Project #1: Concurrent Processes, IPC, and Threads

Assigned: Oct 04, 2023 Document version: 1.0
Due date: Oct 20, 2023

This project will be done in groups of two students. You can do it individually as well. You will program in C/Linux. Programs will be tested in Ubuntu Linux. No deadline extension will be provided. Therefore, please start as soon as possible.

Objectives: *Practice with process creation, thread creation, multi-threaded programming, multi-process programming, inter-process communication, message queues.*

In this project you will develop an application that will find the prime numbers in a sequence of positive integers included in an input file by using multiple processes or threads. In part A, the application will use multiple child processes, in part B, multiple threads. Therefore, you will develop two programs. The input file will have a sequence of positive integers, one integer per line. An integer value may appear multiple times in the input file. Your program will write the numbers that are prime (i.e., the result) to an output file. A prime number will appear in the output file multiple times, if it exists in the input file multiple times. The output does not have to be sorted. But each integer (prime number) must appear in a separate line in the output file (the first integer in the first line, the second integer in the second line, and so on).

1. Part A: Concurrent Processes and Message Queues (50 pts)

In this part you will write a program that will create and use **multiple processes** to find out the numbers that are prime. The program will be called `primeP` and it will have the following command line arguments:

```
primeP -n N -m M -i INFILE -o OUTFILE
```

The `-n N` option is used to specify the number of child processes that will work concurrently to find out the prime numbers. The minimum value of `N` is 1, the maximum value is 20. The default value is 5. The `-m M` option is used to specify how many prime numbers will be put (packed) into a single message that will sent to the message queue. Minimum value of `M` is 1, maximum value is 20. Default value is 3. The `-i INFILE` option is used to specify the name of the input file. `-o OUTFILE` option is used to specify the name of the output file, which will store the result. In the rest of the document, N will denote the specified value of `N`, and M will denote the specified value of `M`.

Your program will start as a single process (parent). The parent process will first open the input file, read it from beginning to end, and will **split** the content into N *intermediate* text (ascii) files (file 1, file 2, ..., file N). The k^{th} integer ($k \geq 1$) in the input file will go to the intermediate file $((k-1)\%N)+1$. For example, if N is 3, the first integer in the input file will go to file 1, and the 5^{th} integer will go to file 2. After input is split, the parent will create a Linux POSIX **message queue** that will be used for IPC (inter-process communication). The name (descriptor) of the message queue should be stored in a global variable so that child processes can access the same message queue by using that name or descriptor.

Then the parent will create N **child processes** to process N intermediate input files. Each child will read and process another intermediate input file. A child can read its input file one integer at a time, and decide if the integer is a prime or not (with any algorithm that you like). If the integer is not a prime, it will be simply ignored. If the integer is a prime number, the child process will put it into a message. A **message** created at the child process will contain M prime numbers, except the last message, which may contain less than M prime numbers. When a message is full (has M prime numbers), it will be sent to the message queue. In this way, all children will pass their prime numbers to the parent via the single message queue.

The parent process, after creating the children, will start reading the messages from the message queue. These messages contain prime numbers sent from the child processes. The parent process and child processes will run *concurrently*; while children produce prime numbers and send them to the message queue, the parent will read those prime numbers and write them out to the output file.

Note that we can use *automated* tools to test your program and output. Therefore, your output format must strictly follow the specification here. We will also look to your source code, to see if the processes, the message queue, and the threads are created and used properly.

When all children finish finding the prime numbers and terminate, the parent process will also terminate after writing all prime numbers received from the message queue to the output file. Before termination, the parent process must *delete (remove)* all intermediate input files and the message queue.

An example input and output of the program can be as follows:

Input file content:

```
10
17
3
1
3
```

```
9
```

Output file content:

```
17
3
3
```

The order of prime numbers in the output files does not have to be the same with their order in the input file. Therefore, the following is also an acceptable output.

```
3
17
3
```

We will use the Linux **sort** command to sort your output while testing your programs.

An example invocation of the program can be as follows:

```
./primeP -n 10 -m 10 -i infile.txt -o outfile.txt
```

2. Part B: Threads (30 pts)

In this part, you will do the same project by using threads, instead of child processes. The name of the program will be `primeT` in this case. The program will have the following parameters. Their meaning is the same as their meaning in `primeP` program. In this part, you do not need to use message queues, since all threads can share and access global variables.

```
primeT -n N -i INFILE -o OUTFILE
```

Let *N* denote the value of `N` specified at the comment line. Again, when started, your program will first split the input file into *N* intermediate input files. Then *N* threads will be created. Each intermediate input file will be processed by another thread (worker thread). While reading its input file, a thread will place the integers that are prime to a linked list. Each thread will have its own linked list (pointed by a global variable). In a node of the list, you can keep a prime number and its frequency (how many times it appeared in the input file).

When all worker threads finish their work (finished processing their input files) and terminate, the main thread (parent process) will read those lists and will print the prime numbers to the output file. It will then remove all intermediate files. For this part, you can assume that the input file will not contain an integer whose value is greater than 10000 (10 thousand).

3. Experiments (20 pts)

Run your program with various values of *N* and see how long it takes for the program to complete. Do you see a performance increase, when *N* is increased? Explain why you see a performance increase, or why you do not see a performance increase. Do such experiments with both of your programs. Also do some experiments to see the effect of *M*.

At the end write a **report**, explaining your experiments and results. Convert the report to a PDF file. You will upload this PDF file as part of your submission.

4. Submission

Put all your files into a directory named with your Student ID. If the project is done as a group, the IDs of both students will be written, separated by a dash '-'. In a `README.txt` file, write your name, ID, etc. (if done as a group, all names and IDs will be included). The set of files in the directory will include `README.txt`, `Makefile`, and program source files. We should be able to compile your programs by just typing `make`. No binary files (executable files or .o files) will be included in your submission. Then **tar** and **gzip** the directory, and submit it to **Moodle**.

For example, a project group with student IDs `21404312`, `214052104` will create a directory named `21404312-214052104` and will put their files there. Then, they will tar the directory (package the directory) as follows:

tar cvf 21404312-214052104.tar 21404312-214052104

Then they will gzip (compress) the tar file as follows:

```
gzip 21404312-214052104.tar
```

In this way they will obtain a file called `21404312-214052104.tar.gz`. Then they will upload this file to **Moodle**. For a project done individually, just the ID of the student will be used as a file or directory name.

5. Tips and Clarifications

- Start early, work incrementally.
- There is no limit on the number of integers that can exist in the input file. You should not try to store them all in memory.
- **POSIX message queue** API will be used; not System V message queue API.
- `man mq_overview` will give you information about Linux POSIX message queue API (related functions). You can find more information on the Web.
- You will use **POSIX threads** (Pthreads) in your program (part B). To get information about threads, you can type `man pthreads` on Linux command line. You can also find a lot of information in Internet.

- The filename length limit for input/output files is 64 characters (including the NULL character at the end).