TCSS 422: Operating Systems
Fall 2021
http://faculty.washington.edu/wlloyd/courses/tcss422

School of Engineering and Technology
University of Washington – Tacoma
Instructor: Wes Lloyd

Assignment 2

Prime Number Generation: Sieve of Eratosthenes

Due Date: Friday Dec 3rd, 2021 @ 11:59 pm AOE, tentative
Version: 0.10

Note: *This assignment must be completed on a computer with a minimum of two virtual CPU cores (hyperthreads ok). Four to eight CPU virtual cores are recommended. It is recommended to close the web browser while running the program. If the browser cannot be closed, it is recommended to close as many tabs as possible.*

Objective

The purpose of this assignment is to implement a bounded buffer data structure that incorporates locks and condition variables to support prime number generation using a bounded-buffer-based implementation of the Sieve of Eratosthenes *(err-ah-taas-thuh-neez)* algorithm. The source code for the bounded-buffer-based implementation of Sieve of Eratosthenes, a variation on the standard implementation, is provided. Unlike the usual implementation that tracks prime number generation using a Boolean array, this implementation use pthreads and bounded buffers.

The parent thread generates integers starting with 2. The parent passes these numbers using a bounded buffer to the first pthread (p1). This pthread prints out "2" as the prime number it generates and then passes on any integers it finds in the list from the parent from the bounded buffer that are not divisible by 2 to the next pthread (p2). *Essentially, P1 feeds its numbers to p2 using a new bounded buffer.* P2 then prints out "3" and the prime number it generates, and then passes on the integers from its bounded buffer that are not divisible by 3 to the next pthread (p3). This recursive iteration continues to generate every prime less than n, using n threads and n bounded buffers. This program provides a resource intensive solution to rapidly generate prime numbers. The major drawback is overhead from creating pthreads and the use of lock APIs. ***Note:*** *that this method of generating prime numbers is not very efficient.* It is not meant to be. Instead, this program provides a resource intensive program that can be used to stress the system's ability to create pthreads and use locking APIs.

Starter code is provided that implements the Sieve of Erathosthenes algorithm in C. Your task is to implement the required lock-safe Bounded Buffer data type to enable the program to work correctly.

The structure of the bounded buffer is given as follows:

```
typedef struct __buffer_t {
  int * buffer;           // ptr to content of bounded buffer
  int maxSlots;           // size of bounded buffer
  int use_ptr;            // next item to consume
  int fill_ptr;           // next location to fill
  int count;              // number of elements in bounded buffer
  char * name;            // unique name of this bounded buffer (for debugging purposes)
  pthread_mutex_t mutex;  // this buffer's mutex lock variable
  pthread_cond_t cond;    // this buffer's condition variable
} buffer_t;
```

The objective is to implement the methods:
`void initbuff(int nslots, buffer_t *buff, char * n);`

This method is used to initialize the bounded buffer. All elements defined in the structure should be initialized. Memory should be allocated for the buffer using malloc(). The maxSlots variable should be set to nslots. The initialization buffer should initialize the buffer pointed to by buff. The name variable can optionally be set to the String "n". This can be helpful for debugging to print out the specific bounded buffer being acted upon. **A "&" symbol will preface the use of the mutex and cond variables when calling pthread API methods.**

`void freebuff(int nslots, buffer_t *buff, char * n);`

This method is used to release the memory allocated to the buffer variable using the free() API. Implementation is *optional* as memory should automatically be free'd when the program exits.

`void put(int value, buffer_t *buff);`

This method is used to load a new value into the bounded buffer. It is recommended to see the sample code in section 30.2 of the text book to get ideas on the implementation. For this program, do not write a separate producer() and consumer() method. Instead, combine the put() and producer() into one method, and combine the get() and consumer() into one method.

`int get(buffer_t *buff);`

This method is used to retrieve a value from the bounded buffer. It is recommended to see the sample code in section 30.2 of the text book for an idea on the implementation. For this program, do not write a separate producer() and consumer() method. Instead, combine the put() and producer() into one method, and combine the get() and consumer() into one method.

Starter code is online at:
http://faculty.washington.edu/wlloyd/courses/tcss422/assignments/prime_gen.tar.gz

This program must be implemented in C.

