Programming Assignment VI

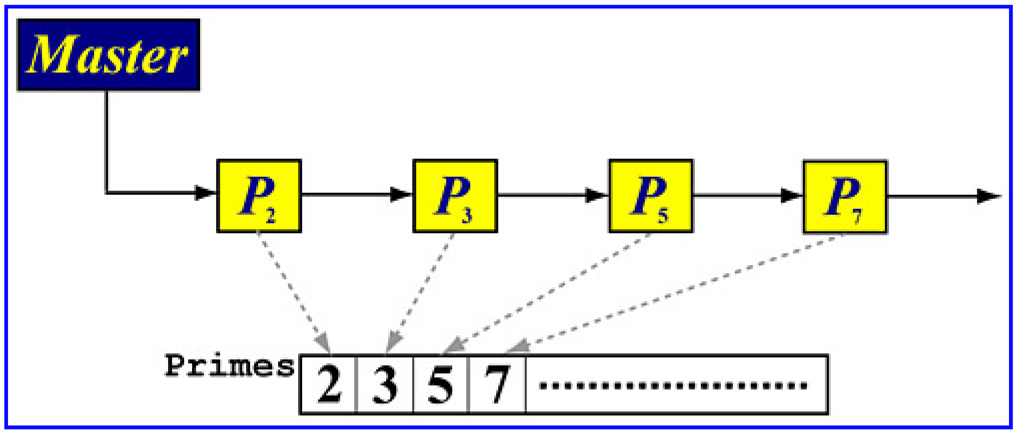
Due on Sunday, April 25th, 2021 @ 11:59pm

70 points

Sieve with Message Passing

In this problem, you are to write a sieve program to find all prime numbers between 2 and ***n***, where ***n*** is an input integer greater than 2. Unlike the sieve program you learned in other classes, this one will do things concurrently with message passing. **You must do this problem using ThreadMentor's synchronous channels. Otherwise, you will receive no credit.**

Suppose we have a sequence of "prime number" threads P_2, P_3, P_5, P_7, P_{11}, P_{13}, ... Each of these prime number threads "memorizes" a prime number (*i.e.*, its index) and is connected with two one-to-one *synchronous channels*: one from its predecessor, and the other to its successor. Note that the last thread has no successor. Finally, there is a master thread, ***Master***. It has a one-to-one synchronous channel to P_2, the first prime number thread. The figure below shows the relationship among these threads and their synchronous channels.



The job for ***Master*** is pumping a sequence of integers into the synchronous channel to P_2. Each P_i receives a number ***k*** from its predecessor and determines if ***k*** is a multiple of ***i*** as follows:

1. If ***k*** is the **END** message, P_i passes this message to its successor (if any) and then waits for its successor's completion. Finally, P_i terminates.
2. If ***k*** is a multiple of ***i***, ***k*** is not a prime number. P_i ignores ***k*** and waits for the next number from its predecessor through the incoming synchronous channel.
3. If ***k*** is not a multiple of ***i***, we have two cases to consider:
   - If P_i is the last thread in the chain (*i.e.*, P_i has no successor), because ***k*** has been checked by all previous prime threads, ***k*** is a prime. In this case, P_i creates a successor P_k, a synchronous channel is built between P_i and P_k, and P_k "memorizes" ***k*** and saves it to a global array `Primes`. After this, P_k becomes the last in the chain and both P_i and P_k are waiting for new numbers coming in from their predecessors.
   - If P_i is not the last thread in the chain (*i.e.*, P_i has a successor), then P_i just sends ***k*** to its successor and waits for new integers from its predecessor.

At the very beginning, we have the master thread ***Master*** and thread P_2 running. Then, ***Master*** starts sending 3, 4, 5, 6, 7, 8, 9, 10, ... to P_2, and other prime threads and synchronous channels will be created as needed. After the last integer has been sent by the master, the master must send the **END** message so that all prime number threads could stop working. After all prime threads finish their work, ***Master*** prints out the prime numbers saved in the global array `Primes`. **Note that the primes printed must be in ascending order and that *Master* cannot sort the array `Primes`. In other words, the prime threads must save their memorized prime numbers in ascending order to global array `Primes`.**

You can ONLY use channels for this exercise and no shared-memory mechanisms (*i.e.*, mutex locks, semaphores and monitors) should be used. The only exception may be mutex locks for protecting the global array `Primes` and `stdout`. However, this is not absolutely necessary. Also note that array `Primes` should be the ONLY global data item. More importantly, array `Primes` should be filled consecutively. In other words, `Primes[0]`, `Primes[1]`, `Primes[2]`, `Primes[3]`, `Primes[4]`, ... should contain 2, 3, 5, 7, 11, ... You will receive zero point if these rules are violated.

