CSE 344

MIDTERM

Burak Kocausta

1901042605

Content

1- System Architecture and Design Decisions

- I planned communication between processes with using shared memory, and fifo. I synchronize them with fifo except resource management. Client processes, send their requests to corresponding server processes. Those server processes are child of main server process. I connect the clients, and servers with fifo. Each client, and child server shares a memory space. I send big chunk of data through that shared memory, communication became fast with this way. For the resource management, I used semaphores. If somebody writes to a file, nobody can access it. I provided mutual exclusion with this way. Reading is free, but writing can cause data race. Using named semaphores, prevented this. For terminating processes, I used signals for communication. Processes are communicating with signals when termination situation occurs. They handle signals properly, clean up, and terminate. I used another semaphore for logging files. All semaphores are named semaphores, and all of them created by main server process. Fifos caused consistent communication, they also automatically block, and guarantees one reader one writer situation. Because of it I chose it. Other than these, no more than max client can be active same time. I held all active process ids in a linked list. Also waiting processes, are held in a queue, because first come will be first served.

2 – Implementation Details

1) Fifos

- I used them for sending, receiving request and response. Request and response are simple structures which includes some variables can be useful for receiver.

```c
typedef struct request_t
{
    pid_t pid;
    int command;
    pid_t sv_pid;
    size_t file_size;
    char file_name[MAX_FILENAME_LEN];
    char message[MAX_BUF_LEN];
    int arg1;
} request_t;

typedef struct response_t
{
    pid_t sv_pid;
    int flag;
    size_t file_size;
} response_t;
```

```c
int get_request(const char *fifo_name, request_t *request);
int send_response(const char *fifo_name, response_t response);
```
child server

```c
int send_request (const char *svc_fifo_name, request_t request);
int get_response (const char *cl_fifo_name, response_t *response);
```
client

2) Client Management

- I held them in a linked list. And all waiting processes are held in queue.

```c
typedef struct cli_queue_t
{
    pid_t pid;
    struct cli_queue_t *next;
} cli_queue_t;

typedef struct cli_wait_queue_t
{
    pid_t pid;
    struct cli_wait_queue_t *next;
} cli_wait_queue_t;
```

```c
int offer_client (pid_t pid);
int poll_client (pid_t *pid);
pid_t peek_client ();
void free_queue ();
void print_queue ();
int remove_client (pid_t pid);
size_t get_num_clients ();

int offer_client_w (pid_t pid);
int poll_client_w (pid_t *pid);
pid_t peek_client_w ();
void free_queue_w ();
void print_queue_w ();
size_t get_num_clients_w ();
int remove_client_w (pid_t pid);
```

- I held ids of client processes in a dynamic memory. So main server have all processes ids. Termination is managed on main server with signals. When they receive signal, they clean up the memory, and terminate.

3) Shared Memory

- I used share memory for transferring large data fastly.

```c
int write_shm_to_file (const char *file_name, void **addr, int shm_fd, size_t size);
int write_file_to_shm (int file_fd, int shm_fd, void **addr, size_t size);
int write_message_shm (void **addr, int shm_fd, const char *message, size_t size);
```

- Some of the shm operations of child server. I held address of the shared memory, and if larger memory requires, it can grow.

```c
/* check if the file size is greater than the shared memory size */
if (file_size > addr_size)
{
    /* resize the shared memory */

    /* unmap the shared memory */
    if (munmap(*addr, addr_size) == -1)
    {
        error_print("munmap");
        close(fd);
        return -1;
    }

    /* truncate the file */
    if (ftruncate(shm_fd, file_size) == -1)
    {
        error_print("ftruncate");
        close(fd);
        return -1;
    }

    /* map the file to the shared memory */
    *addr = mmap(NULL, file_size, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);

    addr_size = file_size;
}
```

