




Primero se pide el mutex. Una vez obtenido el mismo se aumenta el contador de la cantidad de escritores en uno. Luego, mientras haya alguien escribiendo o leyendo, espera al proximo turno. Esto se hace para garantizar el acceso exclusivo a la sección critica ya que si alguien está escribiendo no debería acceder otro para escribir y, también, para que se escriba solamente cuando nadie esté leyendo.

Una vez que no haya nadie escribiendo ni leyendo, se levanta el flag de escritura y se libera el mutex.

2.4. Función *runlock()*

```
Pedir mutex  
Disminuyo en uno el contador de la cantidad de lectores  
Si la cantidad de lectores es igual a cero  
    Dar turno a todos los que esperan uno  
Liberar mutex
```

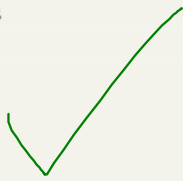


Una vez terminada la lectura, se pide el mutex nuevamente y se disminuye en uno el contador de la cantidad de lectores.

Luego, si ya no hay más lectores se da turno a todos los que esperan uno, tanto lectores como escritores. Y, por último, se libera el mutex.

2.5. Función *wunlock()*

```
Pido el mutex  
Disminuyo en uno el contador de la cantidad de escritores  
Bajo el flag de escritura  
Doy turno a todos los que esperan uno  
Libero el mutex
```



Una vez terminada la escritura, se pide el mutex nuevamente, se baja el flag de escritura y se disminuye el contador de la cantidad de escritores.

Luego, se da un turno a todos los que esperan uno, tanto lectores como escritores y se libera el mutex.

2.6. Inanición

Por lo expuesto anteriormente sobre el funcionamiento de nuestra estructura, ésta contiene los medios suficientes para evitar que una seguidilla de threads con una misma operación hagan que se bloquéen threads con la operación contraria.

En el caso de los lectores, estos esperan a que terminen los escritores de escribir. Y, en el caso de los escritores que no haya threads escribiendo o threads leyendo.

Pero cómo sabés que no pueden quedar siempre llegando escritores ?

2.7. Deadlock

Para mostrar que en la estructura no se generaría deadlock veamos que no se cumplen las condiciones necesarias de Coffman.

- **Exclusión mutua:** Un recurso no puede estar asignado a más de un proceso.
Cómo vimos, todas las operaciones como primera medida realizan un lock con el mutex, esto garantiza que el acceso a un recurso le pertenece a un sólo thread y todos los que quieran utilizarlo esperan hasta que el que lo está utilizando realice un unlock. Si se cae en una variable de condición se liberan los mutex pero todos los threads caerían también en la misma variable de condición y éstas están dentro de ciclos donde, si se quiere escribir se espera a que terminen de escribir y que hayan leído los lectores, o si se quiere leer que terminen de escribir. Entonces los lectores son los únicos que podrían entrar varios a la vez pero no modificarían las variables, en cambio los que quieren escribir tienen acceso exclusivo.
- **Hold and wait:** Los procesos que ya tienen algún recurso pueden solicitar otros.
En este caso, esto ocurriría si un thread no puede leer o escribir se quede con el mutex hasta que pueda realizar la operación. Pero esto no sucede ya que los mutex se liberan en el momento que el thread llama a la función `pthread_cond_wait()` para esperar que la condición se cumpla para realizar la operación. Esto si sucede. Pero por poco tiempo, o sea hasta que escribe en el tablero
- **No Preemption:** Un recurso puede ser liberado solamente de forma voluntaria por el proceso que lo retiene.
En este caso, no podría suceder esto ya que los threads que quieran acceder a un recurso que esta siendo utilizado deben esperar a que la variable de condición sea verdadera y ésta sólo puede ser liberada por el proceso que tiene el recurso. Por eso, sucede.
- **Espera circular:** Tiene que haber un ciclo de $N \geq 2$, tal que P_i espera un recurso que tiene P_{i+1} .
Cómo el recurso al que se quiere acceder es único en este caso, no puede haber espera circular. Y esta condición es la que no se cumple....

2.8. Tests

Para probar nuestra implementación de *Read-Write Lock* realizamos tres tipos de test que se pueden encontrar en la carpeta **locks** junto con la implementación de la estructura. En cada uno de estos lanzamos 10.000 threads.

Pasamos a explicar sobre ellos:

1. En este primer test creamos 5000 threads escritores y 5000 threads lectores todos con un único número, siendo los pares los threads lectores y los impares los escritores. Lo que se hace es guardar en un arreglo cuantos lectores hubo hasta el momento de que esta un thread escritor y viceversa.
Una vez que corrieron todos los threads, se revisa ese arreglo y se mira si la cantidad de threads lectores o escritores está dentro de un rango de 15 threads (tomado este número de forma arbitraria) de diferencia según el thread del índice. Si está por debajo de ese rango quiere decir que se tarda en leer (si el índice es par) o en escribir (si el índice es impar). Si está por arriba del rango quiere decir que tiene una prioridad mayor leer (si el índice es par) o escribir (si el índice es impar).
2. En el segundo test realizamos un experimento con la misma logica que el primero pero con la diferencia que un 90 % de los threads son escritores y un 10 % threads son lectores.
3. En este tercer y último test también realizamos un experimento con la misma logica que el primero pero esta vez un 90 % de los threads son lectores y un 10 % son escritores.

3. Servidor de backend

En esta sección vamos a detallar los cambios realizados al servidor de backend. Originalmente contábamos con una implementación mono cliente y nuestra tarea es adaptar esta implementación para permitir múltiples clientes en simultaneo bien sincronizados.

