GTU 2021-2022
CSE344 - System Programming

Midterm Project - Report

Ahmet Tuğkan Ayhan
1901042692

In this report, I will describe how I solved the given problem and my design decisions for this problem. Before I submitted my midterm project, I made sure everything was working as it should.

What is the Problem?

The problem was creating a client-server program. In this problem, while there could be any amount of clients, there could be only one server running at the same time. Requests sent by an indefinite number of clients should be handled by the server and the response should be returned to the client. Since the problem contains many details with this simple definition, instead of explaining them, I will proceed by explaining my design decisions.

clientX

Controlling Command Line Arguments

I started implementing the client by checking the command line arguments. If the given arguments are wrong, I informed the user and exited the program.

```
tgknyhn@tugkan-ubuntu:~/Desktop/midterm$ ./client "/tmp/ServerFIFO" -o "data.csv"
Wrong argument count for client. Correct cla should contain 5 arguments.
Success
tgknyhn@tugkan-ubuntu:~/Desktop/midterm$ ./client -p "/tmp/ServerFIFO" -o "data.csv"
Wrong cla format. Proper one should be: "./client -s path_to_server_fifo -o path_to_data_file"
Success
tgknyhn@tugkan-ubuntu:~/Desktop/midterm$
```

Showing Timestamps

To show timestamp, I created a function called **printTimeAndPID** and used the **localtime** function that comes with the "**time.h**" library. I printed the struct I obtained to the stdout file descriptor to show the year, month, day, hour, minute and second information.

```
18/4/2022 | 1:33:25 ->
18/4/2022 | 1:33:30 ->
```

Printing Out Elapsed Time

I measured the time between the client's request and the response it received using the **gettimeofday()** function in the "**sys/time.h**" library. I put the two **timeval** struct objects at the beginning and the end of the main function and print the difference to the console.

```
tgknyhn@tugkan-ubuntu:~/Desktop/midterm$ ./client -s "/tmp/ServerFIFO" -o "data.csv"
18/4/2022 | 2:24:2 -> Client PID#95852 (data.csv) is submitting a 3x3 matrix
18/4/2022 | 2:24:7 -> Client PID#95852: the matrix is not invertible, total time 5.0 seconds, goodbye.
tgknyhn@tugkan-ubuntu:~/Desktop/midterm$
```

Signal Handling

I used the sigaction function from the "signal.h" library to take action against incoming SIGINT signals. Instead of making costy calls inside the handler function, I just set a flag and ran the terminateProgram function that I wrote when this flag was detected outside.

```c
void handleSIG(int signal) {
    switch (signal) {
        case SIGINT:
            sigintFlag = 1;
            break;
        default:
            break;
    }
}
```

```
tgknyhn@tugkan-ubuntu:~/Desktop/midterm$ ./client -s "/tmp/ServerFIFO" -o "data.csv"
18/4/2022 | 2:36:39 -> Client PID#96606 (data.csv) is submitting a 3x3 matrix
^C
18/4/2022 | 2:36:39 -> Client PID#96606 SIGINT signal detected. Terminating the program.
tgknyhn@tugkan-ubuntu:~/Desktop/midterm$
```

Sending Requests & Receiving Responses

To send requests and receive responses I used **FIFO**. While sending the requests, I filled necessary information into **request_t** struct which I wrote(inside **"fifo.h"** library). Then I sent this struct object to read-end side of the FIFO to be read by the serverY. Then I waited for the response from serverY by using read end side of the FIFO for the client.

```c
typedef struct request_s
{
    pid_t pid;
    char pathToDataFile[MAX_FILE_LEN];
} request_t;

typedef struct response_s
{
    int isInvertible;
} response_t;

typedef struct result_s
{
    int total;
    int invertible;
    int nonInvertible;
} result_t;
```

Inside clientX there are total of 2 FIFOs and first one used to send request(**server FIFO**) while other one is used for receiving response(**client FIFO**).