4) Signal Handling

- I handled signals with simple flag variable.

```c
sig_atomic_t signal_occured = 0;

void sig_handler (int signal_number)
{
    signal_occured = signal_number;
}
```

all processes signals are controlled with this way.

```c
if (signal_occured > 0)
{
    switch (signal_occured)
    {
        case SIGINT:
            snprintf(buffer, MAX_BUF_LEN, "MAIN SERVER: %d >> SIGINT received. Terminating..\n", pid);
            log_msg(log_file_name, buffer, log_sem);
            cleanup_and_terminate(fifo_name, child_pids, num_child, log_sem, log_sem_name, write_sem, write_sem_name);
        break;

        case SIGTERM:
            snprintf(buffer, MAX_BUF_LEN, "MAIN SERVER: %d >> SIGTERM received. Terminating..\n", pid);
            log_msg(log_file_name, buffer, log_sem);
            cleanup_and_terminate(fifo_name, child_pids, num_child, log_sem, log_sem_name, write_sem, write_sem_name);
        break;

        case SIGQUIT:
            snprintf(buffer, MAX_BUF_LEN, "MAIN SERVER: %d >> SIGQUIT received. Terminating..\n", pid);
            log_msg(log_file_name, buffer, log_sem);
            cleanup_and_terminate(fifo_name, child_pids, num_child, log_sem, log_sem_name, write_sem, write_sem_name);
        break;

        case SIGCHLD:
            snprintf(buffer, MAX_BUF_LEN, "MAIN SERVER: %d >> SIGCHLD received.\n", pid);
            log_msg(log_file_name, buffer, log_sem);
        break;

    }
```

- Then I decide their situation in proper time.

5) Semaphores

- I used 2 types of semaphore for logging, and writing.

`sem_t *log_sem, *write_sem;` they are created and initialized in main server.

```c
/* create log file semaphore */
snprintf(log_sem_name, MAX_FILENAME_LEN, LOG_SEM_NAME, pid);

/* create the semaphore */
if ((log_sem = sem_open(log_sem_name, O_CREAT | O_EXCL, 0666, 1)) == SEM_FAILED)
{
    if (errno != EEXIST)
    {
        /* check its value */
        sem_getvalue(log_sem, &val);
        if (val == 0)
            sem_post(log_sem);
    }
}
```

```c
 /* create write semaphore */
 snprintf(write_sem_name, MAX_FILENAME_LEN, WRITE_SEM_NAME, pid);

 /*    fprintf(stderr, "write_sem_name: %s\n", write_sem_name); */

 /* create the semaphore */
 if ((write_sem = sem_open(write_sem_name, O_CREAT | O_EXCL, 0666, 1)) == SEM_FAILED)
 {
     if (errno != EEXIST)
     {
         /* check its value */
         sem_getvalue(write_sem, &val);
         if (val == 0)
             sem_post(write_sem);
     }
 }
```

6) Implementation of Commands

- Help

Help is implemented with sending messages through shared memory, code of what kind of help are sent with fifo, but printing stuff is sent with shared memory.

```
request.command = CMD_HELP;

if (strlen(buffer) > 5)
{
    /* get the command */
    write_next_word(buffer, request.file_name);

    /* remove the newline character */
    if (request.file_name[strlen(request.file_name) - 1] == '\n')
        request.file_name[strlen(request.file_name) - 1] = '\0';

    if (strcmp(request.file_name, "readF") == 0)
        request.command = CMD_HELP_READF;
    else if (strcmp(request.file_name, "writeT") == 0)
        request.command = CMD_HELP_WRITET;
    else if (strcmp(request.file_name, "upload") == 0)
        request.command = CMD_HELP_UPLOAD;
    else if (strcmp(request.file_name, "download") == 0)
        request.command = CMD_HELP_DOWNLOAD;
    else if (strcmp(request.file_name, "list") == 0)
        request.command = CMD_HELP_LIST;
    else if (strcmp(request.file_name, "quit") == 0)
        request.command = CMD_HELP_QUIT;
    else if (strcmp(request.file_name, "killServer") == 0)
        request.command = CMD_HELP_KILLSERVER;
    else
    {
        fprintf(stderr, "Unknown command. help or help <command>\n");
        continue;
    }
}
```

```
/* print the response which is at the shared memory */

/* critical section */
sem_wait(write_sem);

block_all_signals();
if (print_message_shm(&addr, shm_fd, response.file_size) == -1)
    error_print_custom("print_message_shm");
unblock_all_signals();

sem_post(write_sem);
/* end of critical section */
```

(after getting response)

Server side is writing message to shared memory.

