Experimentation with PThreads

The goal of this lab is to become familiar with using the pthreads library.

1 Introduction

In lecture, I have discussed the main PThread functions for creating threads, creating and manipulating mutexes, and creating and manipulating condition variables. The purpose of this lab is for you to implement a multi-threaded application using these primitives. I do not expect you to be able to complete this during the lab, but you should progress far enough so that you can complete it over the Xmas holiday break. You will need to be facile with PThreads for both OS3 and NS3 in the Spring Semester, so you are strongly encouraged to complete this lab. Additionally, if you are not taking NS3 and OS3 in the Spring, this lab will help you reinforce the lecture material as preparation for the May exam.

2 Problem Statement

You are to write a multi-threaded program that computes a specified number of prime numbers, starting at 1, and prints them out to standard output, one per line, in sorted order. Here is a single-threaded program, `primes.c`, which does this; it also logs elapsed time statistics on stderr.

primes.c

```
#include <stdio.h>
#include "isprime.h"
#include <sys/time.h>

int main(int argc, char *argv[]) {
   unsigned long i, limit;
   unsigned long count;
   unsigned long msec;
   double msperprime;
   struct timeval start, stop;

   if (argc == 1)
      limit = 100;
   else if (argc > 2) {
      fprintf(stderr, "usage: ./primes [limit]\n");
      return -1;
   } else
      sscanf(argv[1], "%lu", &limit);
   count = 0;
   gettimeofday(&start, NULL);          /* note start time */
   for (i = 1; count < limit; i++) {
      if (is_prime(i)) {
         printf("%lu\n", i);
        count++;
      }
   }
   gettimeofday(&stop, NULL);
   if (stop.tv_usec < start.tv_usec) {
```

```
      stop.tv_usec += 1000000;
      stop.tv_sec--;
   }
   msec = 1000 * (stop.tv_sec - start.tv_sec) +
                 (stop.tv_usec - start.tv_usec) / 1000;
   msperprime= (double) msec / (double) limit;
   fprintf(stderr, "%lu primes computed in %lu.%03lu seconds, %.3f ms/prime\n",
                limit, msec/1000, msec%1000, msperprime);
   return 0;
}
```

As you will have discerned, this program refers to a function, `is_prime()`; here are `isprime.h` and `isprime.c`:

isprime.h

```
#ifndef _ISPRIME_INCLUDED_
#define _ISPRIME_INCLUDED_

int is_prime(unsigned long number);

#endif /* _ISPRIME_INCLUDED_ */
```

isprime.c

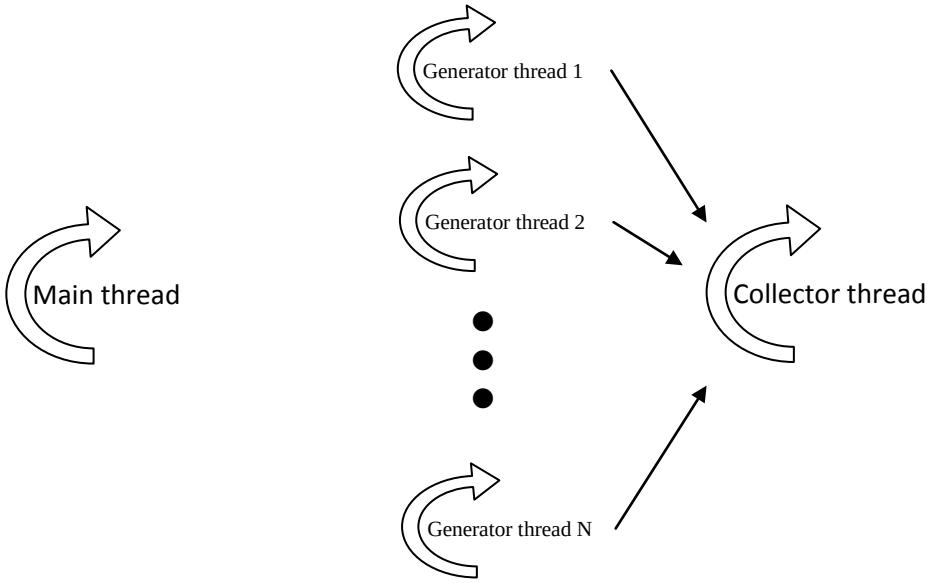
```
/*
 * is_prime() - returns 1 if argument is prime, 0 if not
 */

int is_prime(unsigned long number) {
   unsigned long tNum;
   unsigned long tLim;

   if (number == 1 || number == 2)
      return 1;
   if ((number % 2) == 0)
      return 0;
   for (tNum=3, tLim=number; tLim > tNum; tLim = number / tNum, tNum += 2) {
      if ((number % tNum) == 0)
         return 0;
   }
   return 1;
}
```

3 General Structure of the Multi-threaded Program

The program will consist of two mandatory threads, plus a variable number of threads that actually produce the prime numbers. The following diagram shows the threads and their relationships:



The responsibilities of these threads are as follows:

- Collector thread – it receives prime numbers from the generator thread[s]; it is responsible for collecting the prime numbers until a specified number of primes (limit) have been received, and then printing the collection of prime numbers in sorted order on standard output, one per line. It terminates after printing the collection of prime numbers.
- Generator thread – after being started, a generator thread obtains the next unexplored closed interval of the set of positive integers; for each integer in that interval which is prime, it gives it to the Collector thread. It continuously repeats this until interrupted by the Main thread. The set of Generator threads must safely share information about the next unexplored closed interval of integers.
- Main thread – it creates and initializes any data structures required for interactions among the other threads. It then creates the Collector thread, followed by creation of a specified number of Generator threads. It then waits for the Collector thread to terminate, followed by cancelling each of the Generator threads.

