Sistemas Operativos

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

Trabajo Práctico 2

Integrante	LU	Correo electrónico
Bayardo Julián	850/13	julian@bayardo.com.ar
Cuneo Christian	755/13	chriscuneo93@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Servidor Multi Cliente

Al abordar este trabajo lo primero que buscamos fue implementar de forma correcta la adaptación de el servidor mono-cliente a un servidor multi-cliente utilizando la implementación de bloqueo de recursos provista por 'pthreads', que esta preparada para atacar un problema de Lectores-Escritores, el cual es el problema principal en este trabajo, ya que los múltiples clientes van a necesitar actualizar el estado de la tabla de juegos localmente, es decir 'leer' la tabla de juegos, y formar palabras nuevas y reflejarlas en la tabla de juegos, es decir 'escribir' la tabla se juegos.

Para esto lo primero que pensamos fue que datos son los que necesitamos proteger con estos bloqueos, es decir cuales son los datos para los cuales necesitamos garantizar un acceso controlado de lectura y escritura. Llegamos a los siguientes: una tabla de juego donde guardar las palabras ya ingresadas y una tabla donde guardar las letras de las palabras aun no terminadas (ya que necesitamos garantizar que a la hora de finalizar una palabra esta pueda ser ingresada a la tabla de juego y que no estén ocupados sus lugares).

Para administrar estos recursos lo que pensamos fue crear un lock por posición de la tabla de letras temporal, y un lock único para la tabla del juego real. De esta forma podemos permitir que se acceda de forma simultanea a posiciones diferentes del tablero a la hora de formar palabras, y un acceso en serie a la tabla de juego a la hora de grabar los cambios (finalizar una palabra).

Luego de establecer los recursos, la implementación del servidor consistió en adaptar el código del servidor mono-cliente para lograr que el servidor cree un nuevo hilo en el momento que se conecte un nuevo cliente y que todos estos hilos compartan el tablero (es global tanto el tablero de juego y el temporal, como las variables de bloqueo), y luego se colocaron los bloqueos de recursos de forma tal que permita un uso controlado de los mismos. Para lograr todo esto se necesito modificar levemente la implementación original, mas que nada se cambio la función principal para que lance threads a medida que se conecten nuevos clientes, los cuales van a iniciar ejecutando la función que se encarga de atender clientes, nuevamente levemente modificada para aceptar un numero de socket por parámetro y para aplicar el uso de RWLocks para proteger los recursos segun sea necesario, un bloqueo para lectura del tablero a la hora de recibir un pedido de actualización de tablero, uno para escritura del tablero a la hora de grabar el tablero, y por ultimo uno de escritura del tablero temporal cuando se recibe una nueva letra, para ver si esta posición es valida y grabarla.

2. Read Write Locks

Para la implementación de read write locks utilizamos 2 variables de condición, 2 mutexes y 2 variables enteras. A las variables enteras les llamaremos *cant_escritores* y *cant_lectores*, mientras que los mutexes serán los mismos nombres con guión bajo y m agregados al final, y las variables de condición guión bajo y c agregados al final. Estas variables representan, como sus nombres invitan a pensar, la cantidad de escritores y cantidad de lectores que poseen el lock en simultaneo; mientras que los mutexes son para garantizar acceso exclusivo a las mismas y las variables de condición para avisar cuándo cambian sus valores.

Queda claro que, si la implementación fuese correcta, la cantidad de escritores sólo puede ser mayor a 0 si la cantidad de lectores es exactamente 0, y viceversa. Además, la cantidad de escritores debe ser a lo sumo 1 (pues no deberíamos tener más de 1 escritor andando al mismo tiempo). Con esta observación queda bastante claro qué deberíamos hacer al intentar hacer lock: si fuese un read lock, deberíamos esperar a que la cantidad de writers sea 0, para luego aumentar la cantidad de readers; en el caso de un write lock, deberíamos esperar a que la cantidad de readers sea 0, luego que la cantidad de writers sea 0 también, y recién ahí podríamos aumentar la cantidad de writers. En términos del unlock, en el caso en que es un read unlock tenemos que decrementar la cantidad de lectores; y el caso en que es un write unlock tenemos que decrementar la cantidad de escritores.

A grandes rasgos, esto parecería solucionar el problema de tener exclusión mutua entre writers con writers y writers con readers, aunque aun no determinamos cómo evitar condiciones de carrera e inanición de writers o readers. Para solucionar el problema de las condiciones de carrera, queda claro que deberíamos utilizar un mutex por variable, que tomaríamos antes de modificar y cuando pretendemos leer la variable, de la misma forma en que hacemos típicamente. Sin embargo, observemos que tanto en los locks como en los unlocks estaríamos realizando busy waiting esperando que

se cumpla una proposición, indicándonos que una variable de condición podría ayudarnos. Entonces, decidimos reemplazar el loop original de busy waiting con un loop que utilizase una variable de condición, esperando que la cantidad que buscábamos se actualizase. Es así como llegamos al código de los algoritmos 1.