- list

I implemented list command similar to the homework 2. I used pipes for getting the output of ls -l command, and I use fork-exec in the child process. Then connect its stdout to stdin of parent with redirection.

```
/* child process */
else if (pid == 0) {

    /* close read end */
    close(pipefd[0]);

    /* redirect stdout to pipe */
    dup2(pipefd[1], STDOUT_FILENO);
    close(pipefd[1]);

    /* execute ls <dir_name> */
    execlp("ls", "ls", dir_name, NULL);

    perror("execlp");
    exit(1);
}
```

```
/* parent process */
/* close write end */
close(pipefd[1]);

/* wait for child */
if (wait(&status) == -1) {
    perror("wait");
    return -1;
}

/* check child status */
if (!WIFEXITED(status)) {
    perror("failed to execute ls");
    return -1;
}

/* read from pipe */
read_bytes = read(pipefd[0], buffer, sizeof(buffer));
if (read_bytes == -1) {
    perror("read");
    return -1;
}

/* close read end */
close(pipefd[0]);
```

then I write the data to shared memory.

- readF

I implemented readF, with receiving the output from shared memory.

Server writes to shared memory, wanted line or whole file. Then client reads from there. Synchronization is provided with fifos, and semaphore.

```
case CMD_READF:
    response.flag = SV_SUCCESS;

    /* get file name */
    snprintf(file_name, MAX_FILENAME_LEN + 1, "%s/", dir_name);
    strncat(file_name, request.file_name, MAX_FILENAME_LEN - strlen(file_name));

    block_all_signals();
    /* open file */
    if ((file_fd = open(file_name, O_RDONLY)) == -1)
        response.flag = SV_FAILURE;
    unblock_all_signals();

    block_all_signals();
    /* enter critical section for read */
    sem_wait(write_sem);
    sem_post(write_sem);

    if (response.flag != SV_FAILURE &&
            write_line_to_shm(file_fd, shm_fd, &addr, &response.file_size, request.arg1) == -1)
    {
        error_print_custom("readF failed");
        response.flag = SV_FAILURE;
    }
    unblock_all_signals();

    /* send response to the client */
    if ((send_response(cl_fifo_name, response)) == -1)
        error_print_custom("readF failed");

    snprintf(buffer, MAX_BUF_LEN, "CHILD SERVER: %d >> readF command is executed for client %d\n", pid, client_pid);
    log_msg(buffer, log_sem, log_file_name);
break;
```

This is server part

- writeT

I implement the writeT as putting the wanted argument to wanted line. And content of that line is removed, after writing new one. For this I needed temporary storage place, and I used shared memory for that.

```
int write_nth_line(int file_fd, int line_num, const char *message, int shm_fd, void **addr)
```

- upload

I similarly synchronize them with fifos, and semaphores. Client process loads whole data to shared memory, than server process reads from here. No process can access the file before it is completely uploaded to server directory.

```c
case CMD_UPLOAD:

    response.flag = SV_SUCCESS;

    /* get file name */
    snprintf(file_name, MAX_FILENAME_LEN+1, "%s/", dir_name);
    strncat(file_name, request.file_name, MAX_FILENAME_LEN - strlen(file_name));

    block_all_signals();
    /* enter critical section */
    sem_wait(write_sem);

    /* read from shared memory, write to file */
    if (write_shm_to_file(file_name, &addr, shm_fd, request.file_size) == -1)
    {
        fprintf(stderr, "Upload failed\n");
        response.flag = SV_FAILURE;
    }

    /* exit critical section */
    sem_post(write_sem);
    unblock_all_signals();

    /* send response to the client */
    if (send_response(cl_fifo_name, response) == -1)
        fprintf(stderr, "Upload failed\n");

    snprintf(buffer, MAX_BUF_LEN, "CHILD SERVER: %d >> upload command is executed for client %d\n", pid, client_pid);
    log_msg(buffer, log_sem, log_file_name);
break;
```

Server

```c
    /* critical section for reading from file */
    sem_wait(write_sem);
    sem_post(write_sem);

    fprintf(stderr, "> Sending upload request to the server..\n");
    fprintf(stderr, "> Writing file to shared memory..\n");

    block_all_signals();
    if (write_file_to_shm(file_name, &addr, shm_fd) == -1)
        error_print_custom("write_file_to_shm");
    unblock_all_signals();
    fprintf(stderr, "> File written to shared memory..\n");

    strcpy(request.file_name, file_name);

    /* send upload request to the server */
    if (send_request(svc_fifo_name, request) == -1)
    {
        error_print_custom("send_request");
        continue;
    }

    /* get the response */
    if (get_response(cl_fifo_name, &response) == -1)
    {
        error_print_custom("get_response");
        continue;
    }

    if (response.flag == SV_FAILURE)
        error_print_custom("Upload failed");
    else
        fprintf(stderr, "> Upload successful..\n");
```

Some part of the client

- download

I implemented it similar to the upload. This time things become vice versa with upload. They are synchronized with fifo, and semaphores. And one writes data to shared memory, other one reads. No process can access data before it finishes downloading.

