Joan Gracia Taribó - 219176
Josep Tuñí Domínguez - 216737

Operative Systems: Laboratory 4
Process Communication

· Bank Threaded

0. Esta solución se implementa sin sincronización, de modo que se crean los threads y cada uno ejecutará la función de transferir, como esta solución se implementa sin sincronización, el resultado contiene Race Condition ya que puede ser que dos threads o más, esten entrando en la misma zona de memoria simultáneamente si esos threads operan con una cuenta en común. Al final de la ejecución, se nos indica que el Banco tiene problemas, debido a la Race Condition.
Tiempo Medio: 500ms

1. Esta solución se implementa con un Global Lock, que funcionará de la siguiente forma: Cuando un thread quiere hacer una transferencia se activa el Global Lock, de modo que **ningún** otro thread va a poder hacer ninguna operación, esto hace que el programa sea mucho más lento ya que restringe la ejecución en paralelo. En algunos casos, no seria necesario que un thread este bloqueado, ya que si un thread opera con Account1 y Account2, no tiene ningún sentido que se encuentre bloqueado un thread que opera con Account3 y Account4, ya que no va a generar ningún conflicto.
Tiempo Medio: 4200ms

2. En esta solución, se implementa un lock para cada cuenta, de manera que cuando un thread hace un acceso a una cuenta, no se bloquean todos los threads, simplemente se bloquean las cuentas que participan en la transferencia de ese thread, de esa forma bloqueamos las mínimas posibles cuentas. Esto podría generar un problema de DeadLock debido a que un ThreadA podria hacer lock de Account1 y Account2 pero quedarse a mitad y solo hacer el lock de Account1. Otro ThreadB que queria hacer el lock de Account2 y Account1 hará el lock de Account2 y se esperará a que ThreadA termine para hacer el lock de Account1, el problema es que ahora el ThreadA estará esperando a que Account2 sea liberada, esto será un gran problema, ya que el ThreadA espera al ThreadB y viceversa. Esto lo podríamos haber solucionado si siempre seguimos el mismo orden de lock sin importar quien es el que recibe y quien es el que envia, si siempre hacemos lock de Account1 y luego de Account2, esto se evita. En código esto se ve reflejado en:
if (acc_from->id > acc_to->id) { m1 = &to->m; m2 = &from->m; }
Tiempo Medio: 2700ms

3. En esta solución, en vez de locks se implementan monitores que al ser objetos de más alto nivel que el Mutex Lock, es un poco más lento. La idea es la misma que en la solución 2 y el DeadLock se evita de la misma forma que en la solución 2.
Tiempo Medio: 3400ms

4. En esta solución usamos un lock para cada dos cuentas, es decir, que si nuestro ThreadA hace una transferencia de Account1 a Account2, tendremos bloqueadas las operaciones entre Account1 y Account2, esto puede generar Race Condition ya que un ThreadB puede intentar una transferencia de Account3 a Account1, como esta operación no está bloqueada, se podrá ejecutar nuestro ThreadB causando Race Condition. En esta solución también podemos tener DeadLock, que lo solucionamos con la línea:
if (acc_from->id < acc_to->id) pthread_mutex_lock(&bank_locks[acc_from->id][acc_to->id]);
else pthread_mutex_lock(&bank_locks[acc_to->id][acc_from->id]);
Tiempo Medio: 1300ms

Finalmente, se puede llegar a la conclusión de que los más rápidos, son las soluciones 0 y 4, pero esto de nada nos sirve, ya que esas soluciones tienen Race Condition. Por tanto, la mejor solución será la 2, ya que de las que no tienen Race Condition, es la más rápida.

Joan Gracia Taribó - 219176
Josep Tuñí Domínguez - 216737

·Bank multi-process

1. En esta solución sincronizamos bank_transfer.c usando un file lock, para hacerlo ejecutamos desde el terminal ./Bank_Transfer 1. Con tal de evitar una ejecución con race conditions, hacemos los locks de la cuenta de origen y destino en orden ascendente (si el ID de origen es mayor que el de destino, bloqueamos primero el destino, y viceversa), de manera que podemos hacer la operación de sumar y restar dinero sin posibilidad de race condition. Una vez completada la transferencia, hacemos el unlock de ambas cuentas. Después de las 1000 operaciones de cada uno de los 250 procesos, obtenemos un tiempo de ejecución de aproximadamente 1600 ms.

2. En esta solución, conseguimos la sincronización mediante semáforos, ejecutando ./Bank_Transfer 2. Para evitar race conditions, iniciamos el semáforo inmediatamente después de inicializar las cuentas bancarias, y en cada operación el programa espera (bloquea) mediante la función sem_wait(), retira una cantidad de dinero de la cuenta de origen, y la ingresa en la de destino. Una vez completada la transferencia desbloqueamos ambas cuentas mediante sem_post(). Después de las 1000 operaciones de cada uno de los 250 procesos, obtenemos un tiempo de ejecución de aproximadamente 2300 ms, algo más lento que usando file locks.