P4: Process communication

Sergio López and Xavier Marquès

March 2021

2 Bank threaded

0. No synchronization
   In this solution, since there's no sync between the threads, two or more thread can access to the same data and change it at the same time. This provokes that all threads competes between them to access the data and producing a race condition. Therefore, the order of execution of the threads scheduling affects to the result producing, in our case, a change in the total sum of all banks.

1. Global Lock
   This solution is correct because it takes care of the race condition, since there's a global lock and then the threads cannot enter the critical section at the same time. But it has a disadvantage since there's only one lock, all threads must wait until the lock is released and that produces busy waiting consuming to much time, so it's not optimal.

2. Account Lock
   This solution is correct because using a lock for every account, it avoids the overlapping of a deposit or a withdraw of money on each of ones. That affects to the threads and it avoids also that another thread can access it. But it could happen that one thread locks one account and another thread, at the same time, locks the account that the first thread needed and, even more, this second needs the account locked by the first, producing deadlocks. So the solution to this deadlocks is to force all threads to lock always the account with the lesser id.

3. Account Monitor
   This solution it seems the best solution implemented because it controls the race condition since for each account it is waiting until that account is not busy and it is able to operate the next operation. But it is not the fastest solution because it has some more milliseconds than using locks (approx 100ms). The same as before, this solution can have deadlocks, so is needed to take care about the order.

4. N x N locks

3 Bank Multi-Process

In this part, we're going to reuse some functions provided in bank_threaded.c : do_Something, withdraw, check and deposit; and we're going to add 3: child_transfer , trans_sem and trans_file_lock.
First of all, in the main we'll need to create 200 process with fork and in the child one, apply the function child_transfer. In this function, each process is going to do some random transfers between accounts but we need to apply some synchronization solution to avoid race conditions. For this part, we're demanded to do 2: file lock or semaphore solution.

1. **Using file lock**
   If the input is 0 or is there's no input (by default), we apply the function trans_file_lock that is going to apply the transfers using this file lock solution. First we open the file and create 2 accounts to store the information. Then we need to lock the files. Since we need two locks, one for the sender and one for the receiver, if we don't force the system an order for applying the locks, we can have dead locks. So, we force the system to lock always the bank with lower index or id. Then, we can apply the lseek to search for the account in the file and read to store the account information for both account. Next step is to performance the withdraw and deposit operations and finally use again lseek and write to both accounts to store everything back to the file. Do no forget to close the file and the end. The benchmark of this solution it is around 15 ms.

2. **Using semaphores**
   If the input is 1, then we use the function trans_sem that applies semaphores sync solution for the transfers. First we open the file and wait until semaphore becomes positive. When is the case, then, the same way as file locks, we search and read the information, apply the operations and get back the information to the file. Finally we make a signal to the semaphore that we're done and close the file. The benchmark of this solution it is around 800 ms.

For the bank_sum , we apply a for loop to search and store the information for each account and then we use the sleep function to do this loop every 0.5 seconds.