- quit

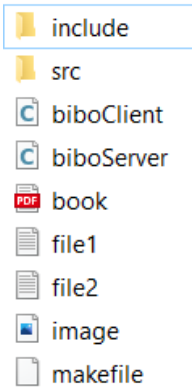
Client process sends the request, main server this time. After sending it, it terminates itself. And main server terminates child server.

- killServer

Client process sends sigint signal to main server, and it terminates everything with proper cleanup.

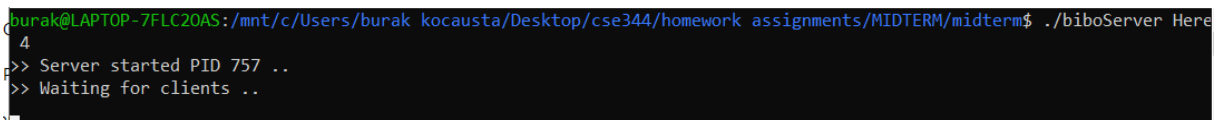
3- Test Results

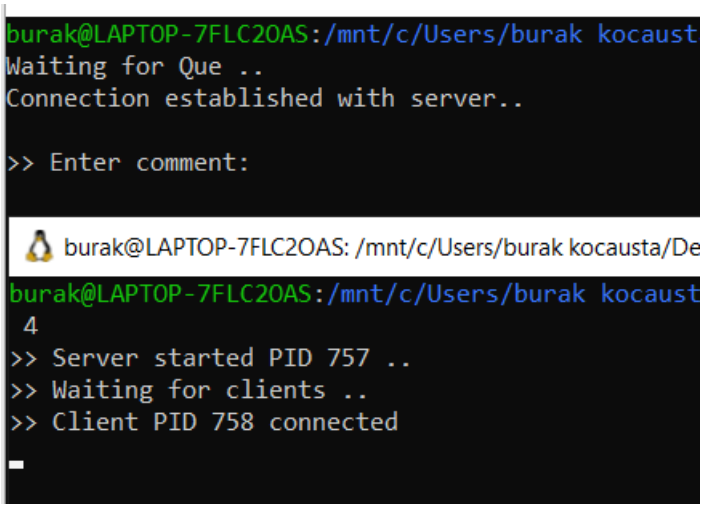
Before Compilation:



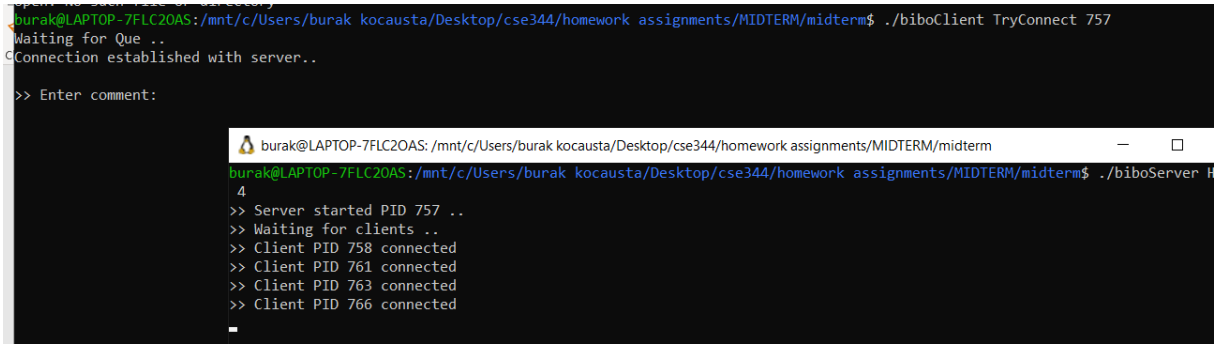
- After compiling log directory will be created.