For this assignment, the use of semaphores is not permitted. Any solution using semaphores will be returned and not graded. Solutions should use the locking constructs: **`pthread_mutex_t`** and **`pthread_cond_t`**.

The following source code is provided:

C Module	Header file	Source File	Description
buffer	buffer.h	buffer.c	Synchronized bounded buffer skeleton data structure See Ch. 30.2 for example code
prime	none	prime.c	Program main module with worker() routine
Makefile	n/a	n/a	C Makefile for project

As-is the provided code generates the first prime number, 1.

The program accepts a command line argument to dynamically set "n", which will generate all prime numbers up til n. On most systems, it is recommended to keep "n" at 50,000 or below. When developing you'll likely want to reduce "n" to a low value such as 10 or 100.

This prime numbers solution has the benefit of being relatively fast, and extremely parallel. The algorithm does not work well when looking to generate a lot of primes because the algorithm is recursive using pthreads. This recursive implementation creates a new pthread for each recursive call. The memory and kernel API overhead associated with creating a very large number of threads becomes unwieldy for the system to cope.

Use of system resources can be monitored using Linux commands:

`top -H -d <refresh-interval-in-seconds>`	can be used to inspect the active number of running threads on the system, as well as the system load average

The Linux "pagemon", or page monitor can be helpful to monitor the size of memory when the program runs. This program requires knowing the program's process ID.

Add the following code at the start of int main() to obtain the processID before generating any prime numbers:

```
  char prompt[100];
  printf("PID=%d\nPress [ENTER] to continue\n",getpid());
  scanf("%s",prompt);
```

Using the process ID, pagemon can be used to visualize the use of program memory as follows:

```
sudo apt install pagemon
pagemon -d <process-id>
```

If no command line arguments are provided, the default value of n=50,000 is used. The program will attempt to generate all prime numbers less than 50,000. Alternatively, a value can be provided to restrict output using the command line:

```
$ ./prime 110
main: start
PRIMES:
1 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109
main: end
```

The source code can be built using Make. To purge old files use the "clean" target:

```
$ make clean
rm -f -f prime *.o
```

To compile the project, simply type "make":

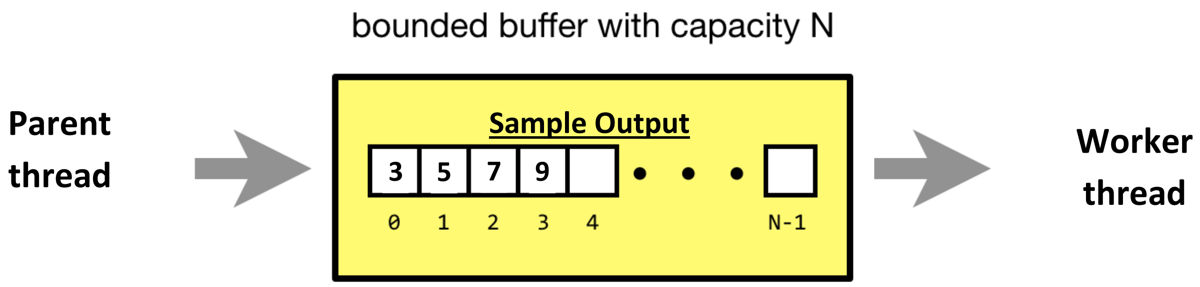
```
$ make
gcc -pthread -I. -Wall -Wno-int-conversion -D_GNU_SOURCE buffer.c prime.c -o prime
```

Lock and Condition Variable Recommendations

There are some nice things about the implementation of this program. If done correctly, it is reasonably clean. Essentially each pair of threads, let's say the parent thread and p1, will communicate using 1 bounded buffer. Then p1 and p2 will communicate using a new bounded buffer. For each new pthread we create, a new bounded buffer is created to facilitate communication with the parent thread. The advantage with this implementation is that only two pthreads share each bounded buffer. This design should limit the complexity of the solution. If implemented correctly, however, it should be easy to max out all available CPU and operating system thread resources when running this program. This will result in a very high Thread Level Parallelism (TLP) value for the program, that is the average number of concurrent threads that run throughout the duration of the program. On a multicore machine, when monitoring load with "`top -d .1`", the max percent CPU utilization demonstrates the highest degree of parallelism achieved. On an 8-hyperthead computer, near 800% is possible if closing all other applications. On a 4-hyperthread computer, near 400% is possible. **It is recommended to increase the number of virtual CPUs on your VM for testing purposes to the highest supported value for this assignment.**

