Document Version: 1.0 (a newer version number means an update on the project)
Assignment Date: 3/9/2022, Noon
Due Date - No Penalty: 3/30/2022, 11:59pm
Due Date - 10% Penalty: 4/6/2022, 11:59pm
Due Date - 20% Penalty: 4/13/2022, 11:59pm
No submission is accepted after 4/13/2022, 11:59pm
Group Info: 1 or 2 students; single submission for each group

Objectives:

- Continue writing non-trivial C programs
- Exercise with process creation and use
- Exercise more with thread creation and multi-threaded programming
- Exercise with the creation and use of shared memory as an IPC facility
- Exercise with process/thread synchronization using semaphores
- Exercise with reader/writer locks through the file locking mechanisms of Unix.

1 Project Description

In this project, you will design and implement a client-server based search application where a single server will receive search requests including a `<directoryPath>` and a `<keyword>` from multiple clients, and will search the keyword in all the files located in the specified directory. Client and server will communicate through a shared memory region. Multiple client processes will be able to talk to the server at the same time. This project will enable you to practice multiple topics that you have learned so far in this course including shared memory, semaphores, readers/writers locks, forking child processes, and multi-threaded servers. **For this project, you are allowed to work in groups of at most two 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.** It does not matter which group member made the submission as long as the group members are listed in the provided `README.txt` file.

1.1 Client Program

The client will issue search requests to the server. The name of the client program must be `client` and it will be invoked as follows:

```
./client <req-queue-size> <inputfile>
```

The `<req-queue-size>` parameter specifies the capacity of the request queue that will be used between the client processes and the server process for communication purposes, and `<inputfile>` is the name of the input file including all the search requests to be directed to the server. The client process will get the search requests from the input file line by line (one request at a time) and will send these requests to the server through the request queue. The format of the search requests included in the input file will be as follows:

```
<directoryPath> <keyword>
```

You can make the following assumptions about the `<inputfile>`:

1. The `<directoryPath>` is a full path not exceeding the MAXDIRPATH characters
2. The `<keyword>` is a single word not exceeding the MAXKEYWORD characters, enclosed by whitespace characters not including any whitespace characters inside. Note that a whitespace character can be either a <SPACE>, or a <TAB>, or a <NEWLINE> character.

A sample input file content can be as follows:

/home/naltipar/dirOne is
/home/naltipar/dirOne a
/home/naltipar/dirTwo a

When the client process reads a search request from the `<inputfile>`, a request message will be generated and sent to the server using the request queue implemented on the **shared memory**. Several client processes can send requests to the server using the same request queue. A special command will be used to terminate the server. When the server receives this special search command, then it will terminate and will not be able to serve any further requests. This special command will only include "exit" in the message as follows:

```
exit
```

1.2 Server Program

The server will accept search requests from several clients and for each request, it will search the `<keyword>` in all files residing in the given `<directoryPath>` using multiple threads. The server program must be named `server` and it will be invoked as follows:

```
./server <req-queue-size> <buffersize>
```

As in the client case, `<req-queue-size>` specifies the capacity of the request queue to be used between the server and the client processes. The same `<req-queue-size>` value should be passed to the server and the client for proper settings. `<buffersize>` specifies the size of the bounded buffer to be used between the threads inside the server (more details below).

The server process should be started before the client processes. When started, the server will create a shared memory segment for IPC and some semaphores for mutual exclusion and synchronization purposes of the shared memory. The shared memory segment and the semaphores will be initialized by the server. Shared memory should be created with a name (as specified by the man page for `shm_open()`) that you will select and define in your program. The name will be given as a parameter to the `shm_open()` function that will be used to create a shared memory region. The clients will use the same name to access the shared memory. Since you are the programmer of both the client and the server, you will decide this name. **Similar situation applies to the named semaphores that will be created and initialized by the server as well.** You will decide which semaphores and how many semaphores you will use and how you initialize them. It depends on how you design your synchronization and critical section solutions. You will use POSIX semaphores.