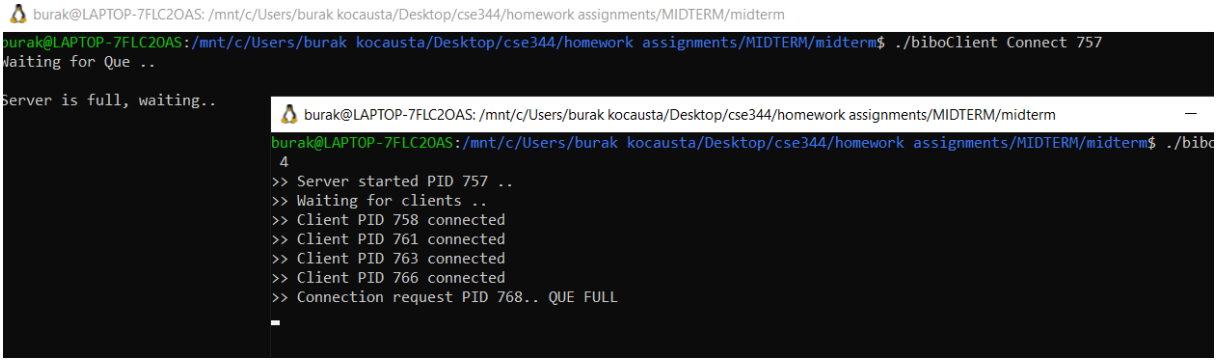
Testing client connection, and try connect:



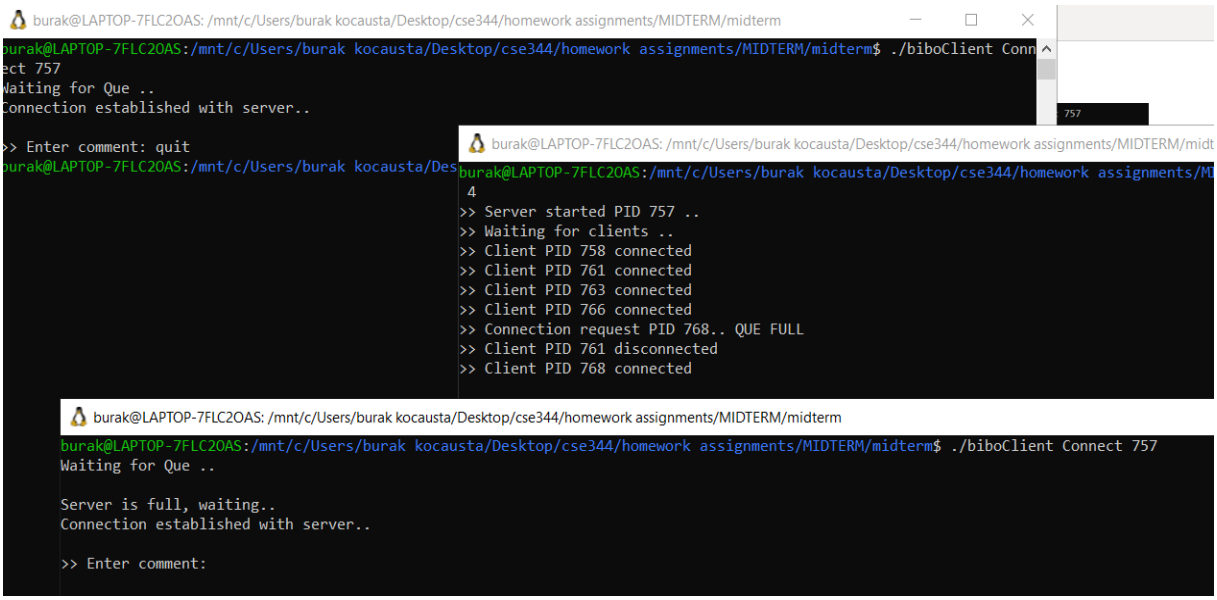


one client is connected





- more than 4 client tried to connect.



- one is disconnected, and waiting client takes.