While it is easy to implement client, it is hard to say the same thing for both serverY and serverZ. Thus, explaining these two server will contain a lot of design decisions.

serverY

Controlling Command Line Arguments

Same as clientX, I started implementing the server by checking the command line arguments. If the given arguments are wrong or pool sizes are unsufficient, I informed the user and exited the program.

```
tgknyhn@tugkan-ubuntu:~/Desktop/midterm$ ./serverY -s "/tmp/ServerFIFO" -o "result.log" -p 2 -r 50 -t
Wrong argument count for serverY. Correct cla should contain 11 arguments.
"Success
tgknyhn@tugkan-ubuntu:~/Desktop/midterm$ ./serverY -s "/tmp/ServerFIFO" -o "result.log" -p 1 -r 6 -t 2
Pool sizes can't be lower than 2.
"Success
tgknyhn@tugkan-ubuntu:~/Desktop/midterm$ ./serverY -o "/tmp/ServerFIFO" -p "result.log" -r 5 -a 6 -b 2
Wrong argument syntax. Proper one should be as: "./serverY -s '/tmp/ServerFIFO' -o 'result.log' -p 5 -r 4 -t 5"
"Success
```

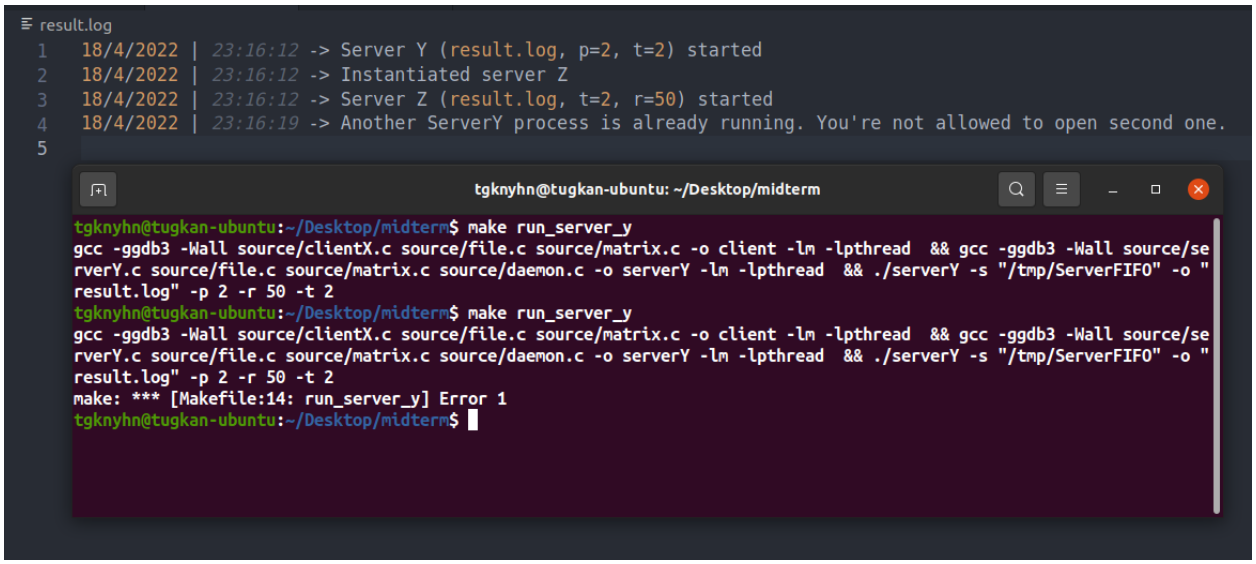
Making Server Daemon

Since serverY and serverZ are both daemon processes, I used the **becomeDaemon** function that is available at lecture slides. To make server work, I used **BD_NO_CHDIR** and **BD_NO_CLOSE_FILES** flags while executing the function. The reason why I used these flags is; relative path doesn't work when I don't put **BD_NO_CHDIR** flag and FIFO files doesn't work as it should be when I don't put **BD_NO_CLOSE_FILES** flag.

To prevent double instantiation, I wrote a function which named **preventDoubleInstance**. Inside this function, I opened the given file(as a parameter) with **O_CREAT | O_EXCL** flags. If this file is not created yet, then file is created and then server starts. But, if this file is already created(by a server which is running), then **open_result** flag is set as -1

and second instance of serverY executes **exitWithErrorServer** which takes error message as argument and exits the program after printing it to the log file.

