CSC 325 Lab 11

pthread mutexes

Table of Contents

This is a **partner** lab which may be completed on **either** VM.

1. Lab Goals

1. Gain more experience in C system programming.

2. Learn about multithreading with competition synchronization in C.

2. Lab Overview

Concurrency is about *dealing with* multiple things at once. Parallelism is about *doing* multiple things at once. An application can be concurrent but not parallel, which means that it processes more than one task at the same time, but no two tasks are executing at the same time instant.

Parallel programs enable users to more fully utilize the multi-core structure of modern CPUs. Pthreads (POSIX Threads) are a set of functions that support applications with multiple flows of control, called threads, within a process. Multithreading can be used to improve the performance of a program. However, multithreading often introduces race conditions which require synchronization. Failure to synchronize, or improper synchronization, can produce wrong results or even deadlock (loss of liveness.) In this lab we will be using GCC and pthreads with mutex synchronization.

Synchronization can be categorized into 2 types:

1. Cooperation: task A must wait for task B to complete some specific activity before task A can continue its execution, e.g., the producer-consumer problem.
2. Competition: two or more tasks must use some resource that cannot be simultaneously used, e.g., a shared counter.

This lab is an example of competition synchronization. Competition synchronization is usually provided by a **mut**ually **ex**clusive access primitive known as a **mutex**.

3. Handy References

- Read and understand textbook chapter 14.3.1 Mutual Exclusion
- View the libc manual pages from the command-line:

```
$ man 7 pthreads
$ man 3 pthread_mutex_lock
$ man 3 pthread_mutex_unlock
```

- If the lock/unlock man pages aren't present on your system, you can install them:

```
$ sudo apt update
$ sudo apt install glibc-doc
```

4. Lab Starting Point Code

4.1. Getting the Starting Point Code

Download the starter files from Canvas:

```
Makefile xfer.c
```

4.2. Starting Point Code

The files included are:

- Makefile: a complete Makefile to build the money transfer application
- xfer.c: starter code for the money transfer application

5. Lab Requirements

Task A

- Compile and link your source file by typing
  - make
- Run your application by typing
  - ./xfer
- Answer the following questions
  - The two accounts begin with $1000 each, so the total of the initial balances is $2000. Run the program, which makes one million random transfers between the two accounts. Is the total of the final balances still $2000, or is it now greater or less? Why?
  - Run the program several times. Do the total of the final balances change? Why?

Task B

- Use pthreads mutexes to provide competition synchronization. You should use these library functions:
  - pthread_mutex_lock()
  - pthread_mutex_unlock()
- Use the UNIX man command for the function documentation
- Recompile and run the program several times to verify that your synchronization is working correctly, i.e., that the total of the final balances is now always $2000.
- Answer the following questions
  - Did your mutexes work properly the first time? Why or why not?
- Describe what approach you used to prevent deadlock.

6. Tips

Read Chapter 14.3.1 Mutual Exclusion

7. Submitting your Lab

7.1 What to submit

- xfer.c (with added pthreads mutexes from Task B)
- Makefile
- Screenshots of your Task A and Task B output
- A PDF document with your responses to the questions for Task A and Task B

Your application must compile, assemble, and link without warnings or errors.

Zip your folder and submit it to Canvas.

7. Example usage

```
$ make clean; make
rm -f xfer.o
gcc  -Wall -g  -c -o xfer.o xfer.c
gcc -g -pthread -o xfer xfer.o
$ ./xfer
Account 0 initial balance: 1000.00
Account 1 initial balance: 1000.00
Total initial balances: 2000.00
Account 0 final balance: -5664.00
Account 1 final balance: 7664.00
Total final balances: 2000.00
```