Testing help:

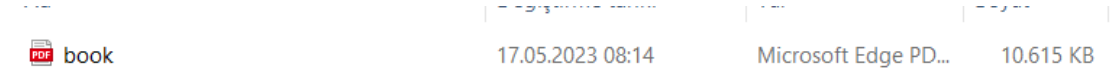
```
>> Enter comment: help

  Available Comments are:
connect, tryConnect, help, list, readF, writeT, upload, download, quit, killServer


>> Enter comment: help list

 list
       display list the files in the server directory.


>> Enter comment: help readF

  readF <file> <line #>
       display the #th line of the <file>, returns with an error if <file> does not exist


>> Enter comment: help writeT

 writeT <file> <line #> <string>
       write <string> to the #th line of the <file>, if <line #> is not given writes end of the file
```

```
>> Enter comment: help killServer

 killServer
       write to log file and kill the server.


>> Enter comment: help quit

 quit
       write to log file and quit.
```

- help command and, help with arguments.

Testing upload:

```
>> Enter comment: upload book.pdf
> Sending upload request to the server..
> Writing file to shared memory..
10869518 number of bytes is loaded on shared memory
> File written to shared memory..
> Upload successful..
```



- File uploaded successfully.

```
Server is full, waiting..
Connection established with server..

>> Enter comment: upload file1.txt
> Sending upload request to the server..
> Writing file to shared memory..
89 number of bytes is loaded on shared memory
> File written to shared memory..
> Upload successful..
```

```
>> Enter comment: upload biboServer
> Sending upload request to the server..
> Writing file to shared memory..
28256 number of bytes is loaded on shared memory
> File written to shared memory..
> Upload successful..
```

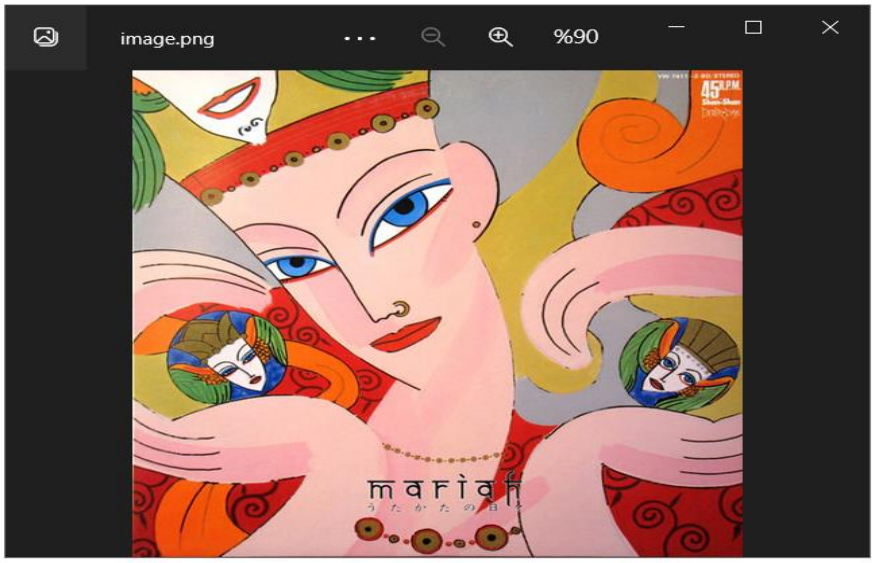
- Upload output for client.

Testing list:

```
>> Enter comment: list
biboServer
book.pdf
file1.txt
```

- Upload another file which is png file

biboServer	17.05.2023 08:15	Dosya	28 KB
book	17.05.2023 08:14	Microsoft Edge PD...	10.615 KB
file1	17.05.2023 08:15	Metin Belgesi	1 KB
image	17.05.2023 08:17	PNG Dosyası	72 KB



It is successfully uploaded.

```
>> Enter comment: list
biboServer
book.pdf
file1.txt
image.png
```

Testing readF:

```
>> Enter comment: readF file1.txt
fagawfawawfgaw
wagawgonwgpa
gapgamfw
abc
a
waawfwafeaw
awgpawgaopgnpawga
awgoawgnoawgpap

>> Enter comment: readF file1.txt 3
gapgamfw
```

```
>> Enter comment: readF book.pdf 5
endobj
>> Enter comment: readF image.png 10
z
```

- reads correct lines.

