CSE381 - Spring 2024

Homework 02: Synchronization-Free Multithreading

Reminder: Per course policy, support for homework assignments ends 24 hours before the due date. This assignment is due before midnight on Friday, April 5, so instructor/TA support ends at midnight on Thursday, April 4.

Overview

In class, we have spent several sessions reviewing the concept of threads and how to write multithreaded programs. The most simple kind of multithreading is ***synchronization-free multithreading*** in which a problem can be divided across multiple threads in such a way that no thread is ever updating a memory location that another thread needs to read from or write to.

Recall that when a thread is updating a shared memory location, it requires *exclusive* access to it to ensure that its update completes before other threads access the variable that is being updated. This requires creating a *critical section* via some kind of lock and this can negatively impact the program's performance. In this homework assignment, you will be writing a fairly straightforward program that can divide a problem across multiple threads without the need for locking. If implemented efficiently, this can result in a significant increase in performance compared to a single-threaded (serial) program.

Breaking a problem up in a way that multiple threads can attack it efficiently without interfering with each other can be a challenge. Consider the problem of using two threads to count the prime numbers less than a number provided. (Which is actually the problem you'll be solving in this assignment…)

If you were to divide this in half, with one thread taking the lower half of the numbers and the other taking the higher half of the numbers, you could assure that each thread is working on an independent set of numbers so they cannot interfere with each other. However, the effort it takes to tell if a number is prime goes up as the number gets larger, so in this implementation, the thread doing the lower half would finish well before the thread doing the upper half, and much of the run would be single-threaded.

Another common approach to dividing a problem like this is to allow one thread to do even numbers while the other thread does odd numbers. Hopefully you can see why this approach cannot be used efficiently when checking if numbers are prime…

The approach I will ask you to take is to break the problem not into even/odd, but consecutive pairs, so 0 and 1, 2 and 3, 4 and 5, and so on. This ensures that both threads will work on numbers spanning the entire range, and each of them will also have the same number of even and odd numbers to check. To make this explicit, one thread should work on these numbers

0, 1, 4, 5, 8, 9, 12, 13, …

while the other thread works on

2, 3, 6, 7, 10, 11, 14, 15, …

If implemented correctly, this can lead to a high level of parallelism. However, even in a synchronization-free multithreaded program, it is still possible to implement multithreading in an inefficient way, so it is not a panacea. To see what impact this can have on performance, I will ask you to implement multithreading in an inefficient way and an efficient one. You should note a significant difference in run times for the two versions. And, in the spirit of incremental development, I will first ask you to implement a serial version that has no multithreading at all.

In the inefficient version, two threads will be created, each thread will check on its pair of numbers, be joined, and then two new threads will be created for the next two pairs of numbers. In the efficient version, two threads will be created, and the same threads will be reused for all of the subsequent pairs they are to check. You should time the execution of your program with **`/usr/bin/time`** to measure the performance difference, and also run the program for a sufficiently large number (500000 or greater) so you can observe the performance via **`top`**.

In addition to learning more about writing and timing multithreaded programs, this homework will also let you gain more experience using the CODE plugin in Canvas to test functionality of a program, and practice the incremental development approach we've discussed in class and which you've used in other homework and lab assignments.. There are multiple tests defined by CODE for this assignment. Passing four of those tests is **required** in order to submit your program while the others are not required, but can give you additional points each time you pass one of them.

Your program will have to pass the **`cpplint`** style checking inside of CODE with no warnings or errors, and will also have to compile with no warnings or errors in order to be submittable. As mentioned in class, **`cpplint`** in the CODE plugin is now checking for the number of lines of code in a function, so make sure you are paying attention to the **`cpplint`** messages in VS Code because you'll see the exact same ones in Canvas.

Make sure you are fixing style errors as they arise, and are submitting your program and making backups of it once you pass additional tests in CODE. **If you follow the incremental development process we discussed in class, and practiced in other assignments, you will be able to get at least partial credit for this assignment even if you cannot pass all of the optional tests in the time allotted for the assignment.**

Description of required/requested functionality

The purpose of this program is straightforward: Given a number as a command line argument, the program will determine how many prime numbers are less than that argument. You can read more about this at the Wikipedia article for the prime-counting function.

I am giving you a starter file called **`primesMT.cpp`**. It contains the implementation of a **`bool`** function called **`isPrime`** which takes an **`int`** argument. **There may be more efficient ways to tell if a number is prime, however, this is the function that your client (me) is requiring you to use and which you must use.** (If it helps your sanity, assume that you are working for a company in a

highly-regulated industry which requires the use of this particular function. This kind of situation is actually not far off the mark for some industries…)

You are to implement this program in three ways:

- A serial implementation that uses the **`isPrime`** function to check all numbers greater than equal to zero and less than a number provided as a command line argument and keeps track of how many of them are prime numbers.
- An inefficient multithreaded program that runs two concurrent threads to work on pairs of numbers, then closes (joins) the threads and creates new threads for the next pairs of numbers.
- An efficient multithreaded program that creates two threads and reuses them to traverse the pairs of numbers that the thread will work on.

