CSE 320 Spring 2019 HW #4

April 11, 2019

Deadline: April 25, 23:59 Stony Brook time (Eastern Time Zone)

1 Introduction

The goal of this homework is to practice writing multithreaded code. It will also expose you to the concepts of synchronization using semaphores to access shared data structures. You need to implement three small programs, modify your previous homework to use threads, and build a new program. As usual, we highly recommend spending some time thinking about your code. You may want to design your code first, think about possible corner cases, and probably even write it down on a piece of paper. It may significantly reduce the amount of time that you may spend on writing the code itself.

The homework is structured as follows: Part I consists of readings, Part II is three short programming exercises to practice thread creation and semaphores, Part III is modifying your previous homework, and Part IV is a bigger programming exercise with threads.

2 Part I

For this homework, it is crucial to start early and read the textbook. You should read *Chapter 12* from the textbook and corresponding man pages for the functions used in this homework. The web-links below point to the material that may be helpful for this assignment. The second and the third reference, you should be able to access the PDF files for free if you are using the university's network.

`https://en.wikipedia.org/wiki/Quicksort`
A Parallel Quicksort Algorithm
`https://dl.acm.org/citation.cfm?doid=366622.366644`

We also recommend to go over the readings about locking/synchronization/threads:

`https://www.geeksforgeeks.org/use-posix-semaphores-c/`
`http://man7.org/linux/man-pages/man3/pthread_spin_lock.3.html`
Mutex versus Spinlock
pthreads

http://www.mit.edu/people/proven/IAP_2000/index.html

3 Part II

In this part, you need to implement three small programs that will get you started with writing multithreaded code and give you some practice with using semaphores.

In the first program, you need to build an application that should print Nth Fibonacci number using separate thread. The program should be called `hw4_progfib` and N should be obtained from the command line arguments. In this program you need to create a new thread using `pthread_create()` function and find Nth Fibonacci number using that thread. As before, print only the resulting number to the output. To run a program that prints fourth Fibonacci, one will type the following in the terminal:

```
$ make progfib
$ ./hw4_progfib 4
```

In the second program, you need to write an application that will count the number of occurrences of numbers zero, one, and two in some arrays. You will be given three files with following names: "file1.dat", "file2.dat", and "file3.dat". Each file will contain two lines. The first line contains a number N. The second line contains N space-separated numbers. Your task is to create three threads such that each thread will handle one and only one of the files. Each thread should count the number of occurrences of numbers zero, one, and two on the second line and increment global counters. For storing that information you need to create a global data structure called `number_counter`. Thus, each thread will share the same data structure and should modify that data structure within the thread function body. You need to print only three space-separated numbers to the output that represent the number of occurrences of numbers zero, one, and two respectively. To run that second program, one will type the following in the terminal:

```
$ make progcnt
$ ./hw4_progcnt
```

In the third program, you need to implement *Parallel Quicksort* algorithm using *pthreads*. Your program should start with reading the file. The file name is given as the command-line argument. That file contains two lines: the first line contains number N and the second line contains N space-separated numbers. Your program should sort numbers from the second line using the parallel quicksort algorithm and print the result as a sorted sequence of space-separated numbers. To run the third program that sorts numbers stored in the file with name "filename.dat", one will type the following in the terminal:

```
$ make progqsort
$ ./hw4_progqsort filename.dat
```

4 Part III

We have discussed what are some reasons to choose a multithread approach versus a multiprocess approach. However, it is also interesting to see the difference from the programming perspective. In this part, you need to modify your previous homework #3. As you may recall, we had `fork()` to create new processes. In this programming exercise instead of creating new processes, you will need to create new threads to serve the same purposes. Thus, your goal is to re-write part of your homework #3 but instead of using `fork()` and creating new processes, now you need to use `pthread_create()` and create new threads. You need to re-implement the following commands from the previous homework:

- List of commands:
  - help (this stays same)
  - hire X
  - list
  - fire TID
  - fireall
  - exit

Hint: You may want to read about `pthread_detach()` function.

5 Part IV

In this part, you need to write a bigger program called `museum_overseer` that will imitate a simple garbage collector that runs in a separate thread. Now let's phrase that in terms of art!

