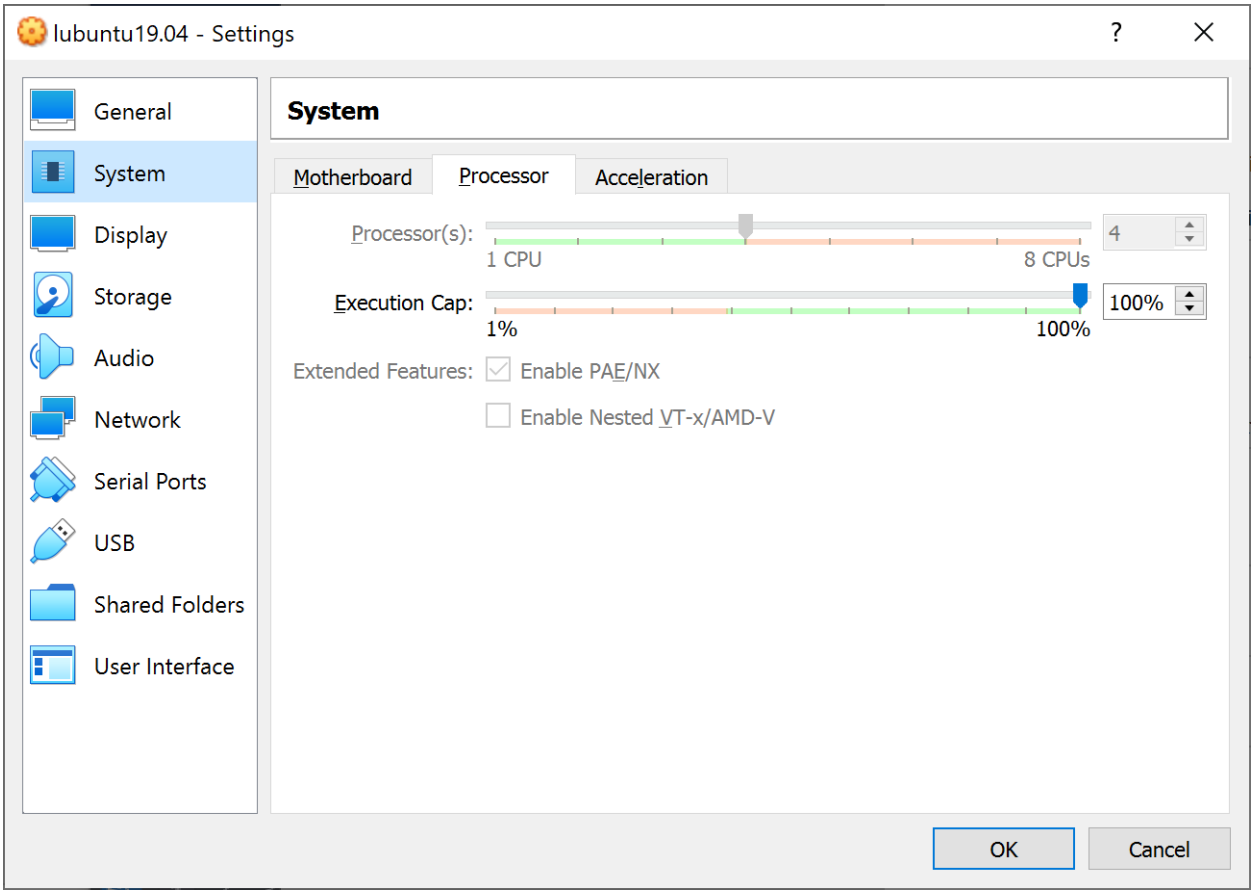
CS 392, Systems Programming: Assignment 6
Multithreaded Primes Sieve

Overview

This assignment requires you to implement the segmented sieve of Eratosthenes to find prime numbers having two or more digits that are '3'. Unlike the standard sieve of Eratosthenes, that operates over the range [2, b], the segmented version can start at an arbitrary lower bound a, working over the range [a, b]. Since a wide range of values can be programmed in segments, this algorithm lends itself nicely to a multithreaded approach.

VirtualBox Settings

Before starting this assignment, increase the number of processors available to the virtual machine. Drag the slider up to the end of the green segment. The value shown at the end of the green segment indicates the number of true cores (not threads) available on your CPU.