Algorithm 1 Pseudocódigo de algoritmos para read write locks

```
function RLock
    cant_lectores_m.lock()
    cant_escritores_m.lock()
    while cant_escritores > 0 do
        cant_escritores_c.wait(cant_escritores_m)
    end while
    cant_lectores++
    cant_escritores_m.unlock()
    cant_lectores_m.unlock()
end function
function RUnlock
    cant_lectores_m.lock()
    cant_lectores–
    if cant_lectores = 0 then
        cant_lectores_c.signal()
    end if
    cant_lectores_m.unlock()
end function
function WLock
    cant_lectores_m.lock()
    while cant_lectores > 0 do
        cant_lectores_c.wait(cant_lectores_m)
    end while
    cant_escritores_m.lock()
    while cant_escritores > 0 do
        cant_escritores_c.wait(cant_escritores_m)
    end while
    cant_escritores++
    cant_escritores_m.unlock()
    cant_lectores_m.unlock()
end function
function WUnlock
    cant_escritores_m.lock()
    cant_escritores–
    cant_escritores_c.broadcast()
    cant_escritores_m.unlock()
end function
```

Cuando decidimos cambiar el código para utilizar variables de condición, quedó claro que debíamos enviar una señal en cada ocasión donde decreciese alguna de las variables de cantidad, despertando de esta forma a los readers o writers que estuviesen en espera. Sabemos que un writer sólo puede tomar el write lock cuando no haya readers, por lo que sólo debemos hacer un signal de cambio de la cantidad de readers cuando la misma sea 0. Por otro lado, sabemos que cuando un writer hace un unlock, deberíamos tener dos posibilidades: o entra un nuevo writer, o entran uno o más readers; esto podemos asegurarlo utilizando un broadcast en lugar de un signal.

Tomó tiempo darnos cuenta de que el código que mostramos en el algoritmo 1 es incorrecto. Si bien no logramos encontrar que tenga deadlock, el problema que tiene (y que no pudimos reproducir) es que si entrasen múltiples readers, uno atrás del otro, tendríamos un inanición de writers. Se nos ocurrió poner una variable que indicase cuántos writers están esperando, y que los readers no puedan entrar con alguna condición sobre esta variable. Queda claro que poner esta

cantidad en un número fijo causaría inanición de readers a causa de los writers, pero si en el caso en que haya más de 1 writer tirásemos una moneda y en alguno de los dos casos hiciésemos pasar a todos los writers que habían hasta el momento antes de seguir, ya estaría.

En el acto de programar esta nueva implementación nos dimos cuenta que el código se había tornado demasiado complejo y no podíamos entender claramente si lo que estábamos haciendo estaba bien, por lo que decidimos comenzar de vuelta. Nos dimos cuenta que teníamos que utilizar 3 variables: una con la cantidad de writers, otra con la cantidad de readers, y una última con la cantidad de writers esperando para entrar. Luego, simplemente utilizar un sistema de turnos podría andar bien: queremos de alguna forma que las lecturas y escrituras se ïntercalen", entonces basta con utilizar una variable de condición que diga de quién es el turno: es el turno de un writer si no hay nadie leyendo ni escribiendo, y es el turno de un reader si no hay nadie esperando escribir o escribiendo. Es decir, es el turno de quien quiera pueda hacer lo que precise con la variable; de esta forma transferimos el problema de inanición a que la política de scheduling no produzca inanición.

Algorithm 2 Pseudocódigo de algoritmos para read write locks corregido

```
function RLock
    mutex.lock()
    if escritores > 0 then
        turno.wait(mutex)
    end if
    while escribiendo > 0 do
        turno.wait(mutex)
    end while
    lectores++
    mutex.unlock()
end function
function RUnlock
    mutex.lock()
    lectores–
    turno.broadcast()
    mutex.unlock()
end function
function WLock
    mutex.lock()
    escritores++
    while lectores > 0 o escribiendo > 0 do
        turno.wait(mutex)
    end while
    escribiendo++
    mutex.unlock()
end function
function WUnlock
    mutex.lock()
    escritores–
    escribiendo–
    turno.broadcast()
    mutex.unlock()
end function
```

Observando el código de los algoritmos 2, es claro que no hay espera circular, ya que utilizamos un único mutex, que se toma al principio de cada función y se libera al final de las mismas. Por lo tanto, no hay deadlock.

Entender por qué no hay inanición es más difícil:

En términos de los writers, observemos que en cuanto un writer entra, aumenta la cantidad de escritores esperando, lo que forzará a cualquier reader que entre después del writer a hacer la

primer llamada al wait, haciendo que todos los readers que entren después queden esperando por un turno extra. Luego, comenzamos a hacer polling sobre la variable de condición: en cada cambio de turno, verificamos si hay alguien que ya haya comenzado a leer o escribir; si nadie lo hizo, podemos tomar el control y comenzar a escribir nosotros, en caso contrario, seguimos esperando.