In the beautiful city of New Stony Brook, there are five art museums. In this city, there are many people who enjoy going to art museums on a daily basis. Every morning all of the museums open their door to the public and in each of the museums there will be one person who opened it and we consider that person as the first visitor. Usually, art museums close in the evening, but in the city of New Stony Brook, an art museum cannot close if there is at least one visitor. However, once it is evening and there are no visitors in the art museum, it can close its doors. And now your goal is to make sure that eventually all museums are closed.

Now let's discuss some internals of this program. First, you need to create some data structure that will represent art museums and will keep a count of the number of visitors in each museum. That data structure should be called `museum_ds`. Then you need to implement several functions:

- `visitor_in(int N)`. This function imitates that a visitor entered museum number N.
- `visitor_out(int N)`. This function imitates that a visitor left museum number N.

- `museum_info()`. This function prints information about the current state of the museums. More information about this function will be provided later.
- `museum_close(int N)`. This function imitates closing museum number N.
- `museum_clean()`. This function kicks out all visitors from all museums if there are any, closes all museums, and frees all the memory.

museum_info(). This function should print information about the current state of each museum in the following form:
<MUSEUM NUMBER>:<NUMBER OF VISITORS>:<STATUS>.
For example, if museum number five has four visitors, then the respective line for that museum will look as "5:4:OPEN". The museum can either be "OPEN" or "CLOSED".

Now, once all those functions are implemented, we finally can proceed to our version of the garbage collector. You need to implement a shell that supports the following commands:

- `in N X`. This command adds X more visitors to the museum N.
- `out N X`. This command removes X visitors from the museum N.
- `info`. This command prints information about the current state of the museums in the format described earlier.
- `start`. This command starts the garbage collector.
- `exit`. This command exits the program.

The "`in N X`" command should create X threads where each thread will call `visitor_in(int N)` to modify number of visitors at an art museum number N. You need to use for-loop to create these threads and you should not join your threads within that for-loop.

The "`out N X`" command should create X threads where each thread will call `visitor_out(int N)` to modify number of visitors at an art museum number N. You need to use for-loop to create these threads and you should not join your threads within that for-loop.

The "`info`" command should create a thread that will internally call the `museum_info()` function to print the information about the current state of the museums.

Finally, the `start` command should create a thread that will check every three seconds if there are open art museums with zero visitors. If there is such an art museum then that thread should close that art museum using `museum_close(int N)` function.

Don't forget to call `museum_clean()` before exiting your program.

6 Notes and Requirements

No zombies. No memory leaks. No crashes.

You need to create a `Makefile` that will allow us to compile your code and put the documentation for your programs into README file. The documentation should be written in such way, that a person who will read it should be able to understand how to compile/run your program, what it is about, and high-level details about your implementation.

Your code should be thread-safe. We will use many threads during testing of your code to verify the correctness of your submission.

You need to follow the provided specification, especially the required names for executable files and data structures. Failing to do so may result in a severe penalty for the homework grade even if the difference is just one symbol.

If you are asked for some output then you need to print only what was asked and nothing else. You should create a function called `cse320_print(char* msg)` that you need to use for any required output. The body of that function may be modified during grading.

7 Submission

Please click on the link below that will set up a repository for you for this homework. In case, it will take more than a few minutes please contact me so I can set up the repository for you manually.

https://classroom.github.com/a/oe5bE6uU

8 Extra Credit I

For this extra credit, you need to implement the following parallel algorithms using pthreads and report runtime performance gain/loss with varying number of threads:

- Parallel Mergesort,
- Parallel Samplesort,
- Parallel partitioning,
- Parallel matrix multiplication, and
- Parallel algorithm of your choice.

As before, you need to demonstrate your extra credit work in-person before the last lecture of this semester.

9 Extra Credit II

For this extra credit, you need to implement three versions of second program in the Part II using different types of synchronization mechanisms:

- `pthread_mutex_t` mutexes,
- `sem_t` semaphores, and
- `pthread_spinlock_t` spinlocks.

You also need to measure running time for your program while varying input size and make a write up to report your findings.

As before, you need to demonstrate your extra credit work in-person before the last lecture of this semester.