To unlink the file, I executed **atexit** function with my **unlinkServer** function as parameter. It is only executed by first instance of the server, thus other instances can't execute this line.



Communication Between Client and ServerY

The requests sent by clients are handled by parent ServerY process. The FIFO I used is opened inside infinite parent for loop with read-end side. After getting a request, parent process stops waiting and stores given request inside **request_t** struct object. Then, this time it opens a pipe and sends the request object to any child process available at the moment. The response created by child process then sent back to client using FIFO

Communication Between ServerY and Workers

There are 3 important functions I used here. First function is **createChildrenProcesses.** This function creates pool_size amount of children processes using **fork** function. Then assigns their pid to a global

pid_t pointer array to store their pid values(later it is used to prevent zombie processes).

Second function is **parentProcessY.** This function runs in a infinite for loop waiting for requests to come(from clients). After getting a request with FIFO, it uses write function to send request to a child process. A pipe is initialized before calling **createChildrenProcesses** function, thus every child and parent process has acces to them. This function also forwards any incoming requests to the serverZ if busy count of children processes equals to pool_size. To control busy count I used an integer pointer in a shared memory.

Third function is **childProcessY.** After createChildrenProcesses function, every child created calls childProcessY function. Once they are in, they loop in an infinite for loop. They don't do any busy waiting since they are idle until a request is sent by a parent process through pipe.

Scheduling in serverY with their worker processes is easy because pipes are handling all of it for us. When parent process receives a request it forwards it to write-end of the pipe and whichever child is using the CPU at the moment receives the request and handles it.

Matrix Inverse Calculation

To make it simple, easy to read and write I created my own header and c file for matrices(**matrix.c** and **matrix.h**). I determined if a matrix is invertible or not by looking their determinant value. If determinant is 0, then given matrix is not invertible.

To calculate determinant value, I used cofactor method. Since we need to pick a row or column to calculate determinant, I chose first row of every matrix. For example, if we choose a 3x3 matrix, then equation becomes :

$$det = (a_{11} * A_{11}) + (a_{12} * A_{12}) + (a_{13} * A_{13})$$

Here 'a_{11}' means value of element at first row and first column.

And 'A_{11}' means cofactor value of matrix element at first row and first column. By using this equation it is easy to implement determinant function.

Some people starts calculating determinant directly at 3x3 matrix and finishes their recursion but I decremented my matrices to 2x2 to calculate their determinant. Because calculating determinant of a 2x2 matrix is much easier than 3x3. Inside matrix library, there are 3 important function that calculates and returns the result we want.

First one is **_get_cofactor.** This function takes 2D int array with column and row values, ignores these row and columns and calculates determinant of remaining matrix. After getting determinant value, it multiplies it with (-1)^(row+colum). Then returns the cofactor value.

Second one is **_get_determinant.** This function takes 2D int array as well and if given array size is 2x2, directly calculates determinant and returns it, else it uses equation I wrote above to calculate determinant. After calculation is done, it returns the determinant value.

Third and final one is **is_matrix_invertible.** This function only calls **_get_determinant** function to get determinant value. Then returns 1 if determinant value is different than 0. Returns 0 when determinant is 0.

Signal Handling and Termination

To change initial behavior of SIGINT signal, I used a function named **initializeSignalHandler.** This function is called right after becomeDaemon, meaning I called it right at the start. Inside the function, I used **sigaction** function and made error checking.

As a handler function to sigaction I gave **handleSig** function(same as clientX). Inside handleSig, I only changed sigintFlag to SIGNAL(a define value with 1, it is initially NO_SIGNAL which value is 0).

Instead of checking flag value only inside infinite for loop of server and workers, I made flag if condition available in everywhere as much as can. Thus, my program became more stable and handled SIGINT signals more efficiently also correctly.

Handling SIGINT before executing fork is easy but after creating a lot of child processes it is important to handle it right. When serverY breaks infinite for loop it enters to termination part. Here it sends SIGINT signal to

every child process including QueueWorker, serverZ, serverZ workers and serverY workers.

Since I already stored pid values of every children, I waited every child process to terminate before executing **terminateServerY** function.