The server will do some initialization on the shared memory segment. As part of this initialization, it can for instance create and initialize a shared array inside the shared memory segment. This array can be used as the request queue and it can hold `<req-queue-size>` number of requests. Request queue can be accessed by all client processes and the server process. While trying to insert an item, if a client process finds the request queue full, then it should go into sleep (block) until there is space for one more request. Similarly, if there is no request in the queue, server process should go into sleep. Besides, since many processes access the request queue concurrently, the access must be coordinated. Otherwise, race condition can happen. As you see, this is the common producer-consumer problem that we discussed in class, so the provided *sem-pc.c* code in the class website will be very useful. **One difference you should pay attention is that you will need to use named semaphores instead of regular semaphores to protect the request queue and synchronize the actions of different processes since named semaphores can be shared among processes (clients and server).** On the other hand, regular semaphores are used to achieve synchronization within a process, such as synchronizing the actions of the threads of a process.

The server will perform the search requests received from the request queue as follows. First, it will immediately create a new child process to serve each request and the search information will be passed to this child process. The child process will then create a number of threads to serve the search request. The 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 *printer* thread. Each worker thread will be responsible for a different search file, it will scan the search file and look for the matching lines based on the `<keyword>`. **Your program will only search the first level files in the given `<directoryPath>`. If the `<directoryPath>` includes any sub-directories, you should skip them without searching the files in these sub-directories.**

A match in a line will be an **exact match** of an **entire word only** and the maximum length of a line in a search file will never exceed MAXLINESIZE. When a *worker* thread finds a match, it will prepare an *item* that includes the following information: the name of the file where the match has occurred, the line number of the match, and the line itself (i.e. the line string excluding the newline character at the end). If the keyword is seen, for example, in 5 separate lines, then 5 separate items will be created. If the keyword is seen more than once in the same line, then a single item will be created for that line. Whenever an item is created, then the worker thread will add this item to a memory buffer: a bounded buffer.

The second argument of the server (`<buffersize>`) specifies the size of the bounded buffer to be used by the threads. Each worker thread will add the produced items to 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 we might end up with a corrupted buffer. **Again, this is the producer-consumer problem, but the difference here is that you will use regular semaphores (not named semaphores) to protect the buffer and to synchronize the threads so that they can sleep and wake-up when necessary without corrupting the buffer.** Again, the provided *sem-pc.c* example will be very useful, which directly uses regular semaphores. **Please also note that reusing any code that we provided will not cause any plagiarism issue!**

Beside the worker threads, *printer* thread will also access the bounded buffer. The job of the *printer* thread will be to retrieve the items from the bounded buffer and print them into an output file. While the items are added to the buffer by the *worker* threads, the *printer* thread can work concurrently and try to retrieve the items from the buffer and print them to the output file. The *printer* thread will try to retrieve one item from the bounded buffer at a time. If the buffer is empty, the printer thread has to go into sleep until the buffer has at least one item. When the *printer* thread is successful in retrieving an item from the buffer, it will print it to the output file in the following format:

filename:linenumber:linestring

For example, if the keyword is "*dream*" and the content of the file to be searched (assume the file name is *poe.txt*) is as follows:

Take this kiss upon the brow!
And, in parting from you now,
Thus much let me avow
You are not wrong, who deem
That my days have been a dream;
Yet if hope has flown away
In a night, or in a day,
In a vision, or in none,
Is it therefore the less gone?
All that we see or seem
Is but a dream within a dream.

I stand amid the roar
Of a surf-tormented shore,
And I hold within my hand
Grains of the golden sand
How few! yet how they creep
Through my fingers to the deep,
While I weep while I weep!
O God! Can I not grasp
Them with a tighter clasp?
O God! can I not save
One from the pitiless wave?
Is all that we see or seem
But a dream within a dream?

Then, the *printer* thread will print the following to the output file:

poe.txt:11:Is but a dream within a dream.
poe.txt:25:But a dream within a dream?

