QUIZ # 5
PRIMITIVAS DE SINCRONIZACIÓN

Responsables:

Neiber de Jesús Padierna Pérez C.C.: 1 022 095 657
Andersson García Sotelo C.C.: 1 037 622 083

Universidad de Antioquia
Medellín
2014

INFORME QUIZ #5

1. *Escriba la estrategia de solución inicial, explicando la estructura del programa y las ideas para resolver la práctica*.

Se plantea crear una estructura de C (struct) la cual contendría la información de la cuenta, tal como su saldo y algún número de identificación inequívoco. Similarmente, se planteó crear archivos de cabecera los cuales contendría la definición y su correspondiente implementación de los métodos que se podrían aplicar sobre variables o elementos del tipo de estructura para las cuentas.

Como medida para controlar las condiciones de carrera y deadlocks que posiblemente surjan al momento de seleccionar dos cuentas para operar sobre ellas y efectuar una transacción, se propone el uso de Semáforos (en la librería semaphore.h de C) entre el conjunto de Threads.

Los parámetros enviados al método que correrán los hilos o threads de aplicación son implementados a través del uso de una estructura. Los threads tienen como objetivo realizar transacciones de dinero aleatorio entre dos cuentas, seleccionadas de la misma forma, y la información, tanto de las cuentas como de los mecanismos de exclusión mutua pasados a dicho método para los hilos, haciendo uso de la estructura ya mencionada.

Se pretende utilizar arreglos de memoria dinámica reservada en el Stack, tanto para las cuentas (account) como para los mecanismos de exclusión mutua. En otras palabras, se desea utilizar una sintaxis similar a la siguiente: struct account accounts[size_accounts];

Desde el programa de testeo, el cual está encargado de leer el archivo que contiene los vectores de prueba, se planea utilizar una función de la familia exec, la cual será encargada de invocar el programa que generará los Threads para efectuar las transferencias de montos aleatorios entre las cuentas especificadas en el vector de prueba.

2. *Explique cómo piensa utilizar las primitivas ofrecidas por el sistema operativo para evitar condiciones de carrera y posibles interbloqueos*.

El uso de las primitivas busca efectuar una transacción bancaria atómica, es decir, tanto el retiro como la consignación del dinero para ambas cuentas se tiene que realizar lo más pronto y seguro posible, evitando la presencia de condiciones de carrera entre los Threads. Partiendo de esta idea, se ha propuesto utilizar el siguiente patrón: “Para un Thread en particular, se trata de bloquear ambas cuentas seleccionadas e involucradas en la transacción. Si el bloqueo es exitoso, realizar la transacción (retiro más depósito de dinero), y finalmente liberar ambos recursos, es decir, las cuentas. Por otro lado, si no se logra realizar el bloqueo simultáneo de ambas, puede darse los siguiente: Se logró, al menos, bloquear alguna de las dos cuentas o no se logró adquirir ninguna.” En el primer caso, se debe liberar el recurso adquirido exitosamente; para el último caso, no es necesario realizar ninguna acción.

3. *Describa los problemas que encontró durante la solución de la práctica mostrando cómo se manifestaban ¿cómo los detectó? y ¿cómo los solucionó?*

- ✓ Se detectaron condiciones de carrera en vectores de prueba que se debían procesar miles de veces, así, la sumatoria de los montos finales de las diversas cuentas o el balance total final no coincidía con el valor esperado y correcto. Ello debido a que no se estaban liberando

correctamente los mutexes correspondientes a las dos cuentas asociadas a una determinada transacción en un Thread dado; esta liberación hace referencia a soltar un recurso, en nuestro caso un mutex de una cuenta, cuando no se logran obtener los necesarios para efectuar un procedimiento, es decir, si un hilo no tomaba correctamente los dos mutexes para bloquear las dos cuentas, este tendría que liberar algún mutex que haya logrado adquirir exitosamente. El problema se corrigió observando detenidamente la aplicación de este patrón, y logrando llevar a cabo su implementación en correcto orden.

- ✓ Al momento de inicializar los semáforos, cuando aún se tenía la aplicación con esta implementación, no se estaban creando o inicializando correctamente; este error surgió ya que el apuntador que recorría el vector de semáforos reservados dinámicamente, se estaba incrementando de una forma incorrecta, así, al final, se alcanzaba un error o excepción desastrosa en la implementación de malloc en la librería estándar de C, dando como conclusión un cierre inesperado de la aplicación o su caída por completo.