Once every child process terminate, **terminateServerY** is called by only parent process since deallocating and unlinking operations must be done once. Inside **terminateServerY**, I wrote every information to log file(saying SIGINT detected, how much matrices handled, forwarded, etc.). After writing is done, I closed all pipes, semaphore variables using sem_close, I unlinked fifo files, semaphores. Then, clean-up for shared memory segments. Finally I exit with success.

serverZ

Communication Between ServerY and ServerZ

Any request that forwarded from serverY to serverZ is sent through pipe. Once serverZ receives a request it puts this request into queue. It doesn't directly send to a child(worker). If queue has request and if there are available children then request is sent to the available child.

Communication Between ServerZ and Workers

Instead of using pipes, I used **semaphores** to achieve mutual exclusion and prevent race condition. To add semaphores to shared memory space, I used **mmap** function. Then I used **sem_init** to give semaphores their initial values and allow sharing between processes.

Once semaphores are placed, everything is similar to serverY and their workers. Different from serverY, this time request is not sent to worker by server but queue. My queue has a fixed size limit(I made it 1000 but it can be every value as long as RAM is big enough).

```
void queueWorker() {
    for(;;) {
        sem_wait(sem_available_child); if(sigintFlag == SIGNAL) break;
        sem_wait(sem_queue); if(sigintFlag == SIGNAL) break;

        // Telling one child process can answer this request
        sem_post(sem_workerZ);
        if(sigintFlag == SIGNAL) break;
    }
}
```

There are 3 semaphores to make queue system work:

sem_available_child, waits until a child process is available. Once a child process reaches start of their for loop, they send a sem_post, indicating that they are ready to take a request.

Once first semaphore passed, then **sem_queue,** halts the queueWorker until a request reaches to queue. Then, since a child process is ready and a request is available at queue, **queueWorker** says “ok, go get the request.”.

By the way, **queueWorker** is an another process created using fork when serverY started. It is only responsible for making communication between serverZ and it’s workers(children).

Writing functions for queue(adding request to queue, removing request from queue) is rather easy since I used fixed size array.

Signal Handling and Termination

Since signal handler initialized with serverY, there are not much different things happening here from serverY. Once SIGINT sent by serverY to QueueWorker, ServerZ and worker processes, they break their for loop and except serverZ all of them terminates.

serverZ doesn’t terminate until every worker of it terminates (pool_size2). Then serverZ closes pipes and terminates.

***

– Execution Examples –

$ run.sh

```
#!/bin/bash
for ((i = 0; i < 5; i++))
do
  ./client -s "/tmp/ServerFIFO" -o "data.csv" &
  sleep .1
  ./client -s "/tmp/ServerFIFO" -o "data2.csv" &
  sleep .1
done
```

```
tgknyhn@tugkan-ubuntu:~/Desktop/midterm$ ./run.sh
18/4/2022 | 23:24:53 -> Client PID#20212 (data.csv) is submitting a 3x3 matrix
18/4/2022 | 23:24:53 -> Client PID#20224 (data2.csv) is submitting a 3x3 matrix
18/4/2022 | 23:24:53 -> Client PID#20227 (data.csv) is submitting a 3x3 matrix
18/4/2022 | 23:24:53 -> Client PID#20230 (data2.csv) is submitting a 3x3 matrix
18/4/2022 | 23:24:54 -> Client PID#20233 (data.csv) is submitting a 3x3 matrix
18/4/2022 | 23:24:54 -> Client PID#20236 (data2.csv) is submitting a 3x3 matrix
18/4/2022 | 23:24:54 -> Client PID#20239 (data.csv) is submitting a 3x3 matrix
18/4/2022 | 23:24:54 -> Client PID#20243 (data2.csv) is submitting a 3x3 matrix
18/4/2022 | 23:24:54 -> Client PID#20246 (data.csv) is submitting a 3x3 matrix
18/4/2022 | 23:24:54 -> Client PID#20249 (data2.csv) is submitting a 3x3 matrix
tgknyhn@tugkan-ubuntu:~/Desktop/midterm$ 18/4/2022 | 23:24:58 -> Client PID#20212: the matrix is not invertible, total time 5.0 seconds, goodbye.
18/4/2022 | 23:24:58 -> Client PID#20224: the matrix is invertible, total time 5.0 seconds, goodbye.
18/4/2022 | 23:24:58 -> Client PID#20227: the matrix is not invertible, total time 5.0 seconds, goodbye.
18/4/2022 | 23:24:58 -> Client PID#20230: the matrix is invertible, total time 5.0 seconds, goodbye.
18/4/2022 | 23:24:59 -> Client PID#20233: the matrix is not invertible, total time 5.0 seconds, goodbye.
18/4/2022 | 23:25:3 -> Client PID#20236: the matrix is invertible, total time 9.69 seconds, goodbye.
18/4/2022 | 23:25:3 -> Client PID#20239: the matrix is not invertible, total time 9.69 seconds, goodbye.
18/4/2022 | 23:25:4 -> Client PID#20243: the matrix is invertible, total time 10.969 seconds, goodbye.
18/4/2022 | 23:25:8 -> Client PID#20246: the matrix is not invertible, total time 14.38 seconds, goodbye.
18/4/2022 | 23:25:8 -> Client PID#20249: the matrix is invertible, total time 14.38 seconds, goodbye.
tgknyhn@tugkan-ubuntu:~/Desktop/midterm$
```

