CPSC 457: Principles of Operating Systems

Assignment 3: threads, pthreads, C++ threads

Weight: 21%

Collaboration

Discussing the assignment requirements with others is a reasonable thing to do and an excellent way to learn. However, the work you hand in must ultimately be your work. This is essential for you to benefit from the learning experience and for the instructors and TAs to grade you fairly. Handing in work that is not your original work but is represented as such is plagiarism and academic misconduct. Penalties for academic misconduct are outlined in the university calendar.

Here are some tips to avoid plagiarism in your programming assignments.

1. Cite all sources of code you hand in that are not your original work. You can put the citations into comments in your program. For example, if you find and use code found on a website, include a comment that says, for example:

   # The following code is from https://www.quackit.com/python/tutorial/python_hello_world.cfm.

   Use the complete URL so that the marker can check the source.

2. A tool like chat-GPT can be used to improve small code blocks. For example, three lines of code. If you get help from code assistance like Chat-GPT, you should comment above the block of code you requested assistance on debugging or improving and cite the tool used to get that suggestion. Using a tool like chat-GPT to write the majority of your assignment requirements will be treated as plagiarism if found without citation, and with citation, it will be treated as 0 for the component the student did not complete. Code improvement of short length will get credit if commented/cited properly.
3. Citing sources avoids accusations of plagiarism and penalties for academic misconduct. **However, you may still get a low grade if you submit code not primarily developed by yourself. Cited material should never be used to complete core assignment specifications unless clearly approved. Before submitting, you can and should verify any code you are concerned about with your instructor/TA.**
4. Discuss and share ideas with other programmers as much as you like, but make sure that when you write your code, it is your own. A good rule of thumb is to wait 20 minutes after talking with somebody before writing your code. If you exchange code with another student, write code while discussing it with a fellow student, or copy code from another person's screen, this code is not yours.
5. **Collaborative coding is strictly prohibited**. **Your assignment submission must be strictly your code**. Discussing anything beyond assignment requirements and ideas is a strictly forbidden form of collaboration. This includes sharing code, discussing the code itself, or modelling code after another student's algorithm. **You can not use (even with citation) another student's code.**
6. Making your code available, even passively, for others to copy or potentially copy is also plagiarism.
7. We will look for plagiarism in all code submissions, possibly using automated software designed for the task. For example, see Measures of Software Similarity (MOSS - https://theory.stanford.edu/~aiken/moss/).
8. Remember, if you are having trouble with an assignment, it is always better to go to your TA and/or instructor for help rather than plagiarizing. A common penalty is an F on a plagiarized assignment.

Late Policy

All students will have 6 total days during the semester that can be used for none, one, some, or all the 6 assignments. Students can use these days all on one assignment, or some on each of the six.

Each 24-hour period late after an assignment deadline counts as one full day regardless of how many hours the assignment was late within that period. For example, deadlines are generally Fridays 11:59pm local time. That means an assignment submitted any time Saturday before 11:59pm local time will be considered as 1 day late and count against the students 5 total days.

As long as a student still has days left to use, their assignment will be graded without penalty. If a student has no more days left or their submission exceeds the days they have remaining, then they will receive a 0 grade for a late assignment. Students who use up late days on earlier assignments will not have them available for later assignments.

TAs will indicate the student's late day usage in grading feedback.

Due date is posted on D2L. Your D2L submission should include the files requested and a link to a Gitlab repository you used while completing the assignment with your TA added as a **Developer** role. Help with Gitlab Clone/Developer role access is available in D2L video.

Q1. Programming question – calculating π [10 marks]

Improve the performance of an existing single-threaded **calcpi** program by converting it to a multi-threaded implementation.

Start by cloning the repository with starter code (type the following on the command line on the linux lab computers). You may need to update the repository if you pulled it before to see future assignment changes before starting an assignment:

```
$ git clone https://csgit.ucalgary.ca/jwhudson/cpsc457w24.git
$ cd cpsc457w24/pi-calc
$ make
$ ./calcpi
Usage: ./calcpi radius n_threads
   where 0 <= radius <= 100000
     and 1 <= n_threads <= 256
```

The calcpi program estimates the value of π using an algorithm described in:

https://en.wikipedia.org/wiki/Approximations_of_%CF%80#Summing_a_circle's_area.

