CS342 –Project 2

Report

Arif Can Terzioğlu

21302061

Environment

My main system works on Windows 10 OS. On the hardware side, computer processer is 4 cores Intel i7 4700HQ 2.Ghz series. Ram amount is 16GB.
I installed Ubuntu 16.04 64bit via virtual machine. I allocated two core of my computer processor. 5000 megabyte is allocated for main memory.

Design Stage

I started to project by gaining well understanding on project assignment. At the beginning it seems really hard and long since it requires a lot of concepts. During the resolving stage of the project, I mainly used supplementary book of our course, course book and YouTube. As I was progressing, I realized that project is strongly connected to classic problems of synchronization such as bounded buffer and readers-writers problem. Then I note down the required shared memory parts. Basically, from the assignment I noted down request queue, several result queue, state queue and some other data structures for indexing. I made them struct array instead of multiple arrays. Because indexing of multiple result queues was hard task since it is hard to distinguish them. I was given two option one that keep them in a multi-dimensional array or another way. Then, I decided keep each result queue in a struct. Each struct contains size of 100 integer array and in and out indexes. By doing this design, I was able to access each result queue during the coding stage. I also applied this principle to request to hold both queue index and keyword easily.

I determined my prefixes for both semaphores and shared memory. I decided to create all semaphores inside the server and open them in every client. During the decision stage of which semaphore should be created, one semaphore for state queue to ensure mutual exclusion between clients, three semaphores for request queue one that is similar to bounded buffer problem on page 269 of our course book (One for mutual exclusion, one for full array and one for empty array semaphore.) and in a same way three semaphores for each result queue. Since reading the file does not violate or change any data, there is no need to use semaphores for file. In other words, there is no writer on file.

Usages of semaphores and pthreads have been learnt from supplementary book and YouTube. Reading the file and iterate over the lines are learnt from my good friend “stackoverflow”.

Implementation Stage

Throughout the implementation, I tried to stick with working incrementally principle. All challenges will be explained in next section of report.

On the server side, I first created shared memory and I ensured that data transaction is successful. I used struct method a lot due to reasons I mentioned in the design stage. Then, I write the codes of reading file and I gave words from the console then print the line numbers. I applied it with one thread.

After I ensured that runner function of thread doing well, I have written a dummy client file which basically scan the queue array and finds the 0 index available in a dummy client and sends the request that is index and keyword. By using printf I tested the weather request arrives to server or not. When I successfully print them, I tried to send them to runner function and real challenges began here. Runner functions first printed the line values on server console. Then I tried to write them to result queue and print on the client side.

After I successfully completed the one client case, to be able to handle multiple clients' implementation, I created and placed the necessary semaphores. Semaphores were put in a place that their critical section is a data modification. I created all semaphores on the server side, open them on the client side to use them. State queue semaphore is only used in clients.

For thread handling, I created array of pthead_t size of 10. I assigned each of them to one result queue. For example, result queue on the index 3, uses thread on the index of 3.

Since server is always running, there is no need to close any structure.

Result queue semaphores were defined globally to make them accessible to threads.

In final state of both server and client works in follows:

Server

1. Create shared memory.
2. Create semaphores.
3. Wait for a request.
4. If request arrives, parse it and prepare it as struct parameter for runner function.
5. Create a thread with an unused pthread_t at that moment.
6. Main program goes back to looking for a new request.
7. Runner function scans the file and sends the line number to given result queue.

Client

1. Open shared memory.
2. Open semaphores.
3. Looking for available state queue in other words result queue unused.
4. If there is no result queue available, terminate by giving error.
5. Send request to server.
6. Wait for line numbers to arrive
7. Print them one by one as they arrive.
8. Make unused current state queue.
9. Close semaphores (Not delete).
10. Terminate.

Challenges during Implementation

One of the biggest challenges for me was passing the parameters through functions and between clients and server. I realized that I forgot about pointer and pass-by-reference concept. I opened CS201 slides and repeated the some concepts.

As it can be guessed, I suffered a lot from segmentation fold error. These errors occurred at the creation of semaphores. One that I have written the unlink command just after creating the semaphores. It was a small but time consuming mistake. Also, when I was trying to open created semaphores in client, I always got a segmentation fold. I learnt that these errors occurred due to fact that I did not give O_READ authorization to that process. In other word, I was not putting O_READ parameters while trying to open the semaphores (not creation, opening).

Another interesting problem was that when I tried work with multiple clients on the console, each client was waiting for prior clients to begin to print line numbers. I realized that it emerged from same pthread_t. This is the reason why I use array of phtread_t of size 10.

Testing

During testing stage I used ./client …. & ./client … format to understand whether max 10 clients constraint works or not. When 11th client wants to access, program does not give permission successfully.

Also, I tried to work on a 12 different console one with server and others were client. In the server code, I used sleep function so that I will be able to run all clients before one of them terminates. 11th client gave a too many clients error successfully. After completion of one client, It starts successfully also.

