

CS450/550: Parallel Programming

Assignment 2: Programming with PThreads

Preliminaries

You are expected to do your own work on all homework assignments. You may (and are encouraged to) engage in discussions with your classmates regarding the assignments, but specific details of a solution, including the solution itself, must always be your own work. See the academic dishonesty policy in the course syllabus.

Submission Instructions

You should turn in an electronic archive (.zip, .tar., .tgz, etc.). The archive must contain a single top-level directory called CS450_aX_NAME, where “NAME” is your NAU username and “X” is the assignment number (e.g., CS450_a1_mg1234). Inside that directory you should have all your code (no binaries and other compiled code) and requested files, named exactly as specified in the questions below. In the event that I cannot compile your code, you may (or may not) receive an e-mail from me shortly after the assignment deadline. This depends on the nature of the compilation errors. If you do not promptly reply to the e-mail then you may receive a 0 on some of the programming components of the assignment. Because I want to avoid compilation problems, it is crucial that you use the software described in Assignment 0. Assignments need to be turned in via BBLearn.

Turn in a single pdf document that outlines the results of each question. For instance, screenshots that show you achieved the desired program output and a brief text explanation. If you were not able to solve a problem, please provide a brief write up (and screenshots as appropriate) that describes what you tried and why you think it does not work (or why you think it should work). You must provide this brief write up for each programming question in the assignment. The questions may provide test scripts to (try) and validate your programs. When test scripts are used, show the output of both the program and the testing script (this is 2 executions, because the script takes your program output as input). I will demonstrate this in class.

This pdf should be independent of the source code archive, but feel free to include a copy in the top level of that archive as well. Let me know if there are problems uploading multiple files to BBLearn.

Grading

All questions are weighted equally.

Notes Regarding All Questions

All questions require mutual exclusion. In questions that require calling `usleep`, do not sleep inside a critical section. Also, do not use global variables. Instead, declare shared variables in `main()` and pass them in to functions using a struct.

Question 1: Generating a sequence using Pthreads

Having multiple threads call a function, like `do_work(...)`, will have a non-deterministic execution. Write a program with 3 threads that call a function called `do_work`. Each thread will be responsible for generating a number and appending it to a buffer. Thread 1 generates number 1, thread 2 generates number 2, and thread 3 generates number 3. These numbers assigned to the threads are passed in as arguments. Each thread will store its value in a shared buffer of integers having a size of 3 elements called “buffer”. When

the third element is added to the buffer by either thread 1, 2 or 3, then it checks to see if the sequence is “123”. If not, it clears the buffer and the threads *try to generate the sequence again*. Once the total number of sequences of “123” reach 10, **the threads should exit the function and join with the main thread**. Each time you generate “123”, it should be printed to the screen. You should also print out the total number of tries it took to print “123”. For example, keep track of the total number of other sequences generated (including 123), as well: 321, 213, etc. You *must use* the `usleep(500000)` function once at each iteration (after each time a thread updates the buffer, but not in a critical section). Also, each time a thread adds its element to the buffer, it should print out its corresponding number.

Below is example output at the end of the program’s execution. Ensure that your program produces the *exact same output formatting*; you will see why in a moment.

```
...
My id: 2
My id: 1
My id: 2
My id: 3
123
Total sequences generated: 38
Number of correct sequences: 10
```

Testing:

You will test that your program appears to work correctly based on the output. I provide you with a python script that tests the output of your program: `A2sequence_test.py`. After you believe your program works correctly, you can test your program with the command as follows:

```
./question1_NAME | python A2sequence_test.py
```

This python script (python 2) will output a few possible errors, but it is not an exhaustive approach to checking the correctness of your program, and it will not be able to detect race conditions. Some errors include: not outputting the thread id at each iteration, or an incorrect number of thread ids output. Make sure to print to standard out and not standard error (i.e., be sure to use `printf`).

Submission:

Submit your assignment in a subdirectory called `question1`. The name of your c program should be: `question1_NAME.c`, and should compile as follows: `gcc question1_NAME.c -o question1_NAME`. This may vary, for instance, if you need `-lpthread` on your system.

In the pdf file, include a screenshot of the last 10 lines or so of your program that demonstrates that it works correctly. If it doesn’t work correctly, please document it as well, with screenshots and a text description. For instance “my program never stops running and it should stop because the counter value is 10 and therefore...”.

Question 2: Sequence competition

There is a competition to generate the sequences faster between two sets of 3 threads (6 threads in total). This extends question 1, so reuse your code and output the same information. Another 3 threads will generate the values 4,5,6, just like in question 1. Whichever set of 3 threads generates 10 sequences first, either “123” or “456” wins. Furthermore, whichever set of 3 threads wins, must tell the other set of threads that they have won and stop their execution. Only 1 thread must tell the other set of threads. Make functions `do_work` for the first 3 threads, and `do_work2` for the second set of 3 threads. Use two buffers, named “buffer” and “buffer2”, respectively. You should also print out the total number of tries it took to print “123” and “456” and the total number of correct sequences each set of 3 threads generated. One should be 10 and the other should be less than 10. The set of 3 threads that won should print to the screen only once: “Team x won!”, where x is 1 or 2. You must use `usleep(500000)` for both sets of threads at each iteration (but do not sleep in a critical section).