The output file will be named as `output.txt` and all child processes (only the *printer* thread from each child process) will write their output into this single output file. Since the output file can be modified by multiple processes concurrently, you should also synchronize the access to this output file. For this purpose, you will use the `fcntl` system call available in Unix providing file locking mechanism. Note that advisory locking should be sufficient for your case and you can assume that the maximum length of an output line will never exceed MAXOUTSIZE. You do not need to worry about the ordering of the lines in the output file; before testing your output, we will sort the output file ourselves. **Check the *file_locking_solution.c* example and the tutorial provided in the class website for more details on file locking.**

Figure 1 plots the general structure of this application:

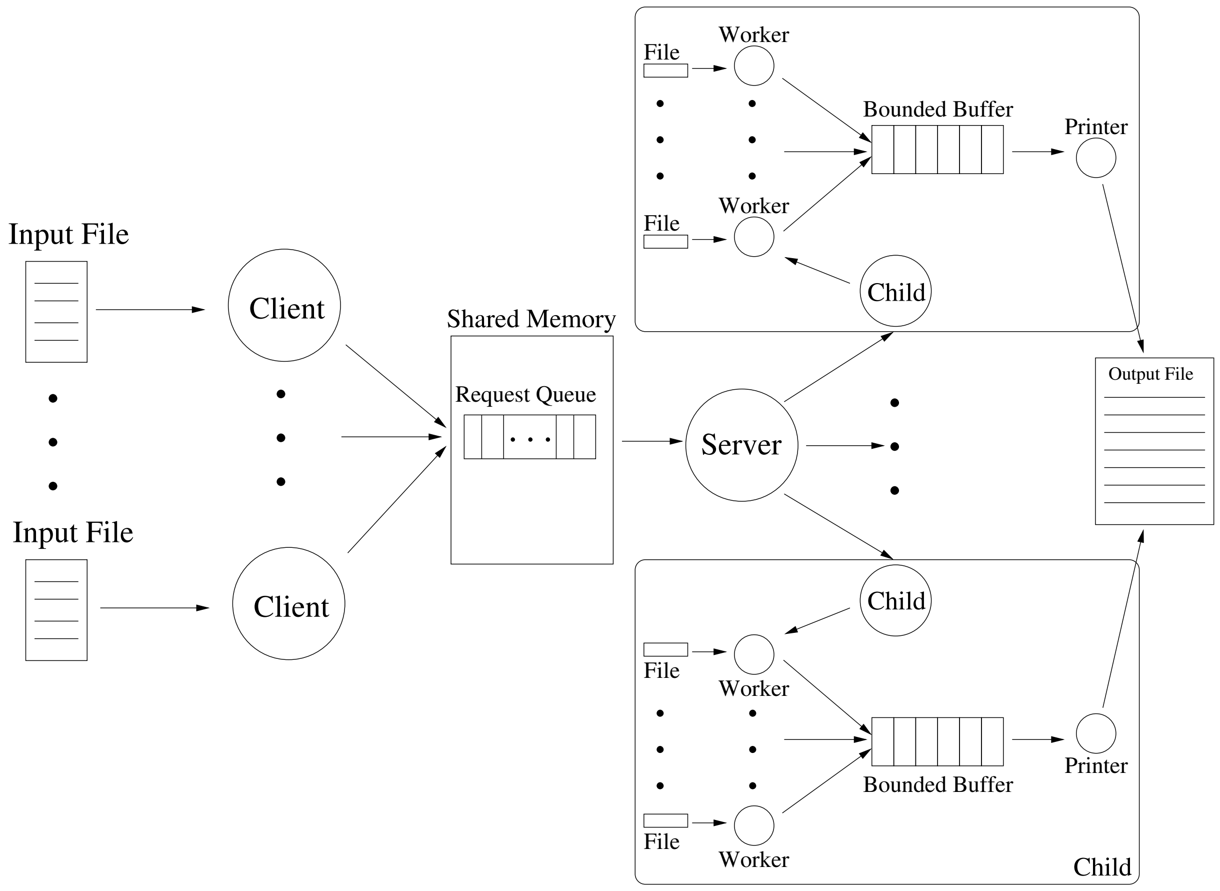

Figure 1: System Structure

Pay attention to the following issues while developing your application:

