CS342 Operating Systems – Spring 2018
Project 2: Multi-threaded Server, Shared Memory and Synchronization

Assigned: March 14, 2018
Due date: April 4, 2018, **23:55** (Moodle) (no deadline extension will be granted)

You can do this project as a group of two students each. You have to program in C on Linux. You are required to use the following distribution of Linux: Ubuntu 16.04 – 64 bit. Because this will be our test environment.

Objectives: Practice developing multiple-threaded applications, practice inter-process communication (IPC) via shared memory, practice solving synchronization problems, and practice use of semaphores.

In this project you will implement a multi-threaded server that will process keyword search queries from clients. You will implement both the server program and the client program. Multiple clients can request service from the server concurrently. The service given will be a keyword search service. Upon receiving a keyword from a client, the server will search the keyword in an input text file and will send back the line numbers of the lines having the keyword at least once (i.e., line numbers of matching lines). Inter-process communication between clients and server will be achieved via shared memory. For each client, the server will create a thread to handle the request of the client. The client program will just send one request and will wait for the results to arrive. Then it will write the results to the screen and will terminate. Several client processes can be started concurrently to run in the background using the & sign, or by use of a different Linux terminal window for each different client.

The server program will be called "server" and will take the following parameters.

server <shm_name> <inputfilename> <sem_name>

<shm_name> is the name of the shared memory segment that will be created by the server and that will be used by the clients and server to share data and communicate with each other. The name of the shared segment is just a filename (for example /mysharedsegment). <inputfilename> is the name of an input text file (ascii) which will be searched upon receiving a request (query) from a client. <sem_name> is a *prefix* for naming your semaphores.

When started, server program will first create a shared memory segment of enough size and will initialize it. The segment will contain shared data. The shared data will include a request queue (to send requests from clients to server), several result (data) queues (to send results from worker threads of the server to the respective clients) and some other data structures or variables necessary to solve this problem.

There can be at most N clients running concurrently. Therefore, there will be N result queues in the shared segment: one queue per possible client. Each active

client will use a separate result queue. An additional data structure in shared memory can be used to keep information about the state of the queues. This structure can be, for example, an array of N elements, where each element may indicate if the corresponding result queue is used or not. The element type can be integer. Let use call this array as queue_state. Initially it will be all zeros (no queue is used yet).

The client program will be a single threaded program and will take the following parameters.

client <shm_name> <keyword> <sem_name>

<shm_name> is the name of the shared memory segment created by the server. Client will use it to access shared data and to communicate with the server (send a request and get the results). <keyword> is an alphanumeric string (including just uppercase and lowercase letters and digits) that will be searched at the server side in the input file that the server has access to. A client can not search the file directly; the server process will search it. When started, the client will first open the shared memory segment and will get ready to access it. Then it may access the queue_state array in the shared memory and try to find a queue that is unused at the moment. Make sure you lock (by use of a semaphore) queue_state array while you are searching and modifying it. If there is no unused queue at the moment, the client will terminate after printing an error message like "too many clients started". This will happen if there are already N clients started and running concurrently. Server can handle at most N clients concurrently. If the client can find an unused result queue while searching the queue_state array, the client will mark the corresponding entry in the queue_state array as "used" and will record the index of that entry. This is the index of the queue that is unused. It will be used by the client to retrieve the results. The client needs to send this index information to the server as well. You can assume result queues have indices like 0, 1, …N-1.

Then, the client will generate a request (a structure) that will include two fields: the keyword taken from the command line and the index of the selected unused queue. The client will put the request into the request queue in the shared memory. This is like putting an item into a bounded buffer. This queue is a circular bounded buffer. The size of the request queue can be N or more (at most N requests should be pending, since at most N clients can be created simultaneously). In this way, the client can send (pass) the request to the server through the request queue in the shared memory. Then the client will wait for the results to arrive at the corresponding result queue.

The server program will wait for requests to arrive from clients. When a request from a client arrives (is found in the request queue), the request will be retrieved from the request queue and a new thread (worker thread) will be created to handle the request. The request is passed to the new thread. Then, the server will retrieve another request from the request queue if there is one in the queue, or it will wait for a request to arrive if there is none. A new thread that is created to handle a

request will first parse the request (extract the keyword and the index of the result queue). Then the thread will open the input file and will read and search the input file for the keyword. Whenever a matching line is found (a line that has the keyword), the corresponding line number will be inserted to the result queue. As said, the index of the result queue is found from the request. The result queue is also a circular bounded buffer. Its size is BUFSIZE. That means it can hold BUFSIZE integers at most. Be careful about synchronization while your are inserting integers into the queue. After finishing searching the file, the worker thread may send one more integer (-1, for example) to indicate the end of the integer stream passed to the client. Then the worker thread will terminate.

The client that was waiting for results (integers) to arrive, will start retrieving and printing the integers to the screen while the integers are arriving. There will be one integer per line printed. Then the client will terminate, after some clean up.

Clients and server will use several semaphores to solve this problem. All these semaphores will be given names. All names must start with the prefix <sem_name> that you specify while starting the server and client. Then the suffix is up to you. You should make sure that a semaphore that you use in the server and in the client for a specific purpose is accessed with the same suffix. For example, as a mutex semaphore to access a result queue with index 3, a student with name Ali Can can use a semaphore named "/alican_semaphore_resultqueue_3". That means the prefix can be "/alican_semaphore", and the suffix can be "_resultqueue_3". In this example, you need to start the server and client with <sem_name> being "/alican_semaphore".