Most of the algorithm is implemented inside the function **count_pixels()** in file **calcpi.cpp**. The included driver (**main.cpp**) parses the command line arguments, calls **count_pixels()** and prints the results. The driver takes **2** command line arguments: an integer radius and number of threads. For example, to estimate the value of π using radius of **10** and **2** threads, you would run it like this:

```
$ ./calcpi 10 2
Calling count_pixels(r=10, n_threads=2)...
Result = 317 pixels (estimated PI=3.17)
```

The function **uint64_t count_pixels(int r, int N)** takes two parameters – the radius and number of threads. It then returns the number of pixels inside the circle with radius r and centered at **(0,0)**, by checking every pixel **(x,y)** in squre $-r \leq x, y \leq r$.

The current implementation is single threaded, so it ignores the 2nd argument **N**. Your job is to re-implement the function so that it uses N threads to speed up its execution, such that it runs N times faster with **N** threads on hardware where **N** threads can run concurrently.

Please note that your code will be marked both for correctness and for the speedup it achieves. In order for your code to be considered correct, your multi-threaded implementation needs to return the same number of pixels as the single-threaded implementation.

For this question, you are **only allowed to create and join threads**. You need to find a way to parallelize **count_pixels()** **without** using any synchronization mechanisms, such as mutexes, semaphores, atomic types, etc.

Write all code into **calcpi.cpp** and submit this file for grading. Make sure your **calcpi.cpp** works with the included driver program. We may use a different driver program during marking, so it is important that your code follows the correct API. Make sure your program runs on **cslinux.ucalgary.ca**.

Assume $\mathbf{0 \leq r \leq 100{,}000}$ and $\mathbf{1 \leq nthreads \leq 256}$.

Timing your code on lab machines

Please note that not all machines may be the same. Before you report and threaded timings you must report the information returned by **lscpu** command.

For example from ssh to cslinux: **"Model name: Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz"**

This model has 12 cores and 1 thread per core (so no Hyper-Threading).

Physical lab machines will be different. For example, some have 6 physical cores with hyper-threading enabled, while others have 8 physical cores, but no hyperthreading. When you are running your timings, please make sure you do all of them on the same machine. Otherwise, you will get inconsistent results.

My basic multi-threaded implementation using cthreads and -O2 achieves the following timings using r=100000. I expect your solutions to achieve similar results on the same CPU.

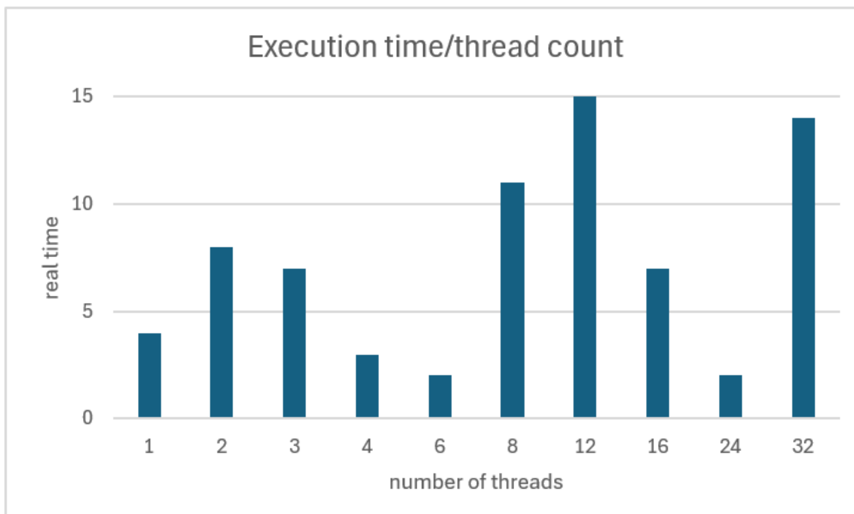
CPU	1 thread	2 threads	4 threads	8 threads	16 threads
Intel(R) Xeon(R) Gold 6240	14.316s	8.543s	4.819s	2.768s	2.146s

Q2. Written answer [3 marks]

Time your multi-threaded solution from **Q1** with ***r*=50000** using the time command on **cslinux.ucalgary.ca**. Record the real-time for **1, 2, 3, 4, 6, 8, 12, 16, 24** and **32** threads. Also record the timings of the original single-threaded program.