Methodology

Each time is measured three times and their average values are taken. After the graphics, results and conclusion are explained.

Note: For a consistent result, I used the same keyword for searching. Number of line count is increasing in every number of clients.

Inputs:

Client Count: 1 5 10

Line Count: 100 1000 10000

Results

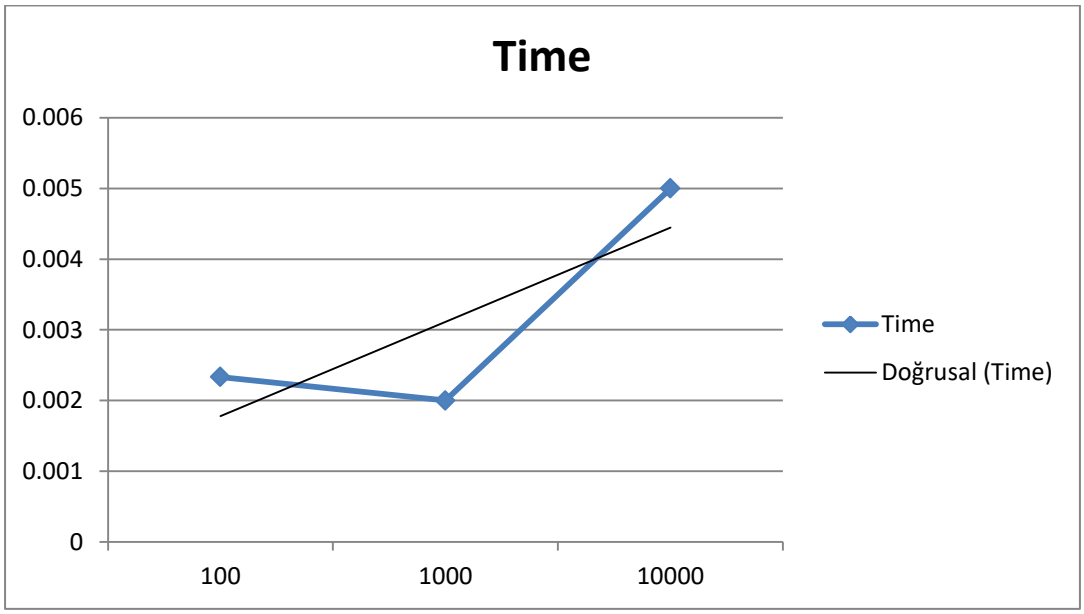
According to values, there were no surprising results for each number of clients. As expected increasing number of clients and line counts increases the time.

Assumption: Scanning whole file is working in θ (n).

Observation: I used the time function of Linux kernel for each experiment. This command can show the duration on both user side and system side. According to time values, as the number of threads using is increasing, time passing on system side will also increasing.

Below graphics only shows the total time elapsed on both system and user side.

