Document Version: 1.0 (a newer version number means an update on the project)
Assignment Date: 10/24/2022, Noon
Due Date - No Penalty: 11/14/2022, 11:59pm
Due Date - 10% Penalty: 11/21/2022, 11:59pm
No submission is accepted after 11/21/2022, 11:59pm
Group Size: 1 or 2 Students; single submission for each group

Objectives:

- Continue writing non-trivial C programs
- Exercise with process/thread creation and handling
- Exercise with thread synchronization using pthread mutex locks
- Exercise with the setup and use of message passing as an IPC facility
- Exercise with multi-processing and multi-threading as well as their performance evaluation

1 Project Description

In this project, you will develop three different programs that will all do the same calculation in three different ways: sequentially (`seq`), in parallel using multiple threads (`par_t`), and in parallel using multiple processes (`par_p`). Your goal is to achieve a better performance from the parallel versions compared to the sequential one. **You will achieve performance improvement by only spreading the work on multiple processors, not changing the algorithm or the calculation that you performed. Intentionally slowing down the sequential version, such as using sleep statements or unnecessarily performing inefficient calculations only in the sequential version, different than the ones used in the parallel versions, will be considered plagiarism.**

The calculations that your programs will perform is very simple. You will be given a positive integer `K`, a `directoryPath`, and an `outputFile` name. The `directoryPath` includes `N` input files of integers (one integer per line) in its first level. Your program will calculate the top `K` **unique** integers appearing in these `N` input files and write them in sorted decreasing order to the `outputFile`, one integer per line. The same integer can appear multiple times in the same file or in different files but they should be discarded as the output should only include **unique** integers.

Your program will only go over the first level files in the given `directoryPath`. If the directory includes any sub-directories, any files inside these sub-directories as well any files starting with a (".") or ending with a "~" should be ignored. You will read the files using `fscanf()`, and find the `K` largest integers using a linked list data structure (not an array). Your linked list will have at most *K* nodes, and it will maintain top `K` integers in sorted decreasing order.

For this project, you are allowed to work in groups of at most 2 members. No more than two students can work together; however, if you prefer to work alone, then you are allowed to do so. Each group will make a single submission. Here are the three programs that you will implement:

1.1 Sequential

The sequential version will be invoked as follows:

```
./seq <K> <directoryPath> <outputFile>
```

In this version, there will only be a single thread/process performing the entire calculation. Therefore, all `N` files will be read sequentially (one by one), one integer at a time using `fscanf()`, and the top `K` integers will be maintained in a single linked list data structure in sorted decreasing order using at most *K* nodes.

1.2 Parallel using Multiple Threads

The multi-threaded version will be a separate program and invoked as follows:

```
./par_t <K> <directoryPath> <outputFile>
```

In this version, the main thread should create `N` worker threads, and each worker thread will read a separate input file one integer at a time using `fscanf()`. The program will have a shared/global linked list data structure (pointed by a global variable) to calculate the top `K` integers, again using at most `K` nodes.

Since multiple threads will access this shared global linked list, you will need to synchronize their access using pthread mutex locks. **Check the *solution-mutex.c* example provided in the class website for the usage of pthread mutex locks. Check also the *mythreads.h* file that it includes, and incorporate it into your program, but do use it by directly including it, especially avoid any cpu binding code that the *mythreads.h* file includes.** In addition to the lecture materials, there are two links provided at the end of the class website for further details/tutorials on pthreads.

Lock contention is a potential cause of slow-down in the multi-threaded version, especially if multiple threads acquire and release the lock to access the shared global linked list very frequently. Due to lock contention, you parallel version can be even slower than the sequential one. In order to alleviate lock contention, per-thread calculations can be performed using local linked lists first, using one local linked list per thread, and then these local linked lists can be used to fill the content of the global linked list using locks. In your README file, you will mention what you did to eliminate lock contention.

All `N` worker threads will do the same task concurrently, working on a different files in parallel. After reading an integer from the input file, the worker thread will either insert it to the linked list or discard it, depending on the current top `K` integers. When all worker threads finish, we will have the `K` largest integers appearing in those `N` files in the global linked list in sorted decreasing order. Then the main thread will print them out in sorted decreasing order to the output file, one integer per line.

1.3 Parallel using Multiple Processes

The multi-processing version will be a separate program and invoked as follows:

```
./par_p <K> <directoryPath> <outputFile>
```

In this version, different than the multi-threaded version, each file will be processed by a separate process (created using `fork()`). Each child process as well as the parent process will only have a single (main) thread. Different than threads, since processes cannot communicate simply using their global variables, you will also need to implement an Inter-Process Communication (IPC) mechanism. For IPC, you will use message queues, which will be used to pass information from children to the parent process. The global linked list will be maintained by the parent process, and the children are free to implement their local linked lists first to reduce the IPC cost, similar to the lock contention solution that you used in your multi-threaded version. Since the synchronization of message passing is guaranteed by the kernel, in this version you will not need to use any synchronization mechanism like semaphores. **Check the *System V Message Queues* example and the tutorial link provided in the class website for more information about message queues.** Again, all `N` processes will do the same task concurrently, working on a different file in parallel, similar to the multi-threading case, and the parent process will print the result to the output file.