A. Record your timings in a table and create a corresponding bar graph. Format the table and the graph like the ones below (the numbers below are random and your timings should look different).

Threads	Timing(s)
1 (original)	5
1	4
2	8
3	7
4	3
6	2
8	11
12	15
16	7
24	2
32	14



B. When you run your implementation with **N** threads, you should see **N**-times speed up compared to the original single threaded program. Do you observe this in your timings for all values of **N**?

C. Why do you stop seeing the speed up after some value of **N**?

Q3. Programming question – detecting primes [30 marks]

For this part of the assignment, you must convert a single-threaded program **detectPrimes** to a multi-threaded implementation.

Start by cloning the repository with starter code (type the following on the command line on the linux lab computers). You may need to update the repository if you pulled it before to see future assignment changes before starting an assignment:

```
$ git clone https://csgit.ucalgary.ca/jwhudson/cpsc457w24.git
```

```
$ cd cpsc457w24/detect-primes
$ make
$ cat example.txt
      0   3   19 25 3
4012009 165 1033
$ ./detectPrimes 5 < example.txt
Using 5 threads.
Identified 4 primes:
  3 19 3 1033
Finished in 0.0000s
$ seq 100000000000000000 100000000000000300 | ./detectPrimes 2
Using 2 threads.
Identified 9 primes:
  100000000000000003 100000000000000013 100000000000000019 100000000000000021
  100000000000000049 100000000000000081 100000000000000099 100000000000000141
  100000000000000181
Finished in 7.3076s
```

The **detectPrimes** program reads integers in range $[2, 2^{63} - 2]$ from standard input, and then outputs the ones that are prime numbers. The first invocation example above detects prime numbers **3**, **19, 3** and **1033** in a file **example.txt** (notice that number **3** is repeated both in the input and output). The second invocation uses the program to find all primes in the range $[10^{17}, 10^{17} + 300]$. If duplicate primes appear in the input, they will be duplicated in the output.

detectPrimes accepts a single command line argument – a number of threads. This parameter is not used in the current implementation, as the starter code is single threaded. Your job is to improve the execution time of **detectPrimes** by making it multi-threaded, and your implementation should use the number of threads given on the command line. To do this, you will need to re-implement the code in **detectPrimes.cpp**, namely the function:

```
std::vector<int64_t>
detect_primes(const std::vector<int64_t> & nums, int n_threads);
```

The function takes two parameters: the list of numbers to test, and the number of threads to use. It returns the prime numbers found, in arbitrary order. The function is called by the driver (**main.cpp**) after parsing the standard input and command line. Your implementation should use **n_threads** number of threads.

Ideally, if the original single-threaded program takes time T to complete a test, then your multi-threaded implementation should finish that same test in T/N time when using N threads. For example, if it takes **10s** to complete a test for the original single-threaded program, then it should take your multi-threaded program only **2.5s** to complete that same test with **4** threads. To achieve this goal, you will need to design your program so that:

- You give each thread the same amount of work;
- your multi-threaded implementation does the same amount of work as the single-threaded version; and
- the synchronization mechanisms you utilize are efficient.

We will mark your assignment by running the code against multiple different inputs and using different numbers of threads. To get full marks for this assignment, your program needs to output correct results but also achieve near optimal speedup for the given number of threads and available cores. If your code does not achieve optimal speedup on all inputs, you will lose some marks for those tests.

You may assume that there will be no more than **100,000** numbers in the input, and that all numbers will be in the range $[\mathbf{2}, \mathbf{2^{63}} - \mathbf{2}]$. Some inputs will include many numbers, some inputs will include just few numbers, some numbers will be large, some small, some will be prime numbers, others will be large composite numbers, etc… For some numbers it will take long time to compute their smallest factor, for others it will take very little time. You need to take all these possibilities into consideration. Design your own test inputs and test your code thoroughly.

Write all code into **detectPrimes.cpp** and submit it for grading. Make sure your **detectPrimes.cpp** works with the included **main.cpp** driver. We may use a different driver during marking, so it is important that your code follows the correct API. Make sure your program runs on **cslinux.ucalgary.ca**.