Input and Output

The input to your program consists of the following:

- The largest integer that the ***Master*** will send to thread P_2 should be taken from the command line argument as follows:

```
./prog6  n
```

  Thus, `./prog6  15` means that the ***Master*** will send 3, 4, 5, ..., 15 to P_2. If this command line argument is missing, the default value should be 30 (*i.e.*, $n = 30$). For example, `./prog6` means that ***Master*** will send 3, 4, 5, ..., 30 to P_2. Moreover, this command line argument is always greater than or equal to 3.
- Your program should generate an output similar to the following:

```
Master starts
  P2 starts and memorizes 2
Master sends 3 to P2
  P2 receives 3
Master sends 4 to P2
  P2 creates P3
  P2 receives 4
Master sends 5 to P2
    P3 starts and memorizes 3
  P2 ignores 4
Master send 6 to P2
  P2 receives 5
  P2 sends 5 to P3
  P2 receives 6
Master sends 7 to P2
```

```
    P3 receives 5
  P2 receives 7
    P3 creates P5
Master sends 8
  P2 receives 8
Master sends 9
      P5 starts and memorizes 5
  P2 receives 9
     ......................
Master sends END
     ......................
          P11 receives END
              P19 creates P23
      P7 receives END
     ......................
Master prints the complete result:
  2 3 5 7 11 13 17 19 23 ....
Master terminates
```

- **The `Master terminates` message should be the last one printed by your program.**
- You may assume that the largest value for ***n*** satisfies **3 <= *n* <= 100**.
- Please note the indentation of messages. A general rule goes as follows:
  - ***Master*** starts on column 1 (*i.e.*, the left most position).
  - Thread P_2 starts on column 3 (*i.e.*, an indentation of two spaces).
  - Each newly created thread has two more spaces indentation.

Submission Guidelines

General Rules

1. All programs must be written in C++.
2. Use `Canvas` to submit your work. This archive will contain both your program as well as your Makefile and README
3. Unix filename is case sensitive, `THREAD.cpp`, `Thread.CPP`, `thread.CPP`, etc are ***not*** the same.
4. We will use the following approach to compile and test your programs:

```
make noVisual     <-- make your program
./prog6  25       <-- test your program
```

   This procedure may be repeated a number of times with different input files to see if your program works correctly.
5. Your implementation should fulfill the program specification as stated. Any deviation from the specification will cause you to receive **zero** point.
6. A `README` file is always required.
7. **No late submission will be graded.**
8. **Programs submitted to wrong class and/or wrong section will not be graded.**

Compiling and Running Your Programs

This is about the way of compiling and running your program. If we cannot compile your program due to syntax errors, wrong file names, etc, **we cannot test your program, and, as a result, you receive 0 point**. If your program compiles successfully but fails to run, **we cannot test your program, and, again, you receive 0 point**.

Therefore, before submitting your work, make sure your program can compile and run properly.

1. **Not-compile programs receive 0 point.** By **not-compile**, I mean any reason that could cause an unsuccessful compilation, including missing files, incorrect filenames, syntax errors in your programs, and so on. Double check your files before you submit, since I will **not** change your program. Note again: Unix filenames are ***case sensitive***.
2. **Compile-but-not-run programs receive 0 point.** **Compile-but-not-run** usually means you have attempted to solve the problem to some degree but you failed to make it working properly.
3. **A meaningless or vague program receives 0 point even though it compiles successfully.** This usually means your program does not solve the problem but serves as a placeholder or template just making it to compile and run.

Program Style and Documentation

This section is about program style and documentation.

