Document Version: 1.0 (a newer version number means an update on the project)
Assignment Date: 2/25/2020
Due Date - No Penalty: 3/24/2020, 11:59pm
Due Date - 10 Points Penalty: 3/31/2020, 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 use
- Exercise with multi-threaded programming
- Exercise with the setup and use of message passing as an IPC facility
- Exercise with process/thread synchronization using semaphores

1 Project Description

For this project, you will design and implement a Map-Reduce based WordCount application, which will receive one or more *directoryPaths* as map commands, and will find the number of occurrences of each word existing in all files located in specified directories. You will write two programs: a *mapper* and a *reducer*, which will communicate using message queues. This project will enable you to practice with multiple topics that you have learned so far in this course, including message queues, semaphores, child creation, and multi-threading. **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.**

1.1 Mapper Program

The mapper program will be invoked as follows:

```
./mapper commandFile bufferSize
```

`commandFile` includes the map commands line by line, each line indicating a specific *directoryPath*. A sample content of the command file including three map commands (directory paths) are as follows:

map /home/naltipar/dirOne
map /home/naltipar/dirOne
map /home/naltipar/dirTwo

You can make the following assumptions about the `commandFile`:

1. The *directoryPath* is a full path
2. Each line of the *commandFile* will never exceed MAXLINESIZE characters

Your mapper program will take the map commands from the command file one by one and it will create a separate child process to serve each map command. The directory path of the map command will be passed to the child process. The child process will then create a number of threads to serve the map command. Number of threads to be created by the child process depends on the number of files located inside the *directoryPath*. If there are N files inside the *directoryPath*, then the child process

will create $N + 1$ threads: N *worker* threads and 1 *sender* thread. **If the *directoryPath* includes any sub-directories, you program will skip these sub-directories without considering the files located inside them. So, only the first-level files will be considered.**

Each worker thread will be responsible from a different input file, it will scan the input file and for every word encountered in the input file, it will create a `<map item>` that includes the word itself and an integer 1 as follows:

`<word:1>`

You will use a C struct with the members *word* and *count* to create this map item. Each word in a line will never exceed MAXWORDSIZE characters. Whenever an item is created, then the worker thread will add this item to a memory buffer: a bounded buffer.

The second argument of your program (`bufferSize`) specifies the size of the bounded buffer to be used by the threads of a child process. Each child process will have a dedicated buffer, and each worker thread of a child process will add the produced items into this bounded buffer. The buffer can hold at most `bufferSize` items. This bounded buffer can be implemented in one of two ways: 1) as a linked list of items; or 2) as a circular array of pointers to items. You can choose either one of these implementation options. A worker thread will add a new item to the end of the buffer. The buffer has to be accessed by all worker threads. While trying to insert an item, if a worker thread finds the buffer full, then the thread has to go into sleep (block) until there is space for one more item. Since all worker threads will access the buffer concurrently, the access must be coordinated. Otherwise you might end up with a corrupted buffer. You will use semaphores to protect the buffer and to synchronize the threads so that they can sleep and wake-up when necessary. **Check the *sem-pc.c* example in the class website.**

Beside the worker threads, the sender thread will also access the bounded buffer. The job of the sender thread will be to retrieve the map items from the bounded buffer and send them to the reducer program using message queues. While the items are added to the buffer by the worker threads, the sender thread can work concurrently and try to retrieve the items from the buffer and send them to the reducer program. The sender thread will try to retrieve one item from the bounded buffer at a time. If the buffer is empty, the sender thread has to go into sleep until the buffer has at least one item. When the sender thread is successful in retrieving an item from the buffer, it will send it to the reducer.

The mapper process should be started before the reducer process. The mapper will create a message queue that will be used to send the map items to the reducer. A key will be necessary to create this message queue. Actually, mapper and the reducer should exactly use the same key so that they can refer to the same message queue. Therefore, the parameters used in the **`ftok`** function should be the same for both the mapper and the reducer. However, only the mapper should use the **`IPC_CREAT`** option in **`msgget`** function to create the message queue. Reducer will not need to re-create a message queue, it will only join this message queue that is previously created by the mapper. **Check the *System V Message Queues* examples and the tutorial provided in the class website for more information.**

As an illustration, assume the content of a file to be mapped by a worker thread is as follows:

I can hear you, Katie.
I could always hear you.
Even in the cockpit, with the engines on.