4. *Describa cómo fue la solución final, explicando la estructura del programa y las ideas finalmente implementadas y funcionales*.

Inicialmente, se había propuesto el uso de los semáforos para controlar la exclusión mutua de los Threads al intentar acceder a los recurso de las cuentas, posteriormente, esta implementación, o uso, de semáforos se reemplazó, y se decidió optar por el uso de Mutexes (pertenecientes a la librería pthread.h), ya que de este modo se garantiza una portabilidad mayor de los algoritmos utilizados, debido a que se está utilizando el estándar de Pthread. Vale la pena decir, que tanto la librería semaphore.h y pthread.h ofrecen métodos que brinda la posibilidad de realizar un "intento" de apropiación de un determinado recurso, y esta implementación permite darle conocer al programador el resultado del intento de bloqueo o apropiación del recurso.

Similarmente, la implementación para guardar o almacenar las estructuras que guardan la información de la cuenta y los mutexes (inicialmente, semáforos) ha sido cambiada y ha sido evolucionada a ser un arreglo dinámico de memoria pero en el Heap, usando malloc y al final, la invocación a la función free para la liberación del espacio reservado. Recordar que inicialmente se había planteado usar arreglos de memoria dinámica pero almacenados en el Stack.

Por otro lado, la llamada al sistema que se ha utilizado para arrancar la ejecución del proceso hijo encargado de crear los hilos y generar las transacciones entre las diversas cuentas ha sido, system, debido a que la implementación de esta función evita la creación de hijos o nuevos Threads, y el hilo principal o programa que ejecuta esto se bloquea hasta que la llamada a la función termine.

5. *Compare la solución inicial y la solución final*.

La solución final respecto a los parámetros de la solución inicial propuesta no difiere en gran proporción ya que la implementación de las ideas iniciales no fue cambiada a grandes rasgos. Por el contrario, se pasó del manejo de la memoria dinámica guardada en el Stack a ser guardada en el Heap ofreciendo una optimización en el manejo de la memoria mucho más alta, cumpliendo con muchos estándares de la programación en C, aunque su implementación y uso acarrea un costo de programación alto para aquellos desarrolladores que no estén acostumbrados a la administración de la memoria.

Igualmente, el uso de mutexes o semáforos no tiene gran diferencia, ya que la implementación de este par de métodos de exclusión mutua ofrece prácticamente el mismo catálogo de métodos para

llevar a cabo el control de bloqueo y concurrencia de procesos o hilos. Un buen punto a favor al usar mutexes es que al ser un estándar en pthread.h agrega un nivel de portabilidad a los algoritmos y programas que se construyan bajo este estándar.

El uso de alguna función perteneciente a la familia de exec se ha reemplazado totalmente por el uso de la función o llamado system. Este reemplazo favorece a que el hilo principal se bloqueará hasta que la función termine su llamado y de ese modo, no se tendrá que controlar concurrencia y exclusiones mutuas cuando se intenta ya sea leer o escribir en el archivo final de resultados.

Por otro lado, vale la pena resaltar pequeñas mejoras que poco a poco se les fue realizando al programa o a los algoritmos, es decir, usar búferes, punteros, métodos, parámetros por referencia en algunas funciones y demás pequeños detalles de programación en busca de una optimización y mejora en el rendimiento de las operaciones.

6. *Notas*

- ✓ Cada hilo se ejecutará hasta cumplir cierta cantidad de transacciones especificadas por el valor *cantidad_tiempo_a_correr*, que es especificado como segundo parámetro en el vector de prueba.
- ✓ Cuando un hilo falla en su intención de adquirir y bloquear dos cuentas durante una transacción, este intento se toma como una transacción fallida y no se realiza un nuevo intento.
- ✓ Cada hilo después de una transacción, sea esta exitosa o no, quedará inactivo o “dormido” durante un lapso equivalente a 100 milisegundos.
- ✓ En el archivo resultante de las pruebas, llamado *Output.txt* se registran los montos finales de dinero arrojados para cada vector de prueba. Ello se almacenará por filas, donde al lado derecho se pueden encontrar dos valores: *OK* o *WRONG*. El primero indica que el balance fue el correcto y esperado, en otro caso, aparecerá el segundo valor.