1 Multithreaded Programming with Pthreads

Multithreading (MT) is a technique that allows one program to do multiple tasks concurrently.\(^1\) The purpose of this project is to give you some experience writing a multithreaded program using the POSIX thread library.

There are three different definitions for thread libraries competing for attention today: Win32, OS/2 and POSIX. The first two are proprietary and limited to their individual platforms. The POSIX specification (IEEE 1003.1c, aka *Pthreads*) is intended for all computing platforms. The POSIX standard defines the API and behavior that all Pthreads libraries must meet. Therefore, in this project, you are expected to use the POSIX thread library.

1.1 Life Cycles of Threads

The fundamental paradigm of threads is the same in all of the libraries. In each of them, the program starts up in the same fashion as single threaded programs always have — loading the program, linking in the dynamic libraries, running any initialization sections and finally starting a single thread running main(). For an MT program, all the same things occur and one of the libraries linked in will be the threads library. The main function will then be free to create additional threads as the programmer sees fit.

In the simplest case, you can call the *pthread_create()* function with a function to run and an argument for the function to run on. Should you desire to pass your start routine more than a single argument, you must create a structure to pass in multiple arguments.

A thread is exited by calling *pthread_exit()* or simply returning from the original function. Beyond the actual call to the create function, there is no parent/child relationship — any thread can create as many threads as it pleases and after creation there will be no relationship between the creator and createe.

\(^1\)Some of the descriptive material about threads for this project comes from the book “Multithreaded Programming with Pthreads” by Bil Lewis and Daniel J. Berg.
Each thread has a thread ID (TID) which may be used to control certain aspects of that thread. POSIX TIDs are of type `pthread_t`.

### 1.2 Returning Status and Memory

Sometimes you specifically want to wait for a thread to exit. Perhaps you’ve created 20 threads to do 20 pieces of a task and you can’t continue until they are all finished. One method is to call the `pthread_join()` function on each of the desired thread IDs. The caller will block until each of the specified threads has exited.

In addition to waiting for the threads to exit, the caller can receive a status from the exiting threads. To ensure no deadlocks can occur, it makes no difference if the waiting thread calls the `join` function first or if the exiting thread calls the exit function first.

Not all pthreads can be joined. At creation time, you can specify that you intend not to join a thread, creating a so-called detached thread. You can specify that you do intend to do a join, creating a nondetached thread. Any thread can call `pthread_join` on any other nondetached thread, but the exiting thread can be joined only once. The main thread is always a nondetached thread, so it can be joined.

In order to write any kind of concurrent program, you must be able to synchronize the different threads reliably. Without synchronization, two threads will start to change some data at the same time, or will overwrite the other. To avoid this disaster, threads must reliably coordinate their actions. This means that all shared data must be protected!

### 1.3 Synchronization Variables

There are two basic things you want to do. First, you want to protect shared data. This is what locks do. Second, you want to prevent threads from running when there’s nothing for them to do. You don’t want them spinning, wasting time. This is what semaphores and condition variables are for.

#### 1.3.1 Mutexes

The mutual exclusion lock is the simplest and most primitive synchronization variable. It provides a single, absolute owner for the section of code (thus a critical section) that it brackets between the calls to `pthread_mutex_lock()` and `pthread_mutex_unlock()`. The first thread that locks the mutex gets ownership, and any subsequent attempts to lock it will fail, causing the calling thread to go to sleep. When the owner unlocks it, one of the sleepers will be awakened, made
runnable and given the chance to obtain ownership. It is possible that some other thread will call `pthread_mutex_lock()` and get ownership before the newly awakened thread does. This is perfectly correct behavior and must not affect the correctness of your program.

1.3.2 Semaphores

A counting semaphore is a variable that can increment arbitrarily high, but decrement only to zero. A `sem_post()` operation increments the semaphore, while a `sem_wait()` attempts to decrement it. If a semaphore is greater than zero, the operation succeeds; if not, then the calling thread must go to sleep until a different thread increments it.

2 Simulation of a Server Program

Once again, this is an individual project to be done by each of you without any outside help except for books and my multi-threaded program (available at my website).

You will simulate a simple web server application in this project. The web server maintains six integer variables, each initialized to value 50. The main program will read a series of lines from the keyboard, one line at a time. Each line has information about one client. The line has three char strings: (i) first string is the name of the client, (ii) the second string is the input file (a text file) that has all the commands this client will give to the server, and (iii) name of the output file which will contain all the results returned by the server on executing the commands given by the client.

There can be a maximum of 15 clients and the end of the client list is signified by the user typing `control D`. Each client’s requests are handled by a newly created thread.

An example:

```
John-Doe input1.txt output1.txt
Eric-Smith in1 myoutput
control D
```

For the example above, create two threads, each thread taking care of one of the two clients. Thread 1 will read the file `input1.txt`, truncate the output file `output1.txt` to zero length, execute each command in the file `input1.txt`, and store the results in file `output1.txt`. Similarly, thread 2 will read the file `in1`, truncate `myoutput` to zero length, execute each command in the file `in1`, and store the results in file `myoutput`. In each output file, identify the name of the client.

Now let us look at each input file.
Each input file has series of lines and each line represents one command. The valid commands are of the form:

I n1 x
M n2 n3 y
R n4 n5 n6
A
Q

Here, n1, n2, n3, n4, n5 and n6 are all numbers in the range 0..5 representing the six variables. x and y are integer values. The commands are explained as follows:

- **I n1 x**
  Increase the value of server variable n1 by value x. x can be either positive or negative. n1 is a number in the range 0..5. When x is negative, the value of variable n1 decreases.

- **M n2 n3 y**
  In this case, we will subtract n2 by y and add y to n3. This has the effect of moving y units from n2 to n3. n2 and n3 are integers in the range 0.. and y can be either positive or negative. This is an atomic operation.

- **R n4 n5 n6**
  Read the values of variables n4 n5 and n6 and print the values. Identify the variables and then print the values.

- **A**
  Read the values of all variables and print them.

- **Q**
  Quit. End of all commands from this client.

For example, the following is a legally correct input file:

```
I 3 -20
M 2 1 14
R 3
R 1 4
M 3 4 10
```
At the end, before the main thread terminates, list the values of all the six variables.

When accessing any of the six variables, assume that the time to perform the access operation is at least 100 milliseconds. That is, if you are getting a lock on a variable, wait for 100 ms after performing the operation on the variable and then you can release the lock.

Think carefully about the following questions:

- What global variables are required?
- What kind of synchronization between threads is needed?
- When and where do you need mutex?

3 Hand-in Instructions

Name your program p1.cc. Turn in a well-written program (under the name p1.cc) with all the documentation needed including the design description, shared data, program structure and other various decisions made by you. Your program should be well structured and well documented. You may write your code in C or C++. In addition, to the source file, include a copy of the input files and the output files. Thus, create a tar file that contains the source file, all the input files, the series of commands given to the main program (in a file called stdin) and the output files as updated by your programs. Upload your tar file using webct.

For example, if the input files are i1.txt, i2.txt, i3.txt and the output files are o1.txt, o2.txt, and o3.txt, then I will do the following:

tar cvf turnin.tar p1.cc stdin i?.txt, o?.txt
upload the file turnin.tar.