result.log

```
18/4/2022 | 23:24:53 -> Server Y (result.log, p=2, t=5) started
18/4/2022 | 23:24:53 -> Instantiated server Z
18/4/2022 | 23:24:53 -> Server Z (result.log, t=5, r=3) started
18/4/2022 | 23:24:53 -> Worker PID#20220 is handling client PID#20212, matrix size 3x3, pool 1/2
18/4/2022 | 23:24:53 -> Worker PID#20218 is handling client PID#20224, matrix size 3x3, pool 2/2
18/4/2022 | 23:24:53 -> Forwarding request of client PID#20227 to serverZ, matrix size 3x3, pools busy 2/2
18/4/2022 | 23:24:53 -> Z:Worker PID#20219 is handling client PID#20227, matrix size 3x3, pool 1/3
18/4/2022 | 23:24:53 -> Forwarding request of client PID#20230 to serverZ, matrix size 3x3, pools busy 2/2
18/4/2022 | 23:24:53 -> Z:Worker PID#20217 is handling client PID#20230, matrix size 3x3, pool 2/3
18/4/2022 | 23:24:54 -> Forwarding request of client PID#20233 to serverZ, matrix size 3x3, pools busy 2/2
18/4/2022 | 23:24:54 -> Z:Worker PID#20221 is handling client PID#20233, matrix size 3x3, pool 3/3
18/4/2022 | 23:24:54 -> Forwarding request of client PID#20236 to serverZ, matrix size 3x3, pools busy 2/2
18/4/2022 | 23:24:54 -> Forwarding request of client PID#20239 to serverZ, matrix size 3x3, pools busy 2/2
18/4/2022 | 23:24:54 -> Forwarding request of client PID#20243 to serverZ, matrix size 3x3, pools busy 2/2
18/4/2022 | 23:24:54 -> Forwarding request of client PID#20246 to serverZ, matrix size 3x3, pools busy 2/2
18/4/2022 | 23:24:54 -> Forwarding request of client PID#20249 to serverZ, matrix size 3x3, pools busy 2/2
18/4/2022 | 23:24:58 -> Worker PID#20220 is responding to client PID#20212: the matrix IS NOT invertible
18/4/2022 | 23:24:58 -> Worker PID#20218 is responding to client PID#20224: the matrix IS invertible
18/4/2022 | 23:24:58 -> Z:Worker PID#20219 is responding to client PID#20227: the matrix IS NOT invertible
18/4/2022 | 23:24:58 -> Z:Worker PID#20219 is handling client PID#20236, matrix size 3x3, pool 3/3
18/4/2022 | 23:24:58 -> Z:Worker PID#20217 is responding to client PID#20230: the matrix IS invertible
18/4/2022 | 23:24:58 -> Z:Worker PID#20217 is handling client PID#20239, matrix size 3x3, pool 3/3
18/4/2022 | 23:24:59 -> Z:Worker PID#20221 is responding to client PID#20233: the matrix IS NOT invertible
18/4/2022 | 23:24:59 -> Z:Worker PID#20221 is handling client PID#20243, matrix size 3x3, pool 3/3
18/4/2022 | 23:25:3 -> Z:Worker PID#20219 is responding to client PID#20236: the matrix IS invertible
18/4/2022 | 23:25:3 -> Z:Worker PID#20219 is handling client PID#20246, matrix size 3x3, pool 3/3
18/4/2022 | 23:25:3 -> Z:Worker PID#20217 is responding to client PID#20239: the matrix IS NOT invertible
18/4/2022 | 23:25:3 -> Z:Worker PID#20217 is handling client PID#20249, matrix size 3x3, pool 3/3
18/4/2022 | 23:25:4 -> Z:Worker PID#20221 is responding to client PID#20243: the matrix IS invertible
18/4/2022 | 23:25:8 -> Z:Worker PID#20219 is responding to client PID#20246: the matrix IS NOT invertible
18/4/2022 | 23:25:8 -> Z:Worker PID#20217 is responding to client PID#20249: the matrix IS invertible
18/4/2022 | 23:25:42 -> Z:SIGINT received, received, exiting server Z.  Total requests handled 8, 4 invertible, 4 not.
18/4/2022 | 23:25:43 -> SIGINT received, terminating Z and exiting server Y.  Total requests handled: 10, 5 invertible, 5 not. 8 requests were forwarded.