Program Testing Recommendations

For testing correctness of concurrent programming, try out different sizes of the bounded buffer (MAX). If the bounded buffer is too large, this could minimize errors, and hide possible concurrency problems. The graders will test a variety of bounded buffer sizes from 1 to 1,000 or more.



Starting Out

As a starting point for assignment 2, it is recommended to become familiar with the the signal.c example from chapter 30. In addition, review chapter 30.2. This provides a working implementation of a bounded buffer in C. The signal.c example program stores matrices in a bounded buffer of size 1. The signal.c example can be found here:
http://faculty.washington.edu/wlloyd/courses/tcss422/examples/Chapter30/

Points to consider:

1. Each bounded buffer will store numbers which will be checked to identify primes. Each worker thread produces only one new prime number. The worker prints the value of it's prime at the start of the worker routine. The use of signals is required to inform consumer threads (here they will be child threads) when there are numbers available to consume to check for primes, and to signal the producer when there is available space in the bounded buffer to add more numbers. For testing, reducing the size of the bounded buffer to a low number, for example 2 is recommended to test for deadlock.

2. Put() will add a number to the end of the bounded buffer. Get() retrieves a number from the other end.

3. This program will require the use of both locks (mutexes) and condition variables for synchronization.

4. On some computers, generating all prime numbers (n<50000) may result in running out of system resources (either memory of available threads). Reduce n in this case or increase the memory of the virtual machine. Releasing memory of the bounded buffer using free() should be optional at the end of the program.

Grading

Rubric:
100 possible points: (15 extra credit points available)

Functionality Total: 50 points

10 points	Generates all prime numbers less than 100
10 points	Generates all prime numbers less than 1,000
10 points	Generates all prime numbers less than 10,000
10 points	Generates all prime numbers less than 50,000
10 points	Program does not deadlock

Written Report: 40 points

40 points	PDF written report that adequately answers questions

Miscellaneous Total: 10 points

5 points	Program compiles without errors, makefile working with all and clean targets
5 points	Coding style, formatting, and comments

Extra Credit: 15 points MAX

Using the same recursive algorithm using pthreads, mutex_locks, and condition variables, reliably generate additional prime numbers up to 100,000 or 500,000. The program must not deadlock or fail under any condition for extra credit and must generate and display all prime numbers below the threshold. Instructions must be provided for how to achieve the result on an Oracle Virtual Box Ubuntu 20.04 VM with 4 virtual CPU cores and 8 GB of RAM.

2 points	EC1 – Program reproducibly generates all prime numbers less than 100,000 To receive extra credit, generate all primes <100,000. Using the Linux time command, capture and report the runtime in a README PDF or text file. Identify the # of CPU cores and memory of the Linux machine used for the test.
3 points	EC2- Program reproducibly generates all prime numbers less than 500,000 To receive extra credit, generate all primes <500,000. Using the Linux time command, capture and report the runtime in a README PDF or text file. Identify the # of CPU cores and memory of the Linux machine used for the test.
10 points	EC3- Provide two modes of operation for the program. "**Standard**" mode is as described in the assignment. "**Preinitialize**" mode involves preinitializing approximately 5,133 pthreads before their use to generate the first 50,000 prime numbers. The mode should be specified using a second command line argument after the number of primes to be generated. If no option is specified, standard mode is used. Use the command line argument value of "1" after the number of primes to specify standard mode. Use "2" to specify "preinitialize" mode. After implementing both modes using the Linux time command to

time and then report in a README PDF or text file how long it takes to generate the first 50,000 prime numbers using both modes:

```
# standard mode (can be invoked two ways)
$ time ./prime 50000
$ time ./prime 50000 1
# preinitialize mode
$ time ./prime 50000 2
```

**** - EXTRA CREDIT- COMMENTS ARE REQUIRED:***

To obtain extra credit, a comment must appear that identifies which extra credit features have been implemented (EC1, EC2, and/or EC3) at the top of prime.c:

*Example of **required** comment:*

```
// EXTRA CREDIT IMPLEMENTED: features EC1 and EC3
```

All programs should provide instructions to explain any special configuration required to properly test the program. Undocumented solutions that require special configuration that do not run on an Oracle Virtual Box Ubuntu 20.04 VM with 4 virtual CPU cores and 8GB of RAM will not receive extra credit. It is necessary to inform the graders of any special instructions/configuration.

