CSE 344

HW4

Burak Kocausta

1901042605

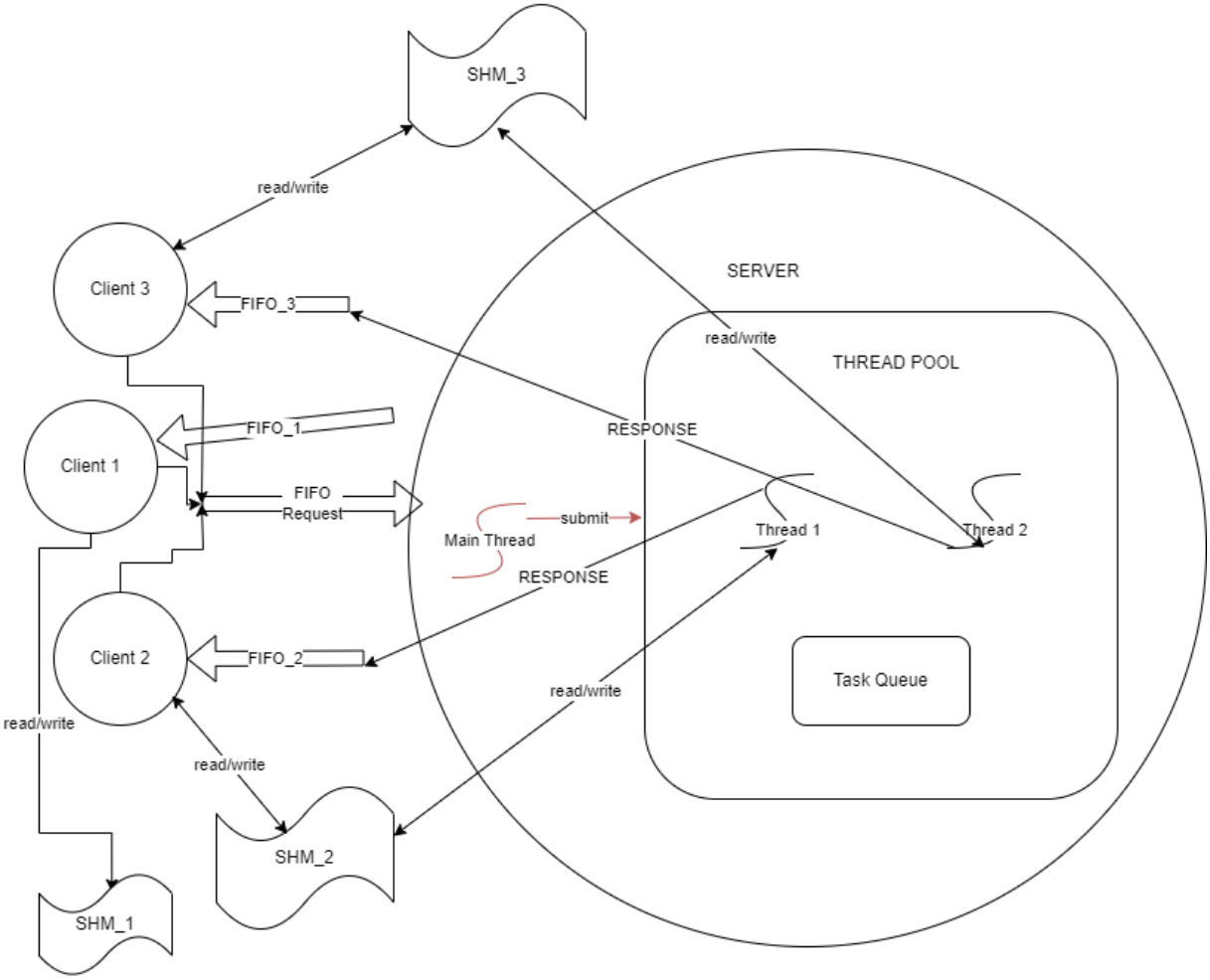
Content

1- System Architecture and Design Decisions
2- Implementation Details
3- Test Results

1- System Architecture and Design Decisions

- I planned communication between client processes and server threads using shared memory, and FIFO. With FIFO, clients are sending requests to the server, after taking the request, empty worker thread performs that command, and sends respond to the client. Command comes to server with FIFO, then one of the sleeping threads is woken up it performs the command. There is only one server process, and it has threads that serves the corresponding command. I send big chunks of data through that shared memory; communication became fast with this way. For the resource management, I used mutexes on the server side. Threads are synchronized with pthread mutexes, and conditional variables. For each request, one of the threads that not working is activated, and performs the given request. I handle this synchronization with conditional variables, and pthread mutexes. For the resources, I used pthread read, and write mutexes. I solved it as a read-write problem. For the client side, I used file locks. Again, it solves the read-write problem. I needed it for writing and reading client-side files. Mutual exclusion is provided in this way. For terminating processes, I used signals for communication between client and server. When one of them is terminated, the other one will be notified with a signal. They handle signals properly, clean up, and terminate. Thread termination is simple, the server process just terminates all threads. I used another mutex for logging files. FIFOs caused consistent communication, they also automatically block, and guarantee one reader one writer situation. Other than these, no more than max client can be active at the 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.

Example situation for, 3 client, and pool size = 2.



- Client 3, and client 2 sends request. Then commands are submitted to task queue, and server threads writes to shared memories, and sends the response through another FIFO.

2 – Implementation Details

1) FIFO

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

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

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

```c
/*
* get request from fifo
*/
int get_request(const char *fifo_name, request_t *request);

/*
* send response to fifo
*/
int send_response(const char *fifo_name, response_t response);
```

```c
/*
* send request to fifo
*/
int send_request (const char *fifo_name, request_t request);

/*
* get response from fifo
*/
int get_response (const char *fifo_name, response_t *response);
```

- Methods for sending request, and response through FIFO.