Below is an example output where team 2 wins (at the end of the output):

```
...
123
My id: 6
456
My id: 2
Team 2 won!
My id: 3
My id: 1
Total sequences generated team1: 46
Number of correct sequences team1: 7
Total sequences generated team2: 45
Number of correct sequences team2: 10
```

Testing:

Use similar testing procedures as Question 1. The testing script for this question can be executed as follows:

```
./question2_NAME | python A2sequence2_test.py
```

Submission:

Submit your assignment in a subdirectory called question2. The name of your c program should be: question2_NAME.c, and should compile as follows: gcc question2_NAME.c -o question2_NAME. This may vary, for instance, if you need -lpthread on your system.

Question 3: Ordering the Execution

Often it's useful to order thread execution. In this problem, we generate 10 threads, each is assigned a value 0 through 9, respectively (i.e., thread 0 is assigned value 0, thread 1 is assigned value 1, ..., and thread 9 is assigned value 9). Each thread calls a function `do_work()`. There is a global variable called "total". Each thread takes a turn adding its value to total, i.e., adding 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, and then adding 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 again, and so on. It must add the values in order, i.e., 5 cannot be added before 2 in a single "sequence". If you do this 22 times, the total value will be 990 $((0+1+2+3+4+5+6+7+8+9)*22=990)$. Write a program where each thread adds its value to total, over and over, which stops adding when total is equal to 990. After the threads join, the value of "total" should be output to the screen. You may not put your threads to sleep during the execution of the program, although feel free to use `usleep()` when testing and debugging. Every time a thread updates total it should output it's assigned number and the updated value of total.

Below is an example output at the beginning and end:

```
my num: 0, total: 0
my num: 1, total: 1
my num: 2, total: 3
my num: 3, total: 6
my num: 4, total: 10
...
my num: 6, total: 966
my num: 7, total: 973
my num: 8, total: 981
my num: 9, total: 990
Total: 990
```

Testing:

Use similar testing procedures as Question 1. The testing script for this question is executed as follows:

```
./question3_NAME | python A2orderexec_test.py
```

Submission:

Submit your assignment in a subdirectory called question3. The name of your c program should be: question3_NAME.c, and should compile as follows: gcc question3_NAME.c -o question3_NAME. This may vary, for instance, if you need -lpthread on your system.

Question 4: Ping Pong using Locks and Condition Variables

Write a program using 2 threads, which call `decrement_work()` and `increment_work()`, respectively. Maintain a shared counter that is initialized to 0. The function `increment_work()` increments the counter by 1 at each loop iteration, and `decrement_work()` decrements the counter by 1 at each loop iteration.

Increment the counter value from 0 to 10 in `increment_work()`. Once the counter hits 10, the thread should wait. Then the decrement thread should decrement the counter until the value is 0. Then it signals the increment thread that the value is 0, and the increment thread begins incrementing the counter. The decrement thread waits until it is signaled that the value is 10 and so on. The total number of increments and decrements combined is 50, such that the final value is 10, (i.e., 30 total increments and 20 total decrements). Implement this using locks and condition variables. You may not put your threads to sleep during the execution of the program. Each time a thread modifies the counter, output it to the screen.

Below is example output at the end of the program:

```
...
Count is now (inc fn): 8
Count is now (inc fn): 9
Count is now (inc fn): 10
Count is now (dec fn): 9
Count is now (dec fn): 8
Count is now (dec fn): 7
Count is now (dec fn): 6
Count is now (dec fn): 5
Count is now (dec fn): 4
Count is now (dec fn): 3
Count is now (dec fn): 2
Count is now (dec fn): 1
Count is now (dec fn): 0
Count is now (inc fn): 1
Count is now (inc fn): 2
Count is now (inc fn): 3
Count is now (inc fn): 4
Count is now (inc fn): 5
Count is now (inc fn): 6
Count is now (inc fn): 7
Count is now (inc fn): 8
Count is now (inc fn): 9
Count is now (inc fn): 10
```

Testing:

Use similar testing procedures as Question 1. The testing script for this question is executed as follows:

```
./question4_NAME | python A2pingpong_test.py
```

Submission:

Submit your assignment in a subdirectory called question4. The name of your c program should be: question4_NAME.c, and should compile as follows: `gcc question4_NAME.c -o question4_NAME`. This may vary, for instance, if you need `-lpthread` on your system.