4 Specific Requirements

You are to write the program `mtprimes.c`; it must implement the following command line arguments:

```
./mtprimes [-b block] [-l limit] [-t threads]
```

where:

- **block** is the size of each closed integer interval that a Generator thread searches – e.g. if the next integer that is yet unexplored is N, and the block size is B, when a Generator thread obtains more work to do, it will be assigned the closed interval

[N..N+B-1], and the next integer that is yet unexplored will become N+B; the default block size is 1

- **limit** is the number of primes that the program is to find; the default limit is 100
- **threads** is the number the Generator threads that are to produce primes for consumption by the Collector thread; the default number of threads is 1

The program is also to provide summary timing statistics, as done in `primes.c` above. For example, my implementation prints the following at the end of the program when invoked as: `./mtprimes –b 100 –l 100000 –t 4`

```
limit/block/nthread = 100000/100/4, 2.015 sec, 0.020 ms/prime
```

The purpose of this lab is to focus on multi-threading issues. To assist you in writing `mtprimes.c`, the following snippet of code will enable you to process the command line arguments into local variables in `main()`[1]:

```
int i, j, nthread;
unsigned long block, limit;

block = 1;
limit = 100;
nthread = 1;
for (i = 1; i < argc; ) {
   if ((j = i + 1) == argc) {
      fprintf(stderr,
           "usage: ./mtprimes [-b block] [-l limit] [-t nthread]\n", USAGE);
      return -1;
   }
   if (strcmp(argv[i], "-b") == 0)
      sscanf(argv[j], "%lu", &block);
   else if (strcmp(argv[i], "-l") == 0)
      sscanf(argv[j], "%lu", &limit);
   else if (strcmp(argv[i], "-t") == 0)
      sscanf(argv[j], "%d", &nthread);
   else {
      fprintf(stderr, "Unknown flag: %s %s\n", argv[i], argv[j]);
      return -1;
   }
   i = j + 1;
}
```

5 Suggested Approach

As with all code development, you should work in small, bite-size chunks, making sure that each small addition is working correctly (i.e. you have thoroughly tested it) before you go onto the next addition. I suggest the following approach:

1. Thoroughly familiarize yourself with primes.c, making sure you understand what it is doing. As usual when converting a single-threaded application to a multi-threaded one, a portion of the code in the single-threaded program will become part of the code for one of your types of threads.

[1] You should be sure to understand why this code works correctly.

2. Write your main program to process the arguments correctly. This version of the program should just print out the variables after finishing the processing and terminate. In this way, you are now certain that the argument processing code is working correctly.
3. Now add the timing code; this should start AFTER you finish processing the arguments, and finish just before your `main()` returns. Note that we are not interested in timing if there are any errors in processing the arguments. More generally, we are not interested in timing if there are any errors in setting up the cooperating threads.
4. Now determine the thread-safe data structure that you are going to use between the Generator threads and the Collector thread. Implement that data structure; I would suggest some form of bounded buffer. You should develop another main program that you can use to test your data structure. Once you know that your data structure works correctly, add code to `mtprimes.c` to create one of these at the appropriate place. We won't use it yet.
5. Now add code for the Collector thread. You will need to figure out how to pass the identity of your shared data structure to it. You also need to be sure that everything that it will depend upon has been successfully set up before you invoke `pthread_create()`. In this version, the Collector thread should just print each prime as it receives it through the data structure. Assuming the default limit, have the Main thread send 100 integers through the data structure; the Collector thread should terminate, and the pthread_join of the Main thread with the Collector thread should complete successfully.
6. Now add code for a Generator thread. You will need to figure out how multiple Generator threads will safely share knowledge of the next unexplored integer block. You also need to figure out how to pass the identity of the data structure shared with the Collector thread to it. You should then create a single Generator thread and make sure that it interacts correctly with the Collector. This is actually the hardest part of this lab, so take your time and be sure to use gdb when it inevitably does not work.
7. Now you can add code that creates the specified number of Generator threads. Leave the Collector to simply print each prime as it receives it. Since the threads can be arbitrarily scheduled by the pthread library, the primes will not be printed in sorted order. Make sure that this works correctly by piping the output from `mtprimes` into `sort` and comparing sort's output to the appropriate TenTo*.out file provided on Moodle.
8. Now you have to change the Collector code to actually store the primes as received, using a data structure to keep the collected primes in sorted order. Alternatively, you can keep them in arrival order, only sorting the entire set after the Collector has received "limit" primes. You might find the heap structure that we discussed in lab 3 of use here if you are attempting to keep the set ordered as each one arrives.