SI411 pthreads lab Name \_\_\_\_\_\_\_\_George Prielipp\_\_\_\_\_\_\_\_\_\_\_\_\_

Fall 2025 (AY26)

84 points total

**Learning Objectives**: Understand how POSIX threads are created and terminated (rejoined), and their functionality in relation to the OS.

**Instructions**:

* Must be completed on a Linux lab machine. **Do not use WSL or a VM.**
* Enter each answer in the box or space provided
  + For this lab, you may download this file and type your answers or may NEATLY hand-write your answers. If it’s not easily legible, it’s not correct!
* Use your own words. Do \*not\* copy and paste text from elsewhere.
* Deliverables
  + **Hard-copy of this worksheet,** due at start of class on the due date.   
    **(**staple required for full credit**)**
  + For part 2, turn in your prime2.c solution to submit.cs.usna.edu under “lab02\_threads”

**Background**:

A “race condition” is when the outcome of a series of operations depends not only on the data, but also on the order of execution – for instance on the relative order in which threads execute  
 [we’ll talk more about this later, but you’ll need this definition below].

**Part 1:** race.c

(4) Write down a command line that successfully builds race.c into an executable. For all parts of this lab, be sure to include the -Wall flag (turning on these warnings may save you some pain!)

|  |
| --- |
| Clang -o race race.c -Wall |

(2) Compile and run the program. Run it many times and observe the results. Do the threads always finish the 'race' in the same order?

|  |
| --- |
| No |

(4) Why or why not?

|  |
| --- |
| It all depends on when the library schedules the threads to run and when they are able to finish. |

(4) Does this constitute a 'race condition'? Explain why or why not.

|  |
| --- |
| Yes, a race condition is based on the order of execution of the threads making the output or data accesses inconsistent. |

(2) Does any particular thread always finish the race last? If so, which one?

|  |
| --- |
| Thread 3. |

(4) While the “sleepy” thread is waiting to finish, what is the state of the *main* thread (ready, running, or blocked) at pthread\_join()?

|  |
| --- |
| The main thread is blocked while it waits for the other thread to join. |

(4) Why?

|  |
| --- |
| This is a blocking call that waits for the specified thread to terminate. |

(4) When the sleepy thread sleeps, do the other threads block, or do they finish on their own?

|  |
| --- |
| The other threads finish on their own. |

(4) Based on your answer to the previous question, can the pthreads implementation possibly be a 'pure ULT' strategy?

|  |
| --- |
| Yes because the library is handling the scheduling and keeping track of the thread control blocks in a thread table. So the library can block a thread that is waiting on another thread to terminate. |

(4) If you run the program often enough, the output formatting will occasionally be skewed -- one printf() statement seems to be interrupting another printf() statement. Explain how this could happen?

|  |
| --- |
| Printf is writing to stdout. During the race condition, some threads begin putting data on stdout, then get essentially “talked over,” allowing the other thread to start putting data on stdout. What ends up a long mess of text from unprotected data writes. |

**Part 2: Complete** prime2.c

You are given a basic program, prime1.c, that uses a single thread to iterate over all the integers in a file, and print only those numbers that are prime. For simplicity, the primality test is a basic, non-optimized version that's given for you.

- Save a copy of your file prime1.c to a new version, prime2.c

- Compile and run prime2.c (let a.out be the executable). Look at the results and make sure it makes sense. How many prime numbers are found? \_\_89\_\_\_\_\_

- Read through prime2.c to understand how it works.

- (20) Follow the TODO directions so that prime2.c executes with three threads instead of one. The threads should divide the labor equally. For example, thread 0 performs a primality test on numbers with array index 0, 3, 6, 9, 12... while thread 1 performs a primality test on numbers with array index 1, 4, 7, 10,… and thread 2 takes numbers with index 2, 5, 8, 11, …. This way, each thread does one third the work. HOWEVER, don’t “hardcode” the number of threads as 3. Your code should work, in a similar way, for any value of NTHREADS (including a value of 1).

Suggestion: temporarily change NTHREADS to 1, and get the code working where you simply create one thread and have it call check\_primes(). Once this works, save a backup copy, then work on having NTHREADS be 3, and modifying check\_primes to divide up the work properly.

- When your code is working, run it with three threads. Verify that you still find the same number of primes.

- Verify your code also works with NTHREADS set to 8.