Testing writeT:

```
>> Enter comment: readF file1.txt
fagawfawawfgaw
wagawgonwgpa
gapgamfw
abc
a
waawfwafeaw
awgpawgaopgnpawga
awgoawgnoawgpap

>> Enter comment: writeT file1.txt 5 "ageaeg awgwg wgwagw"

>> Enter comment: readF file1.txt
fagawfawawfgaw
wagawgonwgpa
gapgamfw
abc
ageaeg awgwg wgwagw
waawfwafeaw
awgpawgaopgnpawga
awgoawgnoawgpap
```

- It successfully replaced.

Testing download:

```
>> Enter comment: download file1.txt
> Downloading file from shared memory..
107 number of bytes is read from shared memory
> File downloaded from shared memory..
> File downloaded..
```

```
>> Enter comment: download book.pdf
> Downloading file from shared memory..
File size: 10869518
Shared memory size: 4096
10869518 number of bytes is read from shared memory
> File downloaded from shared memory..
> File downloaded..
```

- download output.

Testing quit:

```
burak@LAPTOP-7FLC2OAS:/mnt/c/Users/burak koc
 4
>> Server started PID 757 ..
>> Waiting for clients ..
>> Client PID 758 connected
>> Client PID 761 connected
>> Client PID 763 connected
>> Client PID 766 connected
>> Connection request PID 768.. QUE FULL
>> Client PID 761 disconnected
>> Client PID 768 connected
>> Client PID 768 disconnected
```

```
awgpawgaopgnpawga
awgoawgnoawgpap

>> Enter comment: writeT file1.txt 5 "ageaeg awgwg wgwagw"

>> Enter comment: readF file1.txt
fagawfawawfgaw
wagawgonwgpa
gapgamfw
abc
ageaeg awgwg wgwagw
waawfwafeaw
awgpawgaopgnpawga
awgoawgnoawgpap

>> Enter comment: download book.pdf
> Downloading file from shared memory..
File size: 10869518
Shared memory size: 4096
10869518 number of bytes is read from shared memory
> File downloaded from shared memory..
> File downloaded..

>> Enter comment: quit
burak@LAPTOP-7FLC2OAS:/mnt/c/Users/burak kocausta/Desktop/cse
```

- client is disconnected.

Testing killServer:

```
>> Enter comment: killServer
```

```
burak@LAPTOP-7FLC2OAS:/mnt/c/Users/burak kocau
 4
>> Server started PID 757 ..
>> Waiting for clients ..
>> Client PID 758 connected
>> Client PID 761 connected
>> Client PID 763 connected
>> Client PID 766 connected
>> Connection request PID 768.. QUE FULL
>> Client PID 761 disconnected
>> Client PID 768 connected
>> Client PID 768 disconnected
>> Kill signal from client PID 758, terminating..
> Sending SIGINT to child 759
> Sending SIGINT to child 764
> Sending SIGINT to child 767
>> bye..
burak@LAPTOP-7FLC2OAS:/mnt/c/Users/burak kocausta/Desktop/cse344/homework assignments/MIDTERM/midterm$
```