1.4 Tips and Clarifications

- Start with the sequential one and work incrementally.
- Understand the aforementioned example programs from the class website before implementing the parallel versions. Also, make sure to check their related man pages, such as `man pthreads`, `man ftok`, `man msgsnd`, etc., as well as provided tutorials to understand them better. You will see more links at the end of the class website for tutorials on pthreads.
- Your program will only consider the first level files in the given `directoryPath`, that do not start with a "." and end with a "~" characters.
- You need to make sure that the main thread or the parent process does not terminate before its worker threads/processes are done. Consider joining your threads (multi-threading) or receiving a certain number of messages (multi-processes) from the children before termination. For the thread joins to work properly, make sure your threads are set as joinable using `PTHREAD_CREATE_JOINABLE`.
- The same integer can appear multiple times in the same file or in different files. The output should include top `K` unique integers, eliminating duplicates.
- There will be at least 1 file inside the given `directoryPath`, so $N \geq 1$
- $1 \leq K \leq 1000$
- An input file can contain an integer between and including 1 and $1,000,000,000$.

Please see the black-box testing script for the format of the input/output files as well as example correct outputs for some sample runs.

2 Useful Clean-up Commands

- While testing the multi-processing version, your program might create multiple child processes and terminate unexpectedly due to a segmentation fault. At that point, you might have several zombie processes existing in the system. In order to check current processes in the system including its NAME, you can run the following command:
  - `ps aux | grep NAME`
- If you want to kill a process by its NAME, you can run the following command:
  - `pkill -f NAME`
- If you want to kill a process by its process id (PID), you can run the following command:
  - `kill -9 PID`
- If you want to remove all zombie processes by killing their parents, you can run the following command:
  - `kill -HUP $(ps -A -ostat,ppid | awk '/[zZ]/{print $2}')`
- You can check the currently existing message queues using the following command:
  - `ipcs -q`

  If you want to clean this message queue manually, then run the provided `clean_os.sh` script.

3 Development

You will develop your program in a Unix environment using the C programming language. You can develop your program using a text editor (emacs, vi, gedit etc.) or an Integrated Development Environment available in Linux. *gcc* will be used as the compiler. You will be provided a Makefile and your program should compile without any errors/warnings using this Makefile.

Black-box testing will be applied and your program's output will be compared to the correct output. A simple black-box testing script is provided to you for your own test; make sure that your program produces success messages in the provided test. A more complicated test (possibly more then one test) might be applied to grade your program. Submissions not following the specified rules here will be penalized.

In addition to the success/fail messages, you will also be provided the execution time of your program in milliseconds. There should be an obvious speed-up for the parallel versions, especially for larger input files. The provided black-box test will require you to enter your sudo password to be able to clear the cache before each execution so that the execution time of the parallel and the sequential versions can be calculated accurately.

You will prepare a README.txt file, which will include the performance analysis of your programs. For this analysis, you will calculate and use your speed-up values for the provided black-box test, which can be calculated by dividing the execution time of the sequential version by the parallel version. Make sure you run the provided black-box multiple times and average results for more reliable analysis. Explain if your speed-up values make sense or not. You can run additional tests with different `N` and `K` values as well as input files including more lines to draw further conclusions, if necessary. Do not forget to consider your hardware (number of processors) in your analysis as well.

3.1 Checking Memory Leaks

You will need to make dynamic memory allocation. If you do not deallocate the memory that you allocated previously using **`free()`**, it means that your program has memory leaks. To receive full credit, your program should be memory-leak free. You can use `valgrind` to check the memory-leaks in your program. `valgrind` will output:

"All heap blocks were freed - no leaks are possible"

if your program is memory-leak free. **For the parallel versions only, you will not be penalized for still reachable memory blocks, if you have any!**

4 Submission

Submission will be done through Blackboard strictly following the instructions below. Your work will be penalized 5 points out of 100 if the submission instructions are not followed. In addition, memory leaks will be penalized with a total of 5 points without depending on the amount of the leak. Similarly, compilation warnings will also be penalized with a total of 5 points without depending on the type or the number of warnings. You can check the compilation warnings using the **-Wall** flag of gcc.

4.1 What to Submit

1. `seq.c`: Source code of your sequential version.
2. `par_t.c`: Source code of your multi-threading version.
3. `par_p.c`: Source code of your multi-processing version.
4. `Makefile`: Makefile used to compile your program. If different than the provided one, then you should inform the TA about your changes.
5. `README.txt` including the following information:
    - Names and IDs of the group members.
    - Your performance analysis text as described above. We do not expect less than a paragraph and more than a few paragraphs.
    - Did you take any measures against the lock contention and IPC communication costs mentioned above?
    - Did you fully implement the project as described? If not, what parts are not implemented at all or not implemented as following the specified description? Note that for this project, it is very easy to achieve the required output without using any linked lists, message queues, multi-threading, multi-processing, etc., or it is also very easy to distort your execution times as mentioned above.

Implementing the project without following the specified description or distorting execution times will be considered as plagiarism if not properly described in this README.txt file.
- Structure of your messages. Which size did you use for your messages and why?

4.2 How to Submit

1. Create a directory and name it as the project number combined with your UofL ID number. For example, if a student's ID is 1234567, then the name of the directory will be ProjectX-1234567 for project X. If it is a group project, then use only one of the group member's ID and we will get the other student's ID from the README file.
2. Put all the files to be submitted (only the ones asked in the *What to Submit* section above) inside this directory.
3. Zip the directory. As a result, you will have the file ProjectX-1234567.zip.
4. Upload the file ProjectX-1234567.zip to Blackboard using the "Attach File" option. You do not need to write anything in the "Submission" and "Comments" sections.
5. **Your project will be graded only once!** If you make multiple attempts of submission before the deadline, then only your latest attempt will be graded. If your submission is late, then it will be graded immediately and you won't be allowed to resubmit once your project is graded!

5 Grading

Grading of your program will be done by an automated process. Your programs will also be passed through a copy-checker program which will examine the source codes if they are original or copied. We will also examine your source file(s) manually to check if you followed the specified implementation rules, if any.

6 Changes

- No changes.