# CSL303 | Assignment-1 |

### 

### Submitter name:

## Mohit Singhal

### Roll No.:

## 2018eeb1165

### Course:

## CS303

================================================================================================================================================

## What does this program do?

This program mimics the dispatcher of an operating system manages processes. For interprocess communication, it employs the client-server model. After establishing a connection, the client will submit a request to the server. This request was queued by the server. Each request is then dequeued and processed, this process is repeated by multiple threads in parallel. When a client request is pushed, it receives an acknowledgement and a result based on the completion of that request.

## The logic behind the code

In the main function, I first initialize all the variables which I need like mutexes, conditional variables and semaphores. Using the thread limit provided by the user than I create a thread pool. Then the server will keep on listening for the client requests indefinitely. Each time a request is made to the server, the request is pushed to a queue. This request is then popped by some free thread of the thread pool and the request is then executed.

Conditional variables are used to limit the CPU utilization.

Semaphores are used to implement file limit.

Mutexes are used to prevent race conditions while accessing the dispatcher queue.

## How to compile and run the program?

To compile the code for the server which mimics a basic dispatcher run the following in the shell:-

> gcc server.c -o server -pthread -ldl

pthread and ldl are linked dynamically. Now run the following in the shell to run the server and provide the necessary command-line arguments:-

> ./server [Thread Limit] [Memory Limit (in MB)] [File Limit] [Port number]

Thread limit:- number of threads in the thread pool of the dispatcher

Memory limit:- Memory Limit argument here is the total limit of the virtual memory that the process and its child process can use. The limit should be at least 70-80 MB otherwise the process may fail to create all the threads required for the thread pool.

File limit:- The total number of files the dispatcher can open at a time.

Port number:- the Port number on which your local host will listen. If you face a bind error either try running the server with a different port number or kill the current process which is running on this port.

Note:- Imposing a very strict thread and file limit will make the dispatcher slow and the socket listener queue might not hold so many pending requests in its queue. This would force the server to drop some client requests. This would happen in very extreme situations where hundreds of clients are sending the request simultaneously and the resource limits of the server are very few.

## Sample Run

To test the program I have created various test cases to test the functions of the server.

The test.c file contains the test cases. Compile it and run all the tests contained in it.

> gcc test.c -o test -ldl

Assuming the server is running, execute the test:-

> ./test

To check if the server and the dispatcher implementation is actually multithreaded, compile and run the client.c file. This will create multiple concurrent clients and send multiple concurrent connection requests to the server and some of them might be parallel as well as I did the testing on a multi-core system.

If the memory limit is > 100MB and the thread and file limits are sufficient, the server would always be able to serve all the requests.

Depending on the resource limit on which the server is running, some of the requests of the client might be dropped. This will only happen if the resource limits are too less.

To compile the client.c :-

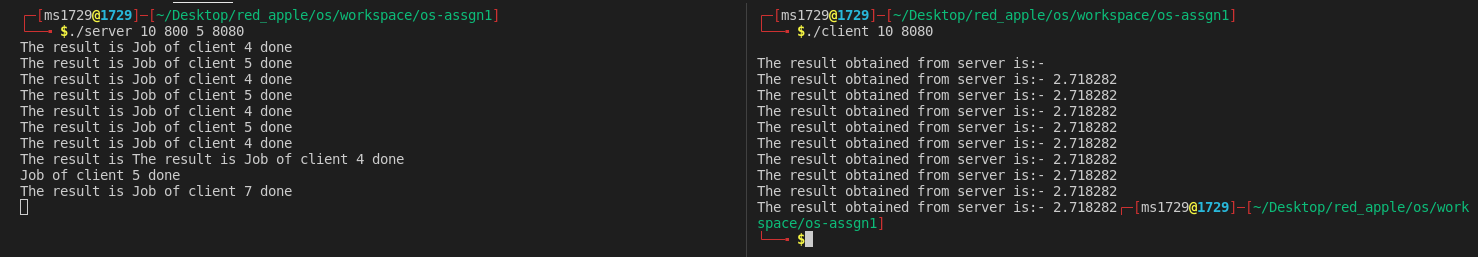
> gcc client.c -o client -pthread

Assuming the server is running the client using:-

> ./client [number of threads] [port number]