You will design the layout of the shared memory. You can put variables to the beginning, to the end, or any other place you want in the shared segment. You can use offsets for variables or structures and use them in the server and client. You can use structures and pointers to structures to access shared memory content easily. For example, if `p` is a pointer pointing to the beginning of the shared memory and `my_struct` is a structure containing some fields, then you can do the following.

```
void *p;

structure my_struct {
      int field1;
      int field2;
      int filed3;
};
…
p = mmap(…);

struct my_struct  *sp;

sp = (struct my_struct *) p;
sp->field1 = 3;
sp->field2 = 7;
sp->field3 = 13;
```

The code above is using a pointer to a structure to update the fields of the structure. Since the structure pointer is made to point to the beginning of the shared memory, this code is in fact updating the shared memory. In this way, the structure (content) is on shared memory.

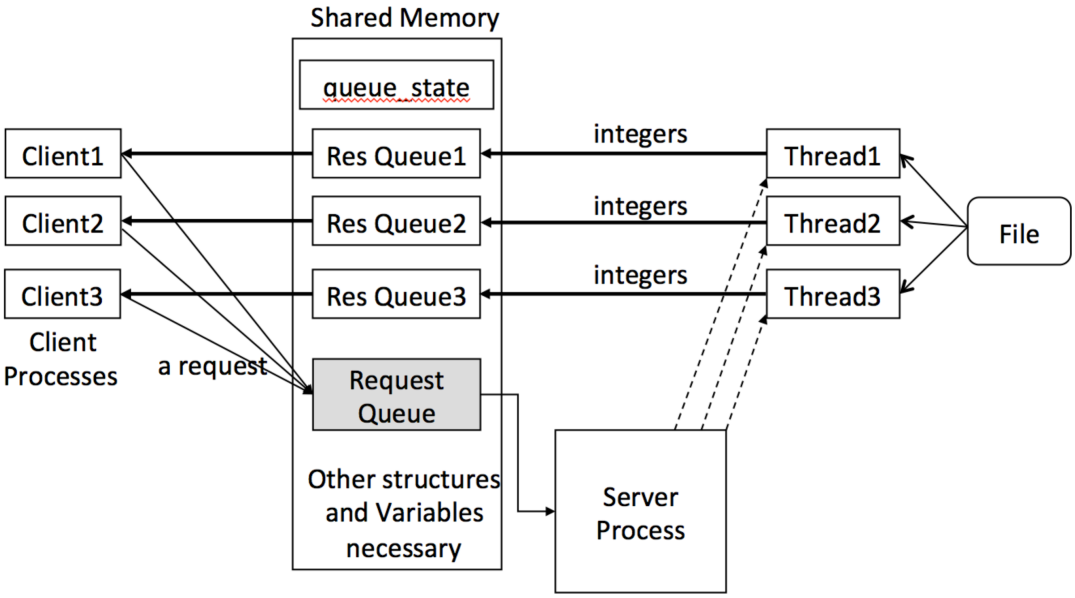

Figure shows the runtime structure of the application when there are 3 concurrent clients. There are 3 threads created. Each thread is handling a different client. Each thread accesses the input file and retrieves and checks the lines whether they contain the keyword. Line numbers of the matching lines are sent to the client using the respective result queue. The figure shows a portion of the data that should be in shared memory. The layout above is just an incomplete example. The design of the shared memory layout (which variables and structures will be in shared memory and where in shared memory) is up to you. You can implement a queue as an array with in and out index variables. As you know, the in an out index variables for a queue (buffer) has to be in shared memory as well.

Having the keyword somewhere in a line is enough as a match. For example, keyword “Apple”, is matching the following lines. Case sensitive matching will be used.
“ThisApple is”
“As a fruit, Apple has”
“a lot of Apples”.

You will use the following constants.

- N (number of concurrent clients) is 10. This is the maximum number of clients running concurrently and accessing the server.
- Max line in the input file is 1024 characters long including the newline character at the end.
- Max filename is 128 characters long (including NULL character at the end).

- Max keyword is 128 characters long (including the NULL character at the end).
- Max shared memory segment name is 128 characters long (including the NULL character at the end).
- Max semaphore name prefix is 128 characters long (including the NULL character at the end)
- BUFSIZE is 100. That means a result queue can store 100 integers (items) at most.
- Line numbering for the input file starts with 1.

You will use POSIX threads. You can get overview information about POSIX threads using the command “man pthreads”. You will use POSIX semaphores. You can get overview information about POSIX semaphores using the command “man sem_overview”. You will use POSIX shared memory. You can get overview information about POSIX shared memory using the command “man shm_overview”. In POSIX shared memory API, there are functions like “shm_open”, “ftruncate”, “mmap”. There is a lot of information on web as well.

Report. Design and do some timing experiments and write a report about the results.

Submission. You will submit via Moodle. Put everything into a folder (as usual) and do you submission as in project 1. Include a Makefile. Include a README.txt file (containing the names of students in the group). One submission per group is enough. Late submissions are not accepted. No deadline extension will be provided.

Clarifications:
- Work incrementally, go step by step.
- Solve the problem first for one client.
- Use of printfs can be a good method for debugging of multi-threaded problems.
- Semaphores have persistence beyond process lifetime. When a process terminates, a semaphore is not removed automatically. For cleaning up while a client is terminating, you may need to remove some of the semaphores. Some other semaphores may need to be staying even though the client terminates.