3.1. Backend mono cliente

Inicialmente se setean los tableros del juego de n por m y también el socket de tipo **INET** con protocolo **TCP**. Una vez hecho esto, se queda a la espera de una conexión entrante y, cuando llega, la acepta y establece el socket entre el cliente y el servidor. Luego, se cierra el socket del servidor para nuevas conexiones. El flujo de ejecución se queda atendiendo las jugadas del cliente, hasta que este finalice. El problema de ésta implementación es que no solo se cierra el socket del servidor sino que, además, si no se hubiera hecho, el servidor se queda ejecutando la rutina de atención del único cliente siendo imposible aceptar nuevas conexiones.

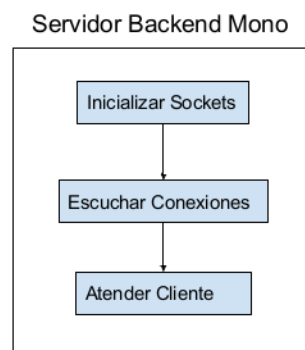


Diagrama el funcionamiento del backend mono.

3.2. Backend múltiples clientes

El cambio más importante para darle soporte multiusuario al backend radica en el uso de threads de la librería **pthread**. Inicialmente realizamos las mismas operaciones que el backend mono para setear el tablero y los sockets y luego nos quedamos a la espera de conexiones entrantes. Pero, al momento de recibir una nueva conexión en lugar de derivar todo el flujo de ejecución del programa a la rutina de atención del cliente, creamos un nuevo hilo de ejecución (*thread*) que se encargara de ello con la función **pthread_create**. También, mantenemos abierto el socket a la espera de nuevos clientes y tenemos ejecutándose concurrentemente la rutina que acepta a los clientes y las rutinas que atienden a los cliente así puedan conectarse muchos clientes en el juego.

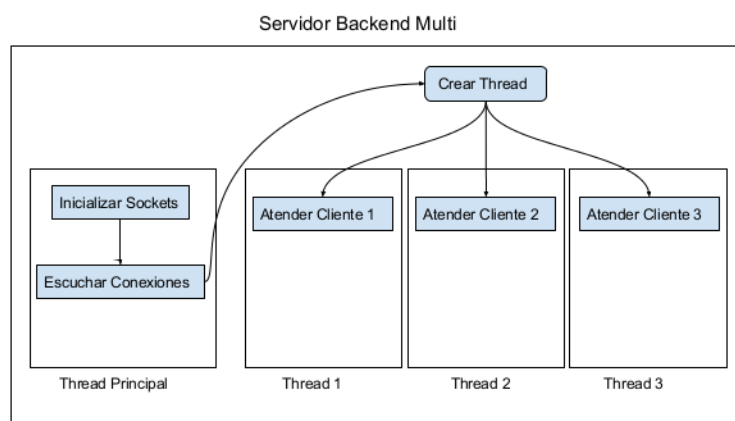


Diagrama ilustrativo del backend multiusuario

La siguiente cuestión a resolver para mantener la correcta sincronización entre los hilos y estar libres de condiciones de carrera, fue determinar las zonas críticas del servidor. El servidor cuenta con dos variables globales compartidas por todos los threads, estas son *tablero_temporal* y *tablero_confirmado*. Estas variables son leídas y escritas con las distintas acciones de los clientes, por lo tanto estas serán nuestras secciones críticas a las cuales debemos garantizar el acceso exclusivo para mantener la coherencia. Esto lo hacemos aprovechando los mecanismos provistos por la clase implementada anteriormente **RWLock**, así que agregamos dos instancias globales de esta clase, *lock_tablero_temporal* y *lock_tablero_confirmado* con el fin de garantizar el acceso exclusivo a las secciones críticas. Vamos a listar las funciones del juego con los cambios efectuados:

3.2.1. Función Atender Jugador

- Colocar una carta: En esta operación estamos modificando el tablero temporal con la carta colocada, solicitamos el *write-lock* del tablero temporal, efectuamos el cambio, y luego lo liberamos *write-unlock*.
- Confirmar jugada: Necesitamos escribir en el tablero confirmado la jugada del cliente por ello pedimos *write-lock* en el tablero confirmado y una vez obtenido pasamos la totalidad de la jugada al tablero y al finalizar liberamos *write-unlock*. Realizamos esto de esta forma para garantizar el acceso exclusivo mientras se esta confirmando la jugada, ya que de no hacerlo podemos tener condiciones de carrera.

3.2.2. Función Enviar Tablero

Enviamos el tablero confirmado a un cliente y para eso leemos el tablero confirmado, así que solicitamos el *read-lock*, efectuamos la lectura de todo el tablero y luego hacemos un *read-unlock* para liberarlo. Nuevamente, la operación debe realizarse exclusivamente al leer todo el tablero para evitar que otros clientes puedan escribirlo mientras se esta enviando.

3.2.3. Función Quitar Cartas

El tablero temporal debe ser limpiado, se debe escribir en todas sus coordenadas, por ello realizamos un *write-lock*, limpiamos todas las coordenadas y luego un *write-unlock* al finalizar.

3.2.4. Función Es Ficha Valida en Jugada

En esta operación comprobamos si una jugada es válida, para ello debemos leer tanto el tablero temporal como el tablero confirmado, solicitamos con sus respectivas instancias de **RWLock** el *read-lock* al efectuar la lectura tanto filas como columnas, y luego lo liberamos con *read-unlock*.

3.2.5. Función Terminar Servidor de Jugador

Por último, luego de garantizar el acceso exclusivo a las secciones críticas, sólo nos queda finalizar el thread cuando el cliente termine su conexión con el servidor, para ello agregamos en esta rutina encargada de terminar el cliente un llamado a **pthread_exit** que indica que el hilo actual ha finalizado.