Lab Session 4: Process Communication
24296 - Sistemes Operatius

1 Introduction

The objective of this lab session is to understand Process communication and synchronization. You will need to deliver a compressed file with:

- A document answering the questions of the lab session and reporting benchmarking.
- Code files "bank_sum.c", "bank_transfer.c" which must compile, if not you will not be evaluated.
- The document is mandatory, is the first thing that will be evaluated after checking that the code compiles. If you don't submit a document your lab session will not be evaluated.

2 Bank Threaded

We have provided the code for a threaded bank server (file bank_threaded.c) that performs thousands of concurrent transfer operations, leaving the total bank sum intact if there was no race condition.

Synchronization is implemented with 5 solutions that can be selected from the command line adding a number for each one **0:** for a threaded bank with no synchronization. **1:** for the solution with a global lock. **2:** for the solution using a lock for every account. **3:** for the solution of using a lock, a boolean, and a condition for every account. **4:** a solution that uses a lock for every pair of accounts.

Comment in the document all solutions from 0 to 4. Which ones have race conditions and why? Which ones could have deadlocks and how was that solved? Which ones are more efficient? For this last question, you need to know that locks in Linux are implemented with monitors. You always need a look in semaphores or monitors to access (as shortly as possible) the shared variables; so what we explained in class of the need of test_and_set being atomic still holds.

3 Bank Multi-Process

The previous bank_threaded.c program also creates a binary file at the end bank.dat that you will use in this exercise. Be sure that you create it using a correct solution.

You will now program two shell commands that will perform the same operations than the bank_threaded.c but with processes. Here the commands description:

1. bank_transfer.c : will perform 10000 transfer operations in the style of the previous threaded version, directly in the bank binary file bank.dat using read, write and lseek.
2. bank_sum.c : will perform periodic sum operations every half a second of all the account amounts to compute the total bank revenue, checking that it is correct: it should always sum to 1000.

To manipulate the file bank.dat use the system calls open, read, write and lseek seen in class:

```
int open(char* name, O_RDWR);
int read(int fd, void *buf, int nbyte);
int write(int fd, void *buf, int nbyte);
int close(int fd);
lseek(int fd, int bytes, SEEK_SET)
```

To synchronize we are going to use two solutions: file locks and named semaphores. Select the synchronization method through an argument in the command line as we did in the previous bank_threaded.c file. Here the signatures of file locks as provided in myutils.h:

```
int file_lock(int fd, int start, off_t len);
int file_unlock(int fd, int start, off_t len);
```

The header file semaphore.h provides functionality for semaphores in Linux. We are going to use the so-called named semaphores to synchronize processes.

Named semaphores exist as files in the system and all processes can access them. When you create a semaphore (now called open) you need to specify a name as a string (char*). Here the declarations:

```
#include <semaphore.h>
sem_t* named_mutex;
sem_open("named_mutex", O_CREAT, 0600, 1);  // inistialized to 1 for a mutex (last parameter)
sem_wait(named_mutex);
sem_post(named_mutex);      // equivalent to sem_signal
```

Explain and benchmark both solutions with file locks and named semaphores in the document as we did in the threaded version of previous section.