Algorithm: Segmented Sieve of Eratosthenes

```
Purpose: Used to find primes in range [a, b].

Steps:

1. Use standard sieve of Eratosthenes to find all primes up to and including
sqrt(b), call them low_primes[]

2. Create a boolean array high_primes[] with length = b - a + 1 and
initialize each element to true

3. for each prime p in low_primes[],

       set i = ceil((double)a/p) * p - a

       if a <= p

            set i = i + p

       starting at i, cross off all multiples of p in high_primes[]

4. for each high_primes[i] that is true, print i + a
```

Specifications

Command-line Arguments

Your program will be named `mtsieve`, short for multithreaded sieve. It will take use getopt to parse three command-line arguments. There are numerous ways for the user to incorrectly supply the arguments, so error checking is a bit tedious. Any usage or error message should be paired with having the program return in EXIT_FAILURE.

First, if too few arguments are supplied, the program will display the usage message:

```
Usage: ./mtsieve -s <starting value> -e <ending value> -t <num threads>
```

For the case involving an unknown option or option missing a required argument, use this code:

```
case '?':
    if (optopt == 'e' || optopt == 's' || optopt == 't') {
        fprintf(stderr, "Error: Option -%c requires an argument.\n", optopt);
    } else if (isprint(optopt)) {
        fprintf(stderr, "Error: Unknown option '-%c'.\n", optopt);
    } else {
        fprintf(stderr, "Error: Unknown option character '\\x%x'.\n",
                optopt);
    }
    return EXIT_FAILURE;
```

Inside the getopt loop, if any argument is not an integer, the program should output:

```
"Error: Invalid input '%s' received for parameter '-%c'.\n"
```

Where %s is the value received and %c is the option that was being parsed, as in -s, -e, or -t.

Similarly, if the integer argument is loo large, the program should output:

```
"Error: Integer overflow for parameter '-%c'.\n"
```

After processing all argument with the getopt loop, the following conditions need to be checked, in this order:

```
"Error: Non-option argument '%s' supplied.\n"

"Error: Required argument <starting value> is missing.\n"

"Error: Starting value must be >= 2.\n"

"Error: Required argument <ending value> is missing.\n"

"Error: Ending value must be >= 2.\n"

"Error: Ending value must be >= starting value.\n"

"Error: Required argument <num threads> is missing.\n"

"Error: Number of threads cannot be less than 1.\n"

"Error: Number of threads cannot exceed twice the number of
processors(%d).\n"
```

The last error message is particularly interesting. We do not wish to create more threads than twice the number of available processors on the system. How do we find out this information through our program? Use `man 3 get_nprocs`.

Creating Segments

After validating all the command-line arguments, your program needs to decide how to segment the range of values. First, compute how many numbers are being tested for primality. If the number of threads exceeds the count, reduce the number of threads to match the count.

Otherwise, take the count and divide it by the number of threads the user requested to create. Each thread will process at least that many numbers. Take the remainder and distribute it among all the threads. Unless the thread number divides the count evenly, the later segments will contain less values.

The program must print the number of segments followed by lines showing the lower and upper bounds of each segment in square brackets. See the examples at the end of this document.

Threads

You will need two global variables:

```
int total_count = 0;

pthread_mutex_t lock;
```

Start the appropriate number of threads and have each one run the algorithm over its desired range. Since the thread's function takes in single void pointer, you may wish to create a struct for each thread that encapsulates all the information the thread will need. In this case, it can be as simple as:

```
typedef struct arg_struct {

    int start;

    int end;

} thread_args;
```

As each thread finds prime numbers meeting the condition, it will increment the global variable total_count. Of course, to guarantee that no race conditions occur, proper use of the mutex must be ensured.

The program will then wait for all the threads to terminate and, immediately afterwards, destroy the mutex.

Before exiting, the program will print

```
"Total primes between %d and %d with two or more '3' digits: %d\n"
```

where the first %d is the starting value, the second %d is the ending value, and the third %d is the count.