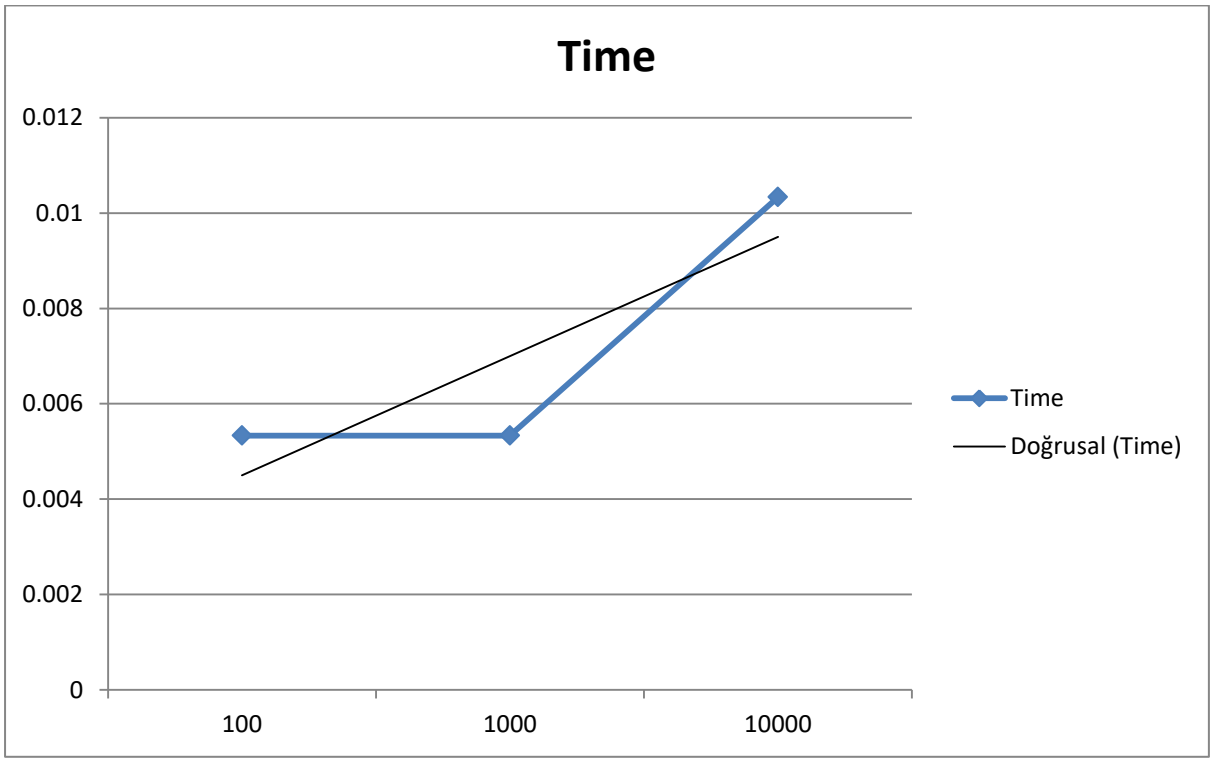
N=1



	1	2	3	Average
100	0.002	0.003	0.002	0.002333
1000	0.002	0.002	0.002	0.002
10000	0.004	0.007	0.004	0.005

While client size is 1, there was no huge difference between results. Since number of threads using is limited, time elapsed is not much.

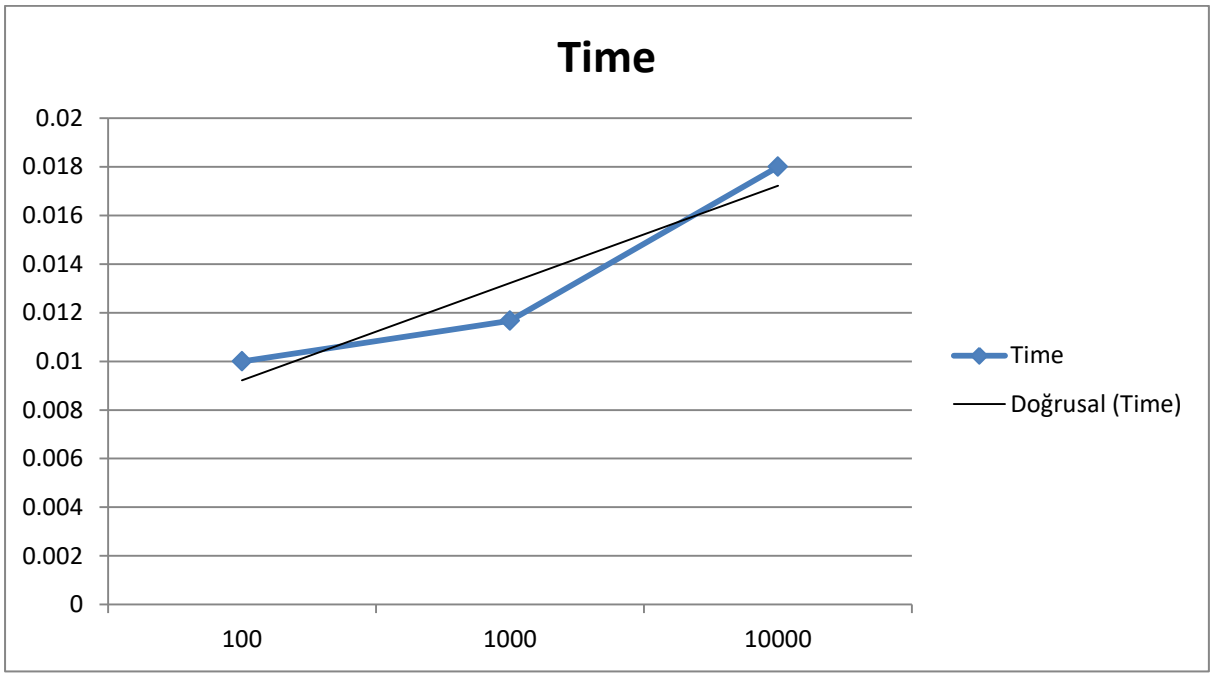
N=5



	1	2	3	Average
100	0.004	0.007	0.005	0.005333
1000	0.005	0.005	0.006	0.005333
10000	0.01	0.011	0.01	0.010333

Overall time is increased compared to 1 client since it requires more system time. Naturally, number of increasing line also increases the total time.

N=10



	1	2	3	Average
100	0.011	0.009	0.01	0.01
1000	0.013	0.012	0.01	0.011667
10000	0.024	0.013	0.017	0.018

It can be observed from the this part that 10 clients increases total system time elapsed a lot compared to other client counts from previous parts.

Note

This experiment could be done in a way that all client commands starts sequentially. In this experiment, all commands have started concurrently since this methods give more realistic results such as all users can try to access at the same time and etc.