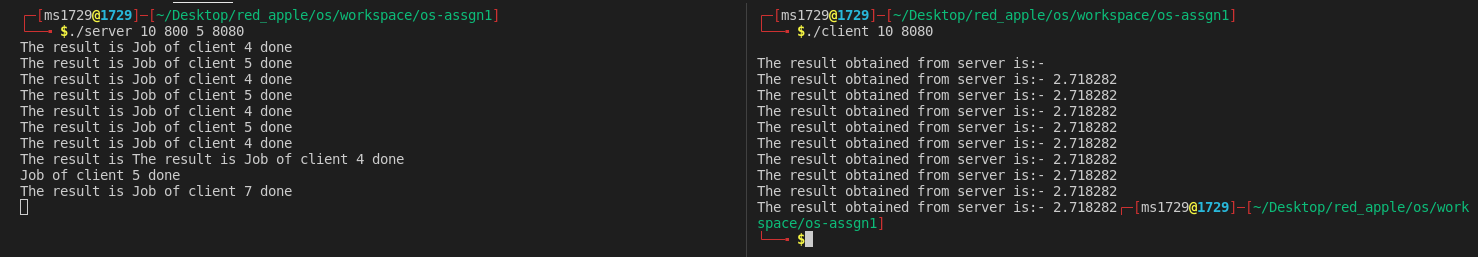
Make sure the port number of the server and client match otherwise you will get a connection failed message.

Running the server with sufficient resources limits and testing it with 10 concurrent clients. All of them call an exp function from the libm.so.6 library. We can see the sockt\_fd assigned to these 2 clients are either 5 or 4. This is because the server is multithreaded and the server can accept connections from more than one request at a time and hence two different socket\_fd. Why only 2 and not more? Because the function executed by the threads are very light and therefore the execution is very fast. Before the server could accept more requests, the requests of the previous clients are already served.



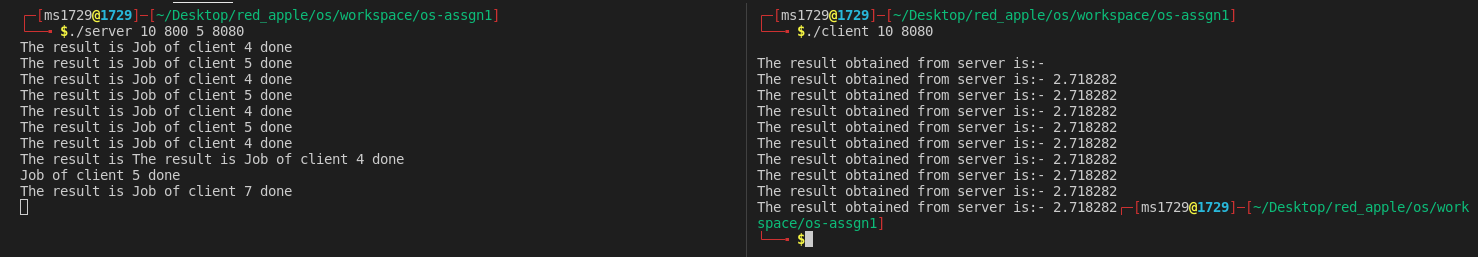
If I insert sleep(5) in the execution of the code and keeping the resource lmiit of the server

same, this time the socket\_fd for each connection will be different.



Also, note that if the execution of the task of client 1 begins earlier than the execution of client 2, it’s not guaranteed that the execution of client 1 will get completed earlier than the execution of the client2.

By increasing the thread limit and file limit, the server can serve more clients concurrently. In the following case, only 99 threads are created because of the limit of the memory but still, the server is able to serve so many clients concurrently and no request is getting dropped.



## References

1. [RLIMIT documentation](https://man7.org/linux/man-pages/man2/getrlimit.2.html)
2. [Ulimit](https://www.geeksforgeeks.org/ulimit-soft-limits-and-hard-limits-in-linux/)
3. [Thread Attributes](https://users.cs.cf.ac.uk/Dave.Marshall/C/node30.html)
4. [Usage of rlimit](https://www.geeksforgeeks.org/get-set-process-resource-limits-in-c/)