Supongamos, entonces, que actualmente hay ejecutando un writer o un reader y sucede que entra un writer y luego entran muchos readers, todos estos se quedarán esperando en el primer wait de la variable de condición, por lo que cuando termine el thread actual de utilizar el lock, el broadcast despertará a todos en algún orden. Si se despierta primero un reader, el mismo podrá pasar de largo el while y comenzar a leer, forzando a que el writer tenga que esperar al menos un turno más; si se despierta primero un writer, el mismo podrá comenzar a escribir, forzando a todos los readers a quedarse esperando en el while. En el caso en que se despierte primero un reader, todos los readers que se ejecuten después del mismo en el broadcast, encontrarán que podrán pasar de largo el while y comenzar a escribir.

Pensemos qué sucedería entonces si tras esta secuencia (el caso en que se despierta un reader) continúan entrando readers: los mismos encontrarán que la variable escritores es mayor a 0, y por lo tanto quedarán esperando en la primer variable de condición. Sin embargo, asumiendo que el algoritmo que utilizan las variables de condición para despertar en el broadcast garantiza que no hay inanición, el writer que había entrado anteriormente debería haber estado esperando en la variable por una cantidad de tiempo mucho mayor a los nuevos readers, por lo que el cambio de turno debería despertar primero al writer, que tomaría el control mientras que todos los nuevos readers pasarían a esperar en el while.

Por otro lado, si hay múltiples writers entrando, queda claro que todos van a terminar esperando en el while, y la política de scheduling de las variables de condición debería despertar a todos eventualmente. El mismo razonamiento puede ser aplicado para el caso en que hay múltiples writers y múltiples readers entrando. Con esto concluimos que no puede haber inanición de writers.

Ahora veamos como evitamos la inanición de readers: Siguiendo el proceso que conlleva pedir un recurso para lectura, vemos que lo primero que se hace, luego de pedir el mutex, es esperar por la condición de turno. Pero esto se hará solo si hay escritores actualmente esperando y/o usando el recurso (esto se ve reflejado en la variable escritores), en el caso que esta variable sea 0, el lector continua, en caso contrario se espera la condición, pero es importante notar que esto se realiza solo una vez, ya que utilizamos un bloque condicional en vez de un ciclo condicional, y este el eje central a la hora de ver que no hay inanición de lectores. Esto es así ya que, al solo esperar una vez por la condición, solo dependerá de la implementación de variables de condición de pthreads, específicamente a la hora de decidir a que proceso despertar cuando haya varios procesos esperando por una misma condición.

Procediendo en el código, veamos que luego se incrementa la variable 'lectores' y se va a esperar otra condición: que no haya ningún writer escribiendo, ahora si, una vez que se encuentre en esta etapa del proceso, si un nuevo writer requiere usar el recurso, este tendrá que esperar ya que la variable 'lectores' fue incrementada, por lo tanto los writers que entren posteriormente tendrán que esperar a los readers que hayan llegado a esta etapa anteriormente.

3. Tests

Para testear la implementación, decidimos armar un test donde tenemos una variable entera v inicializada en 0, y una cantidad R de readers y W de writers. El test consiste en crear R threads de readers y W threads de writers. Los writers se encargan de aumentar en 1 la variable v, y los readers intentan leer la variable hasta que sea exactamente igual a W. Además, cada uno de los threads imprime en pantalla el estado de la variable cuando la tuvo, y cómo la dejó. Variaciones que utilizamos para testear incluyen agregar sleeps entre cada iteración de lectura para ver cómo se intercalan los readers y writers. Cabe destacar que para la impresión en pantalla utilizamos un mutex; sin embargo, el mismo es utilizado fuera de donde utilizamos los read write locks para no interferir con el test (observemos que esto fuerza a que las lineas que imprima cada thread no mezclen sus contenidos, pero que no implica absolutamente nada con respecto al orden de ejecución de los threads, de hecho, las lineas pueden venir fuera del orden esperado). Correr el test múltiples veces alternando parámetros siempre llegó a terminar correctamente, tendiendo a demorar más

cuanto mayor cantidad de readers haya con respecto a la cantidad de writers, ya que la prioridad está puesta sobre los readers.

Este test se llama 'rwlocktest' y se llama con dos parámetros, primero la cantidad de threads 'readers' y luego la cantidad de threads 'writers'.

A pesar de haber usado cuidadosamente un mutex para garantizar que se impriman correctamente los mensajes de output, a la hora de testear se nos ocurrió quitar las escrituras a pantalla y al correr los test con muchos readers y pocos writers se llegaba a un estado de inanición de writers. Por esto fue que llegamos a la conclusión de que el algoritmo que habíamos implementado era incorrecto. Ya que este test fue el que nos permitió ver que el algoritmo fallaba lo decidimos dejar implementado como un segundo test. Este test se llama 'rwlocktest2' y se corre con los mismos parámetros que el anterior.