Seminar 3: Thread Synchronization
Sistemes Operatius

1 Introduction

We are going to practice with thread synchronization. Be sure you understand the difference of the wait and signal operations for semaphores and monitors. You should have an idea of how a lock is implemented. You should know how to implement a lock without busy waiting using monitors and semaphores (seen in class).

2 Bank Account Exercise

We are going to implement a threaded bank server able to deal with millions of operations simultaneously. We are going to do it step by step. Here follow some declarations:

```
pthread_mutex_t global_lock;
bool global_is_busy = false;
pthread_cond_t global_cond_free;

typedef struct str_acc {
 int id; double balance;
 bool busy;
 pthread_mutex_t lock;
 pthread_cond_t cond_free;
} Account;

Account bank[1000];

bool withdraw(Account* ac, float amount);
void deposit(Account* ac, float amount);
void transfer(Account* from, Account* to, float amount);
```

Program the **withdraw** function in these different ways:

1. Synchronizing using only the global_lock. Discuss if this solution is concurrent and if it has busy waiting.
2. Add to the previous solution the fact that the operation returns a boolean whether it was successful or not (there was sufficient balance in the account).
3. Synchronizing using monitors and only the global_lock, the global boolean and the global condition. Discuss if this solution is concurrent and if it has busy waiting.
4. Synchronizing using only the lock of each account.
5. Synchronizing using only the boolean, the lock and the condition of each account.

Assume we have programmed the deposit function likewise (last exercise). Now we are going to implement a **transfer** function always performing the withdraw first for security and consistency reasons (better not give money before ensuring it can be withdrawn). Program the transfer function in the following ways:

1. Reuse the widthdraw and desposit functions that you did in the last exercise assuming they are correctly synchronized. Discuss if this solution could have deadlocks.
2. Now don't reuse widthdraw and desposit. Program the transfer first checking that there is the money available and ensure the transfer operation spends a minimal time actually interchanging the money for consistency and security reasons.
3. Try to break the possible deadlock addressing the fact that you can force a particular order of accessing resources: in this case a resource refers to the mutual exclusion. To solve this we are going to assume that each account has a unique identifier, an integer for example.

2.1 Delivery Exercises

1. Program a bank maintenance operation in which you will need also to provide the code for withdraw with the new feature included. No thread can be doing any operation and all operations must be put on hold till the process is finished. The maintenance operation will subtracted subtract 0.1 euros for all the accounts.