- Now change NTHREADS back to 1. Verify that you \*still\* find the same number of primes.

- (2) Run this: ./a.out | head -3

What are the first three primes that you see? If you run many times, are they always the same?

|  |
| --- |
| 553543  406313  376133  Always the same |

- (4) Change NTHREADS to 3. Run this: ./a.out | head -3

Do this several times. Does your output always stay the same? YES or NO (circle one)

Explain why what you see makes sense.

|  |
| --- |
| NO. It does not make a ton of sense. It is pretty consistently:  32719  68881  408091  Every 1 out of 10 it is:  32719  68881  553543  Old opinion, I think I have a new one - I think what I should be seeing is a consistent output (or at least consistently the same three numbers in the top 3) because I’m making the threads wait while one thread is writing the data and increasing the count. The output above should also be the first prime found from the NTHREADS starting points that I have defined. It also takes time for each thread to find the it’s first prime.  New opinion – The output should be more random and inconsistent because the scheduler is deciding when to run the threads and there is no guarantee that the threads will run in the order they have been consistently running in for some reason. |
|  |

You know that you can find primes with one or more threads. Let’s print a helpful summary of those primes. Change NTHREADS to 1, and uncomment the call to printResults(). Compile and run again. Look at the summary of primes that it prints out (at the end).

(2) Now, change NTHREADS to 25 and run again. You should see that the summary of primes includes many correct answers, but does NOT look fully correct, despite all of the individual “found” messages for the primes looking fine above. We do NOT care about the order of primes – you are looking for a different sort of problem.  
Describe what **specifically looks** **wrong** about the output:

|  |
| --- |
| I cannot see anything wrong with the output. I think I was supposed to make a common mistake and didn’t do that. I’m not entirely certain what I need to be looking for. |

(4) Explain what (in the program/execution) is going wrong.

|  |
| --- |
|  |

(4) You should be able to mostly fix this problem by making a small change to the check\_primes() function. It won’t be perfect – we need to learn about IPC to really fix the problem – but the frequency of incorrect output should be much less. Go ahead and make the changes, and verify that the problem is fixed. Your new solution must continue to properly have multiple threads simultaneously checking for primes.

Describe what you fixed (and include in final prime2.c):

|  |
| --- |
|  |

(4) Now modify prime2.c so that (a) each thread returns back to main() a count of how many primes were found by that particular thread, and (b) in main(), after all threads have completed, you print out the counts for each thread. So your final output might look like this, for 10 threads:

thread 0 found 7 primes  
thread 1 found 7 primes  
thread 2 found 7 primes  
thread 3 found 10 primes  
thread 4 found 8 primes  
thread 5 found 9 primes  
thread 6 found 4 primes  
thread 7 found 11 primes  
thread 8 found 12 primes  
thread 9 found 14 primes  
Found a total of 89 primes.

You may NOT use any global variables as part of solving this task.

Nothing to write on this worksheet – just include this updated code in your final prime2.c

**Part 3: Timing Analysis (prime3.c)**

Your specific machine will matter for this part. Run hostname on your machine and enter the   
result here: \_\_\_\_\_\_lnx1084326govt\_\_\_\_\_\_\_\_\_\_\_

Make a new file:

cp prime2.c prime3.c

In prime3.c, comment out all the printf statements \*except\* this one:  
 printf ("Found a total of %d primes, here they are…

Run your program 3-4 times with NTHREADS set to 1, and 3-4 times NTHREADS set to 5 (don’t forget to recompile prime3.c each time!). Record the average “real” time when you execute using the “time” command line tool like this:   
 time ./a.out

(“real” time is essentially the actual clock time. Replace a.out with name of your executable, if necessary)

(2) Average “real” time with 1 thread: \_\_\_\_0.074s\_\_\_\_\_\_\_\_

(2) Average “real” time with 5 threads: \_\_\_\_0.0294s\_\_\_\_\_\_\_

(4) Do your results suggest that your machine has a single core, or multiple cores? Explain

|  |
| --- |
| Multiple cores. The more cores the CPU has, the more threads it can run in parallel (or rather, can be scheduled to execute at the same time). This is where the real speed up actually happens with threading. |

(4) Why don't the programs take the exact amount of time every run?

|  |
| --- |
| There are other factors at play when programs execute. Such as background tasks, user I/O, context switching to the kernel for other OS tasks. |

**Deliverables**

- See details on page 1.