You may use any of the synchronization mechanisms we covered in lectures, such as semaphores, mutexes, condition variables, spinlocks, atomic variables, and barriers. Make sure your code compiles and runs on **cslinux.ucalgary.ca**.

Please note that the purpose of this question is **NOT** to find a more efficient factorization algorithm. You **must** implement the **exact same** factorization algorithm as given in the skeleton code, except you need to make it multi-threaded.

Q4. Written question (5 marks)

Time the original single-threaded **detectPrimes.cpp** as well as your multi-threaded version on three files: **medium.txt, hard.txt** and **hard2.txt**. For each of these files, you will run your solution **6** times, using **1, 2, 3, 4, 8** and **16** threads. You will record your results in **3 tables**, one for each file, formatted like this:

medium.txt			
# threads	**Observed timing**	**Observed speedup compared to original timing**	**Expected speedup**
original program		1.0	1.0
1			1.0
2			2.0
3			3.0
4			4.0
8			8.0
16			16.0

The '**Observed timing**' column will contain the raw timing results of your runs. The '**Observed speedup**' column will be calculated as a ratio of your raw timing with respect to the timing of the original single-threaded program.

Once you have created the tables, explain the results you obtained. Are the timings what you expected them to be? If not, explain why they differ.

Submission

Submit one file for this assignment to D2L:

- Your solution to Q1 in a file called **calcpi.cpp.**
- Your solution to Q3 in a file called **detectPrimes.cpp.**
- Your answers to written questions Q2/Q4 in a file called **report.pdf.**

Please note – you need to **submit all files every time** you make a submission, as the previous submission will be overwritten.

Submit this as a separate file. **Do not** submit an archive, such as ZIP or TAR. If you submit an archive, you will receive a penalty.

Submit the web address of your Gitlab repository you used for your assignment (I recommend one repository for all 6 of your assignments that you can re-use). Your TA must be added as a Developer. There are penalties for submissions without the ability to access the corresponding students Gitlab by the TA.

While the starter code contains many different files, the only file you are allowed to modify is **calcpi.cpp/detectPrimes.cpp.** Do not modify any other files. All code you write must go in the **calcpi.cpp/detectPrimes.cpp** files, and that should be the only file you will submit for grading. We will test your code by supplying our own **main()** function, which will be different from the **main()** function in the starter code. It is therefore vital that you maintain the same function signature as declared in **calcpi.h/detectPrimes.h.** Before you submit

calcpi.cpp/detectPrimes.cpp to D2L, make sure it works with unmodified files from the starter code!

General information about all assignments:

All assignments are due on the date listed on D2L. Late submissions without remaining late days banked will not be marked.

1. Extensions beyond the late day policy can be discussed more than 5 business days in advance and are granted only by the course instructor.
2. After you submit your work to D2L, verify your submission by re-downloading it.
3. You can submit many times before the due date. D2L will simply overwrite previous submissions with newer ones. It is better to submit incomplete work for a chance of getting partial marks, than not to submit anything. Please bear in mind that you cannot re-submit a single file if you have already submitted other files. Your new submission would delete the previous files you submitted. So please keep a copy of all files you intend to submit and resubmit all of them every time.
4. Assignments are likely going to be marked by your TAs. If you have questions about assignment marking, contact your TA first. If you still have questions after you have talked to your TA, then you can contact your instructor.
5. All programs you submit must run on **linux lab** or **cslinux.ucalgary.ca.** If your TA is unable to run your code on these, you will receive 0 marks.
6. Unless specified otherwise, you must submit code that can finish on any valid input under 10s on **linux lab** or **cslinux.ucalgary.ca**, when compiled with **-O2** optimization. Any code that runs longer than this may receive a deduction, and code that runs for too long (about 30s) will receive 0 marks.
7. **Assignments must reflect individual work.** Here are some examples of what you are not allowed to do for individual assignments: you are not allowed to copy code or written answers (in part, or in whole) from anyone else; you are not allowed to collaborate with anyone; you are not allowed to share your solutions (code or pseudocode) with anyone; you are not allowed to sell or purchase a solution; you are not allowed to make your code available publicly (e.g. via public git repositories). This list is not exclusive. For further information on plagiarism, cheating and other academic misconduct, check the information at this link: http://www.ucalgary.ca/pubs/calendar/current/k-5.html .
8. We will use automated similarity detection software to check for plagiarism. Your submission will be compared to other students (current and previous), as well as to any known online sources. Any cases of detected plagiarism or any other academic misconduct will be investigated and reported.

