DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

Pthreads

8 de Junio de 2016 — Sistemas Operativos

Integrante	LU	Correo electrónico
Costa, Manuel José Joaquin	035/14	manucos94@gmail.com
Coy, Camila Paula	033/14	camicoy94@gmail.com
Ginsberg, Mario Ezequiel	145/14	ezequielginsberg@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

Índice

1. Read-Write Lock

Para implementar el `RWLock` utilizamos los siguientes atributos privados:

- `mutex`
- `cv_write`: Una variable de condición para la variable `write`.
- `cv_read`: Una variable de condición para la variable `cant_reads`.
- `cant_reads`: Tiene la cantidad de lecturas que se estan realizando.
- `write`: Un booleano que cuando llega un write se pone en true. Lo usamos para evitar que los threads que realizan reads le generen inanición a los threads que quieren hacer write.

Además usamos los siguientes métodos públicos:

- `RWLock()`: Inicializa el mutex, las variables de condición, `cant_reads` y `write`.
- `void rlock()`: Genera el lock para los reads. Si llega un write deja esperando los reads para que no pueda haber inanición, luego aumenta `cant_reads`, ya que hay alguien leyendo.
- `void runlock()`: Decrementa `cant_reads` indicando que dejo de leer. Si después de eso `cant_reads` = 0 entonces envía un broadcast a todos los threads que están esperando por la variable de condición `cv_write`.
- `void wlock()`: Genera el lock para los writes. Si llegan reads o writes, quedan bloqueados hasta que se haga el `void wlock()`. Notar que lockea el `mutex` y no lo libera.
- `void wunlock()`: Hace broadcast para la variable condicional de lectura y deslockea el `mutex`

1.1. Test

Para testear el correcto funcionamiento de la clase RWLock hicimos dos tipos de test:

- El test1 permite verificar que los lock tengan el comportamiento esperado. Esto es, que los lock de escritura impidan que se puedan realizar lecturas u otras escrituras, y que los locks de lectura no bloqueen a otros locks de lectura. Para esto tenemos un par de variables `escribiendo` y `leyendo`: la primera es booleana e indica si alguien tomo el lock de escritura; la segunda es un entero que indica cuantos threads están tomando locks de lectura.

  Al acceder a la sección crítica se realizan asserts que verifican que el estado de ambas variables sea consistente con lo esperable. Notar que es necesario proteger a ambas variables con mutex al leerlas o modificarlas debido a la concurrencia. Tanto la función de escritura como la de lectura gastan un tiempo fijo de tiempo durmiendo. La función de lectura adicionalmente imprime la cantidad de procesos leyendo en simultáneo. Esto es útil para ver que efectivamente el lock de lectura no es bloqueante para los otros hilos de lectura.

Se decide si los threads son de lectura o escritura de forma aleatoria. Colateralmente este test podría encontrar deadlocks si los hubiera (en la experimentación realizada no se los encontró).

- El test2 trata de mostrar que no hay inanición. Se divide en dos partes: en la primera se quiere ver que los pedidos de lectura no pueden generarle inanición a los de escritura, y en la segunda lo inverso. Como es análogo explicamos solo el primer caso.

  La idea es poner a correr a la mitad de los threads (la cantidad total es un parámetro) haciendo lecturas, mandar un thread de escritura, seguido de muchos otros de lectura. Cada vez que un thread logra acceder a su sección crítica imprime por pantalla "Leo" o "Escribo" respectivamente, junto con su id (posición en el vector de threads creado).

  Si hubiese inanición sería esperable que el thread de escritura imprimiera último consistentemente a lo largo de las corridas (esto no pasó nunca en la experimentación). Es importante en este sentido que las lecturas tengan una duración lo suficientemente larga como para que no terminen antes de mandar el write.

2. Servidor

Para el servidor nos basamos en `backend-mono`, haciéndole los cambios necesarios para que pudo soportar más de un jugador.

Cada vez que acepta a un jugador crea un thread para atenderlo mediante la función `atenderdor_de_jugador` y guardamos la información del thread en un vector. Además creamos dos matrices de RWLock para proteger ambos tableros, y en todos los lugares donde en `backend-mono` se escribió o leyo algún casillero de algún tablero agregamos las protecciones correspondientes. Para poder saber cuando todos los jugadores mandaron el mensaje de LISTO y poder pasar a la fase de batalla, creamos la variable `cant_clientes` y la protegimos para que no la puedan modificar dos threads a la vez. Entonces cada vez que entra un cliente entra aumentamos en uno esta variable, y cada vez que un thread muere o un jugador dice LISTO esta decrementa. Además ahora a parte de solo tener la variable global peleando, cada thread tiene una variable local `listo`, lo que evita que cuando un cliente pase a pelear todos los demás lo hagan.