- The output should not contain the name of the directory, just the name of the file.
- Only the lines that include the exact match of an entire word is printed (use `strcmp()`).
- If the keyword is seen more than once in a line, then a single output will be printed for that line. For example, if we had a line as "a a a", then the number of matches on this line would be 3 for the keyword "a". However, we have to report such a line only once.
- For each search request received from the request queue, a specific bounded buffer, a specific printer thread, and bunch of worker threads (one for each file in the given `<directoryPath>`) are created by the child process that is assigned to that search request.
- Your program will only search the first level files in the given `<directoryPath>`.
- All the worker threads for a given search request do the same thing on different files and fill the same bounded buffer.
- **Your program passes each received search request to a child process and handle the next**

search request IMMEDIATELY without waiting for the created child to complete its execution. This is very important to achieve a responsive server design! However, you should also make sure that your program does not terminate before printing all the outputs of the already received requests. To achieve this, you can `count` the number of children created by the parent and after handling all the requests (`exit` request is received), then the parent process can call `wait()` system call in a loop for `count` times. We will make sure that the `exit` request is sent as the last request in the queue.

- You will also need to make sure that the created child process in the server does not terminate before its worker and printer threads are done. For this purpose, first you need to join all your worker threads, then insert a dummy item to the bounded buffer to indicate the end of search, and then join your printer thread. A dummy item can be one with a negative line number, and once the printer thread receives this dummy item, it can terminate. For the thread joins to work properly, make sure your threads are set as joinable using `PTHREAD_CREATE_JOINABLE`.

2 The Shared Memory Segment

All data between the clients and the server will be passed through the shared memory. You will use POSIX shared memory API. Some of the useful functions that are part of the POSIX shared memory API are the following: `shm_open()`, `mmap()`, `shm_unlink()`, `ftruncate()`, `fstat()`. The size of the created shared memory should be determined by you considering the content to be kept in the shared memory (request queue, any variable to be stored in the shared memory for indexing the request queue if necessary by your design, etc.). **Check the *POSIX Shared Memory* examples provided in the class website.**

3 Constants and Tips

- `MAXDIRPATH` will be `1024`
- `MAXKEYWORD` will be `256`
- `MAXLINESIZE` will be `1024`
- `MAXOUTSIZE` will be `2048`
- Remember to use thread-safe library functions inside your threads! (for instance `strtok_r()` instead of `strtok()`).
- Named POSIX semaphores can be shared among processes (clients and server).
- **While testing your application, remember to clean the previously opened shared memory and semaphores before restarting your application. If your application does not terminate properly, these regions can be left open without being unlinked, causing inconsistent results. In order to clean these regions manually, check the *clean_os.sh* script provided in the class website.**
- Many of the source code examples posted on the class website and mentioned in the lectures will be very useful for this project.

4 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 "server" name in it, you can run the following command:
  - `ps aux | grep server`
- 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}')`
- Shared memory regions or named semaphores you created may stay in the system if your server has terminated unexpectedly without unlinking these objects. If you do not remove these objects and re-run your program, then these regions will be reused and unexpected results will be produced. Therefore, you should clean these objects before re-running your programs. You can run the following code to see the existing shared memory/semaphore objects in the system:
  - `ls -al /run/shm` or
  - `ls -al /dev/shm`
- If you want to clean the existing shared memory/semaphore objects, then run the provided `clean_os.sh` script.

5 Development and Testing

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.

6 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`. **For this project only, you will not be penalized for *still reachable* memory blocks, if you have any!**

7 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.

7.1 What to Submit

1. `client.c` including the source code of the client.
2. `server.c` including the source code of the server.
3. `README.txt` including the following information:
   - Names and IDs of the group members.
   - Did you fully implement the project as described? If not, what parts are not implemented at all or not implemented by following the specified implementation rules? **Please note that for this project, it is very easy to print out the required output to the screen without using any shared memory, bounded buffers, multi-threading, or file locking mechanisms. Implementing the project without following the specified description will be considered as plagiarism if not properly documented in this README.txt file.**
   - Structure of your shared memory region. What size did you use and why? What information did you store in the shared memory, in which order?

7.2 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!

8 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.

9 Changes

- No changes.