WRITTEN REPORT

For the written report, provide answers for the following questions:

1. Describe the trade-off for adjusting the size of the bounded buffer for the program. What do you observe when testing the program using a small bounded buffer size? What do you observe when testing the program using a large bounded buffer size? What bounded buffer size appears best for fastest program execution on your computer and why?

2. Provide your computer or virtual machine specs used to answer question #1. Include the following:
    a. Processor name (check with lscpu, please identify type such as Intel(R) Core(TM) i7-1234HQ CPU @ 9.99GHz)
    b. Available number of processors cores (check with lscpu)
    c. Available memory (check with free -m)

3. If your program is not working entirely, please provide a detailed description of what is working based on the rubric requirements, and what is NOT working. This description will be used to grant partial credit in the case that not all elements are working.

4. Describe how you've tested your program for deadlock.

5. <OPTIONAL> Detailed description of novel extra credit feature(s).

What to Submit

For this assignment, submit a tar gzip archive as a single file upload to Canvas.

Please do not submit zip files.

Tar gzip archive files can be created by going back one directory from the source directory:

```
cd ..
tar czf primes.tar.gz prime_gen/
```

Upload this file to Canvas. Canvas automatically adds student names to uploaded files.

Upload the PDF for the written report to Canvas separately.

Pair Programming (optional)

O*ptionally*, it is encouraged to complete this programming assignment with **<u>one or two</u>** person teams.

If choosing to work in pairs, ***only one*** person should submit the team's tar gzip archive to Canvas.

Additionally, ***EACH*** member of a pair programming team must provide an **effort report** of team members to quantify team contributions for the overall project. **Effort reports** must be submitted INDEPENDENTLY and in confidence (i.e. not shared) by each team member to capture each person's overall view of the teamwork and outcome of the programming assignment. <u>Effort reports are not used to directly numerically weight assignment grades.</u>

Effort reports should be submitted in confidence to Canvas as a PDF file named: "effort_report.pdf". Google Docs and recent versions of MS Word provide the ability to save or export a document in PDF format. Distribute 100 points for category to reflect each teammate's contribution for: research, design, coding, testing. Effort scores should add up to 100 for each category. Even effort 50%-50% is reported as 50 and 50.

<u>Please do not submit 50-50 scores for all categories.</u>

Programs effort reports with even 50-50 scores for all categories will not be graded. It is highly unlikely that effort is equal for everything. Ratings must reflect an honest confidential assessment of team member contributions.

Here is an **<u>effort report</u>** for a pair programming team (written from the point of view of Jane Smith):

1. John Doe

Research	24
Design	33
Coding	71
Testing	29

2. Jane Smith

Research	76
Design	67
Coding	29
Testing	71

TCSS 422 effort reports should include a **short description** of how pair programming was conducted virtually. The description should LIST tools that were used and how they facilitated pair programming. Some possible tools include: Google Hangouts, Discord, Zoom, and Slack.

Team members may not share their **effort reports**, and should submit them independently in Canvas as a PDF file. Failure of one or both members to submit the **effort report** will result in both members receiving NO GRADE on the assignment… (*considered late until both are submitted*)

Disclaimer regarding pair programming:
The purpose of TCSS 422 is for everyone to gain experience programming in C while working with operating system and parallel coding. Pair programming is provided as an opportunity to harness teamwork to tackle programming challenges. But this does not mean that teams consist of one champion programmer, and a second observer that only passively participates! Tasks and challenges should be shared as equally as possible to maximize learning opportunities.

Change History

Version	Date	Change
0.1	11/15/2021	Original Version