```

```
18/4/2022 | 23:24:58 -> Z:Worker PID#20219 is handling client PID#20236, matrix size 3x3, pool 3/3
18/4/2022 | 23:24:58 -> Z:Worker PID#20217 is responding to client PID#20230: the matrix IS invertible
18/4/2022 | 23:24:58 -> Z:Worker PID#20217 is handling client PID#20239, matrix size 3x3, pool 3/3
18/4/2022 | 23:24:59 -> Z:Worker PID#20221 is responding to client PID#20233: the matrix IS NOT invertible
18/4/2022 | 23:24:59 -> Z:Worker PID#20221 is handling client PID#20243, matrix size 3x3, pool 3/3
18/4/2022 | 23:25:3 -> Z:Worker PID#20219 is responding to client PID#20236: the matrix IS invertible
18/4/2022 | 23:25:3 -> Z:Worker PID#20219 is handling client PID#20246, matrix size 3x3, pool 3/3
18/4/2022 | 23:25:3 -> Z:Worker PID#20217 is responding to client PID#20239: the matrix IS NOT invertible
18/4/2022 | 23:25:3 -> Z:Worker PID#20217 is handling client PID#20249, matrix size 3x3, pool 3/3
18/4/2022 | 23:25:4 -> Z:Worker PID#20221 is responding to client PID#20243: the matrix IS invertible
18/4/2022 | 23:25:8 -> Z:Worker PID#20219 is responding to client PID#20246: the matrix IS NOT invertible
18/4/2022 | 23:25:8 -> Z:Worker PID#20217 is responding to client PID#20249: the matrix IS invertible
18/4/2022 | 23:25:42 -> Z:SIGINT received, received, exiting server Z.  Total requests handled 8, 4 invertible, 4 not.
18/4/2022 | 23:25:43 -> SIGINT received, terminating Z and exiting server Y.  Total requests handled: 10, 5 invertible, 5 not. 8 requests were forwarded.
```

tgknyhn@tugkan-ubuntu: ~/Desktop/midterm

```
tgknyhn@tugkan-ubuntu:~/Desktop/midterm$ make run_server_y
gcc -ggdb3 -Wall source/clientX.c source/file.c source/matrix.c -o client -lm -lpthread  && gcc -ggdb3 -Wall source/serverY.c source/file.c source/matrix.c source/daemon.c -o serverY -lm -lpthread  &&
 ./serverY -s "/tmp/ServerFIFO" -o "result.log" -p 2 -r 3 -t 5
tgknyhn@tugkan-ubuntu:~/Desktop/midterm$ killall -s SIGINT serverY
tgknyhn@tugkan-ubuntu:~/Desktop/midterm$
```