The program should take two **`int`** arguments: The first one specifies the number that you are checking, i.e., you will determine how many prime numbers there are that are less than this number. The second command line argument is the mode the program should run in.

- If the mode argument is `0`, the program should execute the serial version.
- If the mode argument is `1`, the program should execute the inefficient two-threaded version
- If the mode argument is some other **`int`**, the program should execute the efficient two-threaded version.

The program should be fortified in the following manner:

- If there are not exactly two command line arguments, it should exit (remember to exit with status `0`) and display this message:

  ```
  Usage:  primesMT maxToCheck mode
  ```

- If either of the command line arguments is **not** an **`int`**, the program should exit and display this message:

  ```
  Error: arguments must be integers
  ```

Some other tips and hints:

- Here's a table of values and results the program should give you if it's working correctly. In addition, it gives you the approximate elapsed time the serial and efficient programs should take when run with those arguments and compiled with `-O3` optimization. Note that your times will vary based on your CPU and your particular implementation, but these give you the ballpark of what to expect.

maxToCheck	Result	Serial Elapsed (s)	Efficient 2-thread Elapsed (s)
400000	33860	8.6	4.4
600000	49098	18.6	9.4
800000	63951	32.5	16.5

1000000	78498	49.9	25.4

- Make sure you're following the iterative approach to developing the program. In this case, your MVP is going to be a program that is fortified with regard to the command line arguments (described above) and which runs correctly in serial mode. **So, you shouldn't even think about any multithreading AT ALL until you pass the MVP and submit this to Canvas so there's something working**.
- You are welcome to use either **`std::thread`** or **`std::future`** for this program.
- If you're timing the program as a sanity check, you should observe that the single-threaded times are more consistent from run-to-run than the multithreaded ones, and you should find that the inefficient multithreaded program takes **LONGER** to run than the serial version and consumes a lot of system CPU time. This is the thread creation/deletion overhead. Think of it as driving a Lamborghini car in busy city traffic. It's doing so much starting and stopping that it can never get up to full speed.
- **Note that even when there are no style errors, the *Style Errors* section in CODE will turn pink and say `Style Errors (Errors: -2)` in the header, but the text box will be empty. This will NOT keep you from submitting your program. If there is an actual style error, there will be text displayed in the box and you will have to fix the problem before you can submit your code.**

Description of required/requested functionality

- Exit if wrong number of CLAs (**required, 2 points**): If the wrong number of CLAs is given, the program should give this error message and exit (with status `0`):

  ```
  Usage:  primesMT maxToCheck mode
  ```

- Exit if first CLA is not a valid int (**required, 2 points**): If the first CLA is not an **`int`**, the program should give this error message and exit (with status `0`):

  ```
  Error: arguments must be integers
  ```

- Exit if second CLA is not a valid int (**required, 2 points**): If the second CLA is not an **`int`**, the program should give this error message and exit (with status `0`):

  ```
  Error: arguments must be integers
  ```

- Give the correct result for the serial program (**required, 4 points**): When the program is run in serial mode (second CLA is `0`), it should give the correct answer. For example, if the program is run like this

```
./primesMT 10000 0
```

the output should be

```
Found 1229 prime numbers < 10000
```

- Give the correct result for the inefficient multithreaded program (**optional, 10 points**): When the program is run in the inefficient multithreaded mode (second CLA is `1`), it should give the correct answer. For example, if the program is run like this

  ```
  ./primesMT 10000 1
  ```

  the output should be

  ```
  Found 1229 prime numbers < 10000
  ```

  In addition, to get the credit you will have to have implemented the inefficient multithreading as described above. Partial credit is possible for this.

- Give the correct result for the efficient multithreaded program (**optional, 10 points**): When the program is run in the efficient multithreaded mode (second CLA is not `0` or `1`), it should give the correct answer. For example, if the program is run like this

  ```
  ./primesMT 10000 2
  ```

  the output should be

  ```
  Found 1229 prime numbers < 10000
  ```

  In addition, to get the credit you will have to have implemented the inefficient multithreading as described above. Partial credit is possible for this.

Grading and other notes

Minimum Viable Product (MVP). Your program must satisfy this in order to be submitted.

- Your program must compile without error or warning messages in order to be accepted by CODE.
  - Make sure you are compiling with the options discussed in class so that all warning messages are enabled!
- Your program must have no style errors or warnings in order to be accepted by CODE.
- Your program must pass the first four functional tests described above in order to be accepted by CODE.

Submitting the MVP will earn you 10 points. By passing the subsequent tests, you can earn up to 30 points total on this assignment. The number of points awarded for passing each test is detailed above.

Note that if you pass one of the optional tests in CODE, but haven't implemented the requested multithreaded program you will not receive full (or any) credit for that test.