Then, the worker thread responsible from this file will create the following map items (shown as separated by a colon (":") here for clarity) and insert them into the bounded buffer:

I:1
can:1
hear:1
you,:1
Katie.:1
I:1
could:1
always:1
hear:1
you.:1
Even:1
in:1
the:1
cockpit,:1
with:1
the:1
engines:1
on.:1

You do not need to strip the punctuation marks, only the whitespace characters that can either be a <SPACE>, or a <TAB>, or a <NEWLINE>. While the map items are being inserted to the bounded buffer by the worker thread(s), at the same time, the sender thread will extract these items from the bounded buffer and send them to the reducer program one by one, using the message queues.

1.2 Reducer Program

The reducer will be invoked as follows:

```
./reducer outputFile
```

The reducer process that is invoked with the `outputFile` command-line argument will first attach to the mapper's message queue identified by a **key** derived from a file *pathname* and a nonzero integer value. The reducer will use this message queue to receive the map items from the children of the mapper process. Since you are the programmer of both the mapper and the reducer, you will decide this *pathname* and the integer value that the **`ftok`** function expects to uniquely identify a message queue. For instance, you can use the name of the mapper program and the integer `1` in both mapper and reducer to create a key as follows:

```
ftok("mapper.c", 1)
```

See the man page of the **`ftok`** for more details on obtaining a key. You can also check the man page of the **`msgget`** function to learn more about how to get a message queue identifier using the key returned by the **`ftok`** function. Some other useful functions include **`msgget, msgsnd, msgrcv, msgctl`**. **Again, check the *System V Message Queues* example and the tutorial link provided in the class website for more information about message queues.**

Several children of a mapper process will send the map items (received from their respective bounded buffers) to a single reducer process using the same message queue. A special map item needs to be sent to the reducer at the end to indicate the end of mapping process (once every map

command is served). When the reducer receives this special map item (might be a specific keyword as the *word* and -1 as the integer), then it will deallocate the message queue.

The received map items will be aggregated by the reducer to find the total occurrence frequencies of each word. Finally, the words with their total occurrence frequency values will be printed into the `outputFile` in alphabetically (ascending) sorted order dictated by the `strcmp` function. You will use a linked-list for aggregation and sorting purposes. For the example input file presented above, the content of the `outputFile` will be as follows:

Even:1
I:2
Katie.:1
always:1
can:1
cockpit,:1
could:1
engines:1
hear:2
in:1
on.:1
the:2
with:1
you,:1
you.:1

Figure 1 shows the general structure of this project:

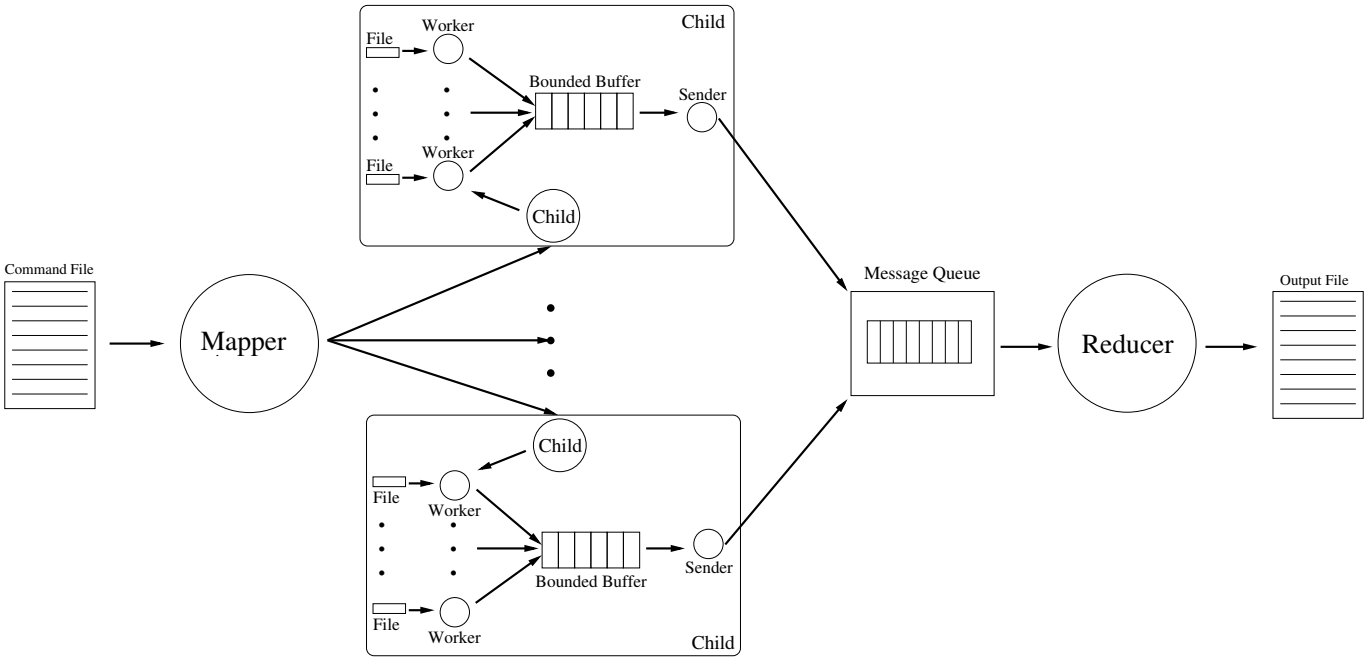

Figure 1: System Structure

Pay attention to the following issues while developing your application:

- In the output file, a word and its frequency should be separated by a colon (":").

- For each line of the commandFile, a specific bounded buffer, a specific sender thread, and bunch of worker threads (one for each file inside the given directory path) are created by the child process that is assigned for that map request.
- Your program will only map the words of the first level files in the given directory path.
- All worker threads for a given map request do the same thing on different files and fill the same bounded buffer.
- Your mapper program passes each received map request to a child process and handles the next map request immediately without waiting for the created child to complete its execution. However, you should also make sure that your program does not terminate before sending all the map items to the reducer. To achieve this, you can `count` the number of children created by the parent mapper and after handling all the requests, the parent mapper can call `wait()` system call in a loop for `count` times.
- Multiple child processes, each responsible from a single map request (a line of the command file), will be sending their map items to the reducer using the same message queue, and the output file created by the reducer will include the aggregated frequency result for all these map requests for a given command file.
- Although it is not required by this project, you can also perform an intermediate aggregation in the sender threads before sending the map items to the reducer. In this case, sender threads will act like intermediate reducers and this approach will speed up your application since the workload of the reducer will be lighter this way.

2 Constants and Tips

- `MAXLINESIZE` will be `1024`
- `MAXWORDSIZE` will be `256`
- Remember to use thread-safe library functions inside your threads! (for instance `strtok_r()` instead of `strtok()`).
- While testing your application, remember to clean the previously created message queues before restarting your application. In order to clean these system resources manually, **run the *clean_os.sh* script provided in the class website.**
- **Some of the examples posted on the class website, especially the ones specifically mentioned above, will be very useful for this project.**

3 Useful Clean-up Commands

- While testing your project, your server 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 the "spks" name in it, you can run the following command:
  - `ps aux | grep spks`
- 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.

4 Development

You will develop your program in a Unix environment using the C programming language. *gcc* must 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 sample black-box testing script will be provided in BlackBoard, make sure that your program produces the success messages in that 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.

5 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. Check `man valgrind` for more details on `valgrind`.

6 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. You can check the compilation warnings using the **-Wall** flag of gcc.

6.1 What to Submit

1. `mapper.c` including the source code of the mapper.
2. `reducer.c` including the source code of the reducer.
3. `README.txt` including the following information:
   - Names and IDs of the group members.
   - Did you fully implement the project as described? If not, which parts are not implemented at all or not implemented as following the specified description? Note that for this project, it is very easy to print out the required output to the screen without using any semaphores, message queues, bounded buffers, forking, or multi-threading. **Implementing the project**

without following the specified description will be considered as plagiarism if not properly mentioned in this README.txt file.

- Structure of your messages. What size did you use for your messages and why?

6.2 How to Submit

1. Create a directory and name it as your UofL ID number. For example, if a student's ID is 1234567, then the name of the directory will be 1234567. If it is a group project, use one of the group member's ID.
2. Put all the files to be submitted (only the ones asked in the *What to Submit* section above) into this directory.
3. Zip the directory. As a result, you will have the file 1234567.zip.
4. Upload the file 1234567.zip to Blackboard using the "Attach File" option. You do not need to write anything to the "Submission" and "Comments" sections. NO LATE SUBMISSIONS AFTER THE PENALTY DEADLINE WILL BE GRADED!
5. You are allowed to make multiple attempts of submission but only the latest attempt submitted before the deadline will be graded.

7 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.

8 Changes

- No changes.