1. For each file, the first piece should be a program header to identify yourself like this:

```
// -----------------------------------------------------------
// NAME : John Smith                         User ID: xxxxxxxx
// DUE DATE : mm/dd/yyyy
// PROGRAM ASSIGNMENT #
// FILE NAME : xxxx.yyyy.zzzz (your unix file name)
// PROGRAM PURPOSE :
//    A couple of lines describing your program briefly
// -----------------------------------------------------------
```

   Here, **User ID** is the one you use to login. It is ***not*** your social security number nor your M number.

   For each function in your program, include a simple description like this:

```
// -----------------------------------------------------------
// FUNCTION  xxyyzz : (function name)
//     the purpose of this function
// PARAMETER USAGE :
//    a list of all parameters and their meaning
// FUNCTION CALLED :
//    a list of functions that are called by this one
// -----------------------------------------------------------
```

2. Your programs must contain enough concise and to-the-point comments. Do not write a novel!
3. Your program should have good indentation.
4. Do not use global variables!

Program Specification

Your program must follow exactly the requirements of this programming assignment. Otherwise, you receive 0 point even though your program runs and produces correct output. The following is a list of potential problems.

1. Your program does not use the indicated algorithms/methods to solve this problem.
2. Your program does not follow the structure given in the specification. For example, your program is not divided into functions and files, etc when the specification says you should.
3. Any other significant violation of the given program specification.

4. **Incorrect output format.** This will cost you some points depending on how serious the violations are. The grader will make a decision. Hence, carefully check your program output against the required one.
5. **Your program does not achieve the goal of maximum parallelism.**

Program Correctness

If your program compiles and runs, we will check its correctness. We normally run your program with two sets of input data, one posted on this programming assignment page (the public one) and the other prepared by the grader (the private one). You program must deliver correct results for both data sets. Depending on the seriousness of the problem(s), significant deduction may be applied. For example, if your program delivers all wrong results for the public data set, you receive 0 point for that component.

The `README` File

A file named `README` is required to answer the following questions:

- The logic of your program
- Why does your program work?
- The meaning, initial value and the use of each variable. Explain why their initial values and uses are correct. Justify your claim.
- Answer the following:
  1. Can we use asynchronous channels to solve this problem? If you say "yes", prove it with a convincing argument. If you say "no", show a counter-example.
  2. If the last thread in the chain receives a number larger than the one this thread memorized, then the incoming number must be a prime number. Why is this true? Prove this fact with a convincing argument.
  3. Explain how you can fill the array **`Primes`** elements in a consecutive way.
  4. You do not need a mutex to protect the global array **`Primes`** when a prime thread is saving its memorized prime number? Prove this with a convincing argument.
- You must terminate your program gracefully. More precisely, The last three output lines must be printed by ***Master***.

You should elaborate your answer and provide details. **When answering the above questions, make sure each answer starts with a new line and have the question number (*e.g.*, Question X:) clearly shown. Separate two answers with a blank line.**

Note that the file name has to be `README` rather than `readme` or `Readme`. Note also that there is ***no*** filename extension, which means filename such as `README.TXT` is ***NOT*** acceptable.

`README` must be a plain text file. We do not accept files produced by any word processor. Moreover, watch for very long lines. More precisely, limit the length of each line to no more than 80 characters with the Return/Enter key for line separation. Missing this file, submitting non-text file, file with long lines, or providing incorrect and/or vague answers will cost you many points. Suggestion: Use a Unix text editor to prepare your `README` rather than a word processor.

Final Notes

1. Your submission should include the following files:
   1. File `thread.h` that contains all class definitions of your threads.
   2. File `thread.cpp` contains all class implementations of your threads.
   3. File `thread-main.cpp` contains the main program.
   4. File `Makefile` is a makefile that compiles the above three files to an executable file `prog6`. **Note that without following this file structure your program is likely to fall into the compile-but-**

   not-run category, and, as a result, you may get low grade. So, before submission, check if you have the proper file structure and correct makefile. Note that your `Makefile` **should not** activate the visualization system.
   5. File `README`.
2. **Always start early, because I will not grant any extension if your home machine, network connection, your phone line or the department machines crash in the last minute.**
3. **Since the rules are all clearly stated, no leniency will be given and none of the above conditions is negotiable. So, if you have anything in doubt, please ask for clarification.**
4. **Click here to see how your program will be graded.**