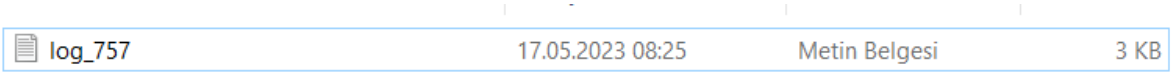
```
> File downloaded from shared memory..
> File downloaded..

>> Enter comment: > SIGINT received. Terminating..
burak@LAPTOP-7FLC2OAS:/mnt/c/Users/burak kocausta/Desktop/cse
344/homework assignments/burak@LAPTOP-7FLC2OAS:/mnt/c/Users/b
urak kocausta/Desktop/cse344/homework assignments/MIDTERM/mid
```

```
burak@LAPTOP-7FLC2OAS: /mnt/c/Users/burak kocausta/Desktop/cse344/homework assignments/MIDTERM
>> Enter comment: help writeT

  writeT <file> <line #> <string>
         write <string> to the #th line of the <file>, if <line #> is not given wr

>> Enter comment: killServer
```

```
>> Enter comment: killServer
burak@LAPTOP-7FLC2OAS:/mnt/c/Users/burak kocausta/Desktop/cse344/homework assignments/MIDTERM/midterm$ ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.0   1056   652 ?        Sl   05:10   0:00 /init
root         654  0.0  0.0   1056   228 ?        Ss   08:06   0:00 /init
root         655  0.0  0.0   1056   228 ?        D    08:06   0:00 /init
burak        656  0.0  0.0  10168  5064 pts/2    Ss   08:06   0:00 -bash
root         684  0.0  0.0   1056   228 ?        Ss   08:06   0:00 /init
root         685  0.0  0.0   1056   228 ?        S    08:06   0:00 /init
burak        686  0.0  0.0  10168  5160 pts/4    Ss+  08:06   0:00 -bash
root         699  0.0  0.0   1056   228 ?        Ss   08:06   0:00 /init
root         700  0.0  0.0   1056   228 ?        S    08:06   0:00 /init
burak        701  0.0  0.0  10168  5032 pts/5    Ss+  08:06   0:00 -bash
root         742  0.0  0.0   1056   228 ?        Ss   08:07   0:00 /init
root         743  0.0  0.0   1056   228 ?        S    08:07   0:00 /init
burak        744  0.0  0.0  10168  5212 pts/0    Ss+  08:07   0:00 -bash
burak        772  0.0  0.0  10616  3256 pts/2    R+   08:26   0:00 ps aux
burak@LAPTOP-7FLC2OAS:/mnt/c/Users/burak kocausta/Desktop/cse344/homework assignments/MIDTERM/midterm$
```

- There aren't any zombie or orphan processes.

Testing Log:



- Inside sv_log directory.

Contents of log file:

```
MAIN SERVER: >> Server started PID 757 ..
MAIN SERVER: 757 >> Client PID 758 connected
CHILD SERVER: 759 >> Client 758 connected to child server
MAIN SERVER: 757 >> Client PID 761 connected
CHILD SERVER: 762 >> Client 761 connected to child server
MAIN SERVER: 757 >> Client PID 763 connected
CHILD SERVER: 764 >> Client 763 connected to child server
MAIN SERVER: 757 >> Client PID 766 connected
CHILD SERVER: 767 >> Client 766 connected to child server
MAIN SERVER: 757 >> Connection request PID 768.. QUE FULL
CHILD SERVER: 762 >> SIGINT received. Terminating..
MAIN SERVER: 757 >> Client PID 761 disconnected
MAIN SERVER: 757 >> SIGCHLD received.
MAIN SERVER: 757 >> Client PID 768 connected
CHILD SERVER: 769 >> Client 768 connected to child server
CHILD SERVER: 759 >> help command is executed for client 758
CHILD SERVER: 759 >> help command is executed for client 758
CHILD SERVER: 759 >> help command is executed for client 758
CHILD SERVER: 759 >> help command is executed for client 758
CHILD SERVER: 764 >> help command is executed for client 763
CHILD SERVER: 764 >> help command is executed for client 763
CHILD SERVER: 764 >> help command is executed for client 763
CHILD SERVER: 767 >> upload command is executed for client 766
CHILD SERVER: 769 >> upload command is executed for client 768
CHILD SERVER: 764 >> upload command is executed for client 763
CHILD SERVER: 764 >> list command executed for client 763
CHILD SERVER: 767 >> upload command is executed for client 766
CHILD SERVER: 767 >> list command executed for client 766
CHILD SERVER: 767 >> readF command is executed for client 766
CHILD SERVER: 767 >> readF command is executed for client 766
CHILD SERVER: 767 >> readF command is executed for client 766
CHILD SERVER: 767 >> readF command is executed for client 766
CHILD SERVER: 769 >> readF command is executed for client 768
CHILD SERVER: 769 >> writeT command is executed for client 768
CHILD SERVER: 769 >> readF command is executed for client 768
CHILD SERVER: 767 >> download command is executed for client 766
CHILD SERVER: 769 >> download command is executed for client 768
CHILD SERVER: 769 >> SIGINT received. Terminating..
MAIN SERVER: 757 >> Client PID 768 disconnected
MAIN SERVER: 757 >> SIGCHLD received.
MAIN SERVER: 757 >> Kill signal from client PID 758, terminating..
CHILD SERVER: 759 >> SIGINT received. Terminating..
CHILD SERVER: 764 >> SIGINT received. Terminating..
CHILD SERVER: 767 >> SIGINT received. Terminating..
```