Appendix – Hints for Q1

I suggest you parallelize the outer loop. Give each thread roughly equal number of columns in which to count the pixels. Then sum up the counts from each thread. Your overall algorithm could look like this:

- Create separate memory area for each thread (for input and output), e.g. as seen in class notes **struct Task**
- Divide the work evenly between threads, e.g. a portion of the range of one of the loops
- Create threads and run each thread on the work assigned to it. Each thread counts pixels for the x-range assigned to it and updates its **partial_count**.
- Join the threads.
- Combine the results of each thread into final result – i.e. return the sum of all **partial_counts**.

Appendix – Hints for Q3

Hint 1 – bad solution (do not implement this)

A bad solution would be to parallelize the outer loop of the algorithm and assign a fixed portion of the numbers to each thread to check. This is a terrible solution because it would not achieve speedups on many inputs, for example where all hard numbers are at the beginning, and all the easy ones at the end. Your program would then likely give all hard numbers to one thread and would end up running just as slowly as a single-threaded version.

Hint 2 – simple solution (start with this)

A much better, yet still simple solution, would be to parallelize the outer loop, but instead of giving each thread a fixed portion of the input to test, each thread would dynamically determine how many numbers it would process. For example, each thread could be setup to process the next number in the list, and if it is a prime, it would add it to the result vector. This would repeat until all numbers have been tested. All you need to implement this solution is a single mutex to guard access to the input vector, and to the result vector. I strongly suggest you start by implementing this simple solution first, and only attempt the more difficult approaches after your simple solution works.

Note that this solution would achieve optimal speedup for many inputs, but not for all. For example, on input with a single large prime number, it will not achieve any speedup at all. Consequently, if you choose this approach, you will not be able to receive full marks for some tests.

Hint 3 – good solution

Even more efficient approach is to parallelize the inner loop (the loop inside the is_prime function). In this approach, all threads would work on testing the same number for primality. If you choose this approach, you need to give each thread a different portion of divisors to check. This will allow you to handle more input cases than the simple solution mentioned earlier. For extra efficiency, and better marks, you should consider implementing thread re-use, e.g., by using barriers. Here is a possible rough outline of an algorithm that you could implement:

```
detectPrimes():
      prepare memory for each thread
      initialize empty array result[] – this could be a global variable
      set global_finished = false – make it atomic to be safe
      start N threads, each runs thread_function() on its own memory
      join N threads
      return results
thread_function():
      repeat forever:
            serial task – pick one thread using barrier
                  get the next number from nums[]
                  if no more numbers left:
                        set global_finished=true to indicate end
                  otherwise:
                        divide work for each thread
            parallel task – executed by all threads, via barrier
                  if global_finished flag is set
                        exit thread
                  otherwise
                        do work assigned above
                        record per-thread result
            serial task - pick one thread using barrier
                  combine the per-thread results
                  update the result[] array if necessary
```

The only synchronization primitive you should need to implement this hint is a barrier. You should not need to use any other synchronization primitives (remember that any code inside your serial code does not require to be protected by mutexes…). You may use my C++ barrier implementation from lecture notes if you wish.

Hint 4 – best solution

This builds on top of the hint 3 above, by adding to it thread cancellation. You need cancellation for cases where one of the threads discovers the number being tested is not a prime, so that it can cancel the work of the other threads. I suggest using a single atomic boolean variable to implement the cancellation flag. Please note that while thread cancellation is very simple to implement, it does take non-trivial effort to get it working correctly.

Appendix – Approximate grading scheme for Q3

The test cases that we will use for marking will be designed so that you will get full marks only if you implement the most optimal solution. However, you will receive partial marks even if you implement one of the less optimal solutions. Hint 2 as the starter would be about half the marks for Q3 and hint 1 which is not suggested would be less than half even if the correct primes are all recognized.

Please note that any tests on which your program produces wrong results will receive 0 marks.