Note: You **must** check the return value of all function calls involving threads and mutexes. If pthread_join, pthread_mutex_lock, pthread_mutex_unlock, or pthread_mutex_destroy fails, print a warning message but do not exit the program. If any other function call fails, the program cannot proceed and should print an error message as well as return EXIT_FAILURE. You may choose the text for these messages.

Sample Executions

```
$ ./mtsieve
Usage: ./mtsieve -s <starting value> -e <ending value> -t <num threads>

$ ./mtsieve -s
Error: Option -s requires an argument.

$ ./mtsieve -s2
Error: Required argument <ending value> is missing.

$ ./mtsieve -e -s2
Error: Invalid input '-s2' received for parameter '-e'.

$ ./mtsieve -e20 -s2
Error: Required argument <num threads> is missing.
```

```
$ ./mtsieve -e20 -s2 -t
Error: Option -t requires an argument.

$ ./mtsieve -e20 -s2 -tfour
Error: Invalid input 'four' received for parameter '-t'.

$ ./mtsieve -e20 -s2 -t2200000000
Error: Integer overflow for parameter '-t'.

$ ./mtsieve -s2 -e2 -t8 HI BYE
Error: Non-option argument 'HI' supplied.

$ ./mtsieve -e20 -s2 -t4
Finding all prime numbers between 2 and 20.
4 segments:
   [2, 6]
   [7, 11]
   [12, 16]
   [17, 20]
Total primes between 2 and 20 with two or more '3' digits: 0

$ ./mtsieve -s2 -e2 -t4
Finding all prime numbers between 2 and 2.
1 segment:
   [2, 2]
Total primes between 2 and 2 with two or more '3' digits: 0

$ ./mtsieve -s1000000 -e200000 -t4
Error: Ending value must be >= starting value.

$ time ./mtsieve -s1000000 -e2000000 -t4
Finding all prime numbers between 1000000 and 2000000.
4 segments:
   [1000000, 1250000]
   [1250001, 1500000]
   [1500001, 1750000]
   [1750001, 2000000]
Total primes between 1000000 and 2000000 with two or more '3' digits: 11477

real    0m0.005s
user    0m0.000s
sys     0m0.007s

$ time ./mtsieve -s1000 -e20000000 -t1
Finding all prime numbers between 1000 and 20000000.
1 segment:
   [1000, 20000000]
Total primes between 1000 and 20000000 with two or more '3' digits: 257814

real    0m0.187s
user    0m0.169s
sys     0m0.017s
```

```
$ time ./mtsieve -s1000 -e20000000 -t8
Finding all prime numbers between 1000 and 20000000.
8 segments:
   [1000, 2500875]
   [2500876, 5000750]
   [5000751, 7500625]
   [7500626, 10000500]
   [10000501, 12500375]
   [12500376, 15000250]
   [15000251, 17500125]
   [17500126, 20000000]
Total primes between 1000 and 20000000 with two or more '3' digits: 257814

real    0m0.050s
user    0m0.154s
sys     0m0.011s
```

Submission Requirements

Submit a zip file called mtsieve.zip containing mtsieve.c, makefile, and runtimes.txt. You may choose to factor out functions into separate .c/.h files. Any structure is acceptable as long as your makefile can build the project.

The content of runtimes.txt should be a summary of the results of running the following command:

time ./mtsieve -s100 -e200000000 -t<vary from 1 through the max value your CPU allows>

Take just the real time:

```
$ time ./mtsieve -s100 -e200000000 -t4
Finding all prime numbers between 100 and 200000000.
4 segments:
   [100, 50000075]
   [50000076, 100000050]
   [100000051, 150000025]
   [150000026, 200000000]
Total primes between 100 and 200000000 with two or more '3' digits: 2689007

real    0m1.165s
user    0m4.509s
sys     0m0.056s
```

In runtimes.txt, list the command, followed by the real time:

```
./mtsieve -s100 -e200000000 -t1 : 0m2.068s

...

./mtsieve -s100 -e200000000 -t4 : xxxxxxxx

...
```

```
./mtsieve -s100 -e200000000 -tn : xxxxxxxx
```

Then answer the following questions:

What CPU do you have? Use your previous lab `cpumodel` to answer that question. Does the time scale linearly with the number of threads? If not, does it continue to improve, plateau, or start to decline? Explain your findings in just a few sentences.