Practice Labs 4a/5a – Pthreads Mutex locks

Operating Systems Comp Sci 3SH3 Term 2, Winter 2018

Dr. Neerja Mhaskar

Practice Labs are not mandatory, they are primarily designed to help you with your Lab tests and assignments.

Lab Format: Practice labs will be posted a day before or on the day of the lab on the course website. You can choose to solve it beforehand and come in with your solutions and check the correctness of your solution with your TA.

The TAs will also be available to answer any questions you might have on your assignments.

Solutions to practice labs will not be posted online.

Outline

Banking System Problem

Consider a simple banking system that maintains bank accounts for its users. Every bank account has a balance (represented by an integer variable **`amount`**). The bank allows deposits and withdrawals from these bank accounts, represented by the two functions: **`deposit`** and **`withdraw`**. These functions are passed an integer value that is to be deposited or withdrawn from the bank account. Assume that a husband and wife share a bank account. The husband only withdraws from the account and the wife only deposits into the account using the **`withdraw`** and **`deposit`** functions respectively. Race condition is possible when the shared data (**`amount`**) is accessed by these two functions concurrently.

In this lab you are to write a C program that provides a critical section solution to the Banking System Problem using mutex locks provided by the POSIX Pthreads API. In particular your solution needs to do the following:

Steps to follow:

1) Take two command line arguments. First argument is the amount to be deposited (an integer value) and the second argument is the amount to be withdrawn (an integer value).
2) Create a total of 6 threads that run concurrently using the `Pthreads` API.
    a. 3 of the 6 threads call the `deposit()` function, and
    b. 3 of the 6 threads call the `withdraw()` function.
3) Create the threads calling the `deposit()` function using the `pthread_Create()` function. While creating these threads you need to pass

the thread identifier, the attributes for the thread, `deposit()` function, and the first integer command line argument `argv[1]` (which is the amount to be deposited).

4) Similarly, create the threads calling the `withdraw()` function.
5) To achieve mutual exclusion use mutex locks provided by `the Pthreads` API.
6) You are to provide print statements that output an error message if an error occurs while creating threads, mutex locks semaphores etc.
7) Your program should print the value of the shared variable `amount,` whenever it is modified.
8) Finally, the parent thread should output the final `amount` value after all threads finish their execution.
   Make sure you use pthread_join() for all the threads created. This will ensure that the parent thread waits for all the threads to finish and the final amount reported by main is correct for every execution of the program.

Notes:

1. Look at the Pthreads example in the textbook Figure 4.10, page 174 to see how command line arguments are passed and used in the function called by a thread.
2. See lecture slides on Chapter 5 for using mutex locks provided by the `pthreads` API.
3. You might see that the amount is negative. This could happen if the threads calling the withdraw function are scheduled to run on the CPU before the deposit function. This is OK for this practice lab.

Sample Output: ./lab5a6a 100 50

Withdrawal amount = -50

Withdrawal amount = -100

Withdrawal amount = -150

Deposit amount = -50

Deposit amount = 50

Deposit amount = 150

Final amount = 150