2) Threads and Task Queue

- I used pthread library for handling threads. I created a structure for thread pool and task queue. The task queue is inside thread pool structure. Also, threads are held in thread pool dynamically.

```c
typedef struct task_queue_t
{
    request_t task;
    struct task_queue_t *next;
} task_queue_t;
```

```c
int offer_task (const request_t *task);

int poll_task (request_t *task);

int peek_task (request_t *task);
```

```c
int task_queue_init ();

int task_queue_destroy ();

size_t task_queue_get_size ();
```

Task queue structure and its methods.

```c
typedef struct thread_pool_t
{
    pthread_t *threads;
    size_t num_threads;
    size_t num_threads_working;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
    pthread_mutex_t log_mutex;
    pthread_rwlock_t rwlock;
    int shutdown;
    task_queue_t *task_queue;
} thread_pool_t;
```

Thread pool structure, for managing threads.

```c
int thread_pool_init (size_t num_threads);

int thread_pool_destroy ();

int thread_pool_submit (const request_t *task);

int thread_pool_get_size ();

int thread_pool_get_num_threads_working ();

void *worker_thread (void *args);

int execute_task (const request_t *task);

void log_msg (const char *log_file_name, const char *msg);
```

Thread Management

```c
void *worker_thread (void *args)
{
    request_t task;

    /* for preventing unused parameter warning */
    (void) args;

    while (1)
    {
        /* lock mutex and wait for signal */
        if (pthread_mutex_lock(&(thread_pool.mutex)) != 0)
        {
            error_print("pthread_mutex_lock");
            return NULL;
        }

        while (task_queue_get_size() == 0 && !thread_pool.shutdown)
        {
            if (pthread_cond_wait(&(thread_pool.cond), &(thread_pool.mutex)) != 0)
            {
                error_print("pthread_cond_wait");
                return NULL;
            }
        }

        /* shutdown */
        if (thread_pool.shutdown)
        {
            if (pthread_mutex_unlock(&(thread_pool.mutex)) != 0)
            {
                error_print("pthread_mutex_unlock");
                return NULL;
            }
```

```
        /* grab task */
        if (poll_task(&task) == -1)
        {
            error_print("poll_task");
            return NULL;
        }

        /* increment number of threads working */
        thread_pool.num_threads_working++;

        /* unlock mutex */
        if (pthread_mutex_unlock(&(thread_pool.mutex)) != 0)
        {
            error_print("pthread_mutex_unlock");
            return NULL;
        }

        block_all_signals();
        /* execute task */
        execute_task(&task);
        unblock_all_signals();

        /* decrement number of threads working */
        thread_pool.num_threads_working--;
    }

    return NULL;
}
```

- When there are no tasks, it is waiting. If task is submitted, condition signal occurs, mutex is locked and worker thread takes the task from queue, and then unlocks the mutex. This algorithm is necessary because only one thread should take the task.

3) Client Management

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

```
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;
```

```
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.

4) Shared Memory

- I used share memory for transferring large data fastly.

```
int write_shm_to_file (const char *file_name, const char *shm_name, size_t size);
int write_file_to_shm (int file_fd, const char *shm_name, size_t size);
```

- Main shm operations of server. It opens the shared memory that is created by client, and writes/reads from there. It makes all read/write operations according to the request given by the client.

5) Signal Handling

- I handled signals with simple flag variable.

```
sig_atomic_t signal_occured = 0;

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

- all processes signals are controlled with this way.

```
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);
            cleanup_and_terminate(cl_fifo_name);
        break;

        case SIGTERM:
            snprintf(buffer, MAX_BUF_LEN, "MAIN SERVER: %d >> SIGTERM received. Terminating..\n", pid);
            log_msg(log_file_name, buffer);
            cleanup_and_terminate(cl_fifo_name);
        break;

        case SIGQUIT:
            snprintf(buffer, MAX_BUF_LEN, "MAIN SERVER: %d >> SIGQUIT received. Terminating..\n", pid);
            log_msg(log_file_name, buffer);
            cleanup_and_terminate(cl_fifo_name);
        break;
    }

    signal_occured = 0;
}
```

- Then I decide their situation in proper time.

```c
/*
* blocks all signals
*/
void block_all_signals ( );

/*
* unblocks all signals
*/
void unblock_all_signals ( );
```

- I used these functions for the situations for preventing signal interruptions.

6) Read Write Mutexes

- I used this mutex for solving read-write problem.

```c
/* enter critical section for read */
if (pthread_rwlock_rdlock(&(thread_pool.rwlock)) != 0)
{
    error_print("pthread_rwlock_rdlock");
    response.flag = SV_FAILURE;
    close(file_fd);
    send_response(sv_fifo_name, response);
    return -1;
}
```

```c
/* exit critical section */
if (pthread_rwlock_unlock(&(thread_pool.rwlock)) != 0)
{
    error_print("pthread_rwlock_unlock");
    response.flag = SV_FAILURE;
    close(file_fd);
    send_response(sv_fifo_name, response);
    return -1;
}
```

- Example usage of read lock.

7) File Locks

- I solved read-write problem for client using file locks.

```c
int read_lock_file (int fd, struct flock *fl)
{
    /* set the lock type */
    fl->l_type = F_RDLCK;

    /* set the lock to the entire file */
    fl->l_whence = SEEK_SET;
    fl->l_start = 0;
    fl->l_len = 0;

    /* set the lock */
    if (fcntl(fd, F_SETLKW, fl) == -1)
    {
        error_print("fcntl, read lock");
        return -1;
    }

    return 0;
}
```

```c
int write_lock_file (int fd, struct flock *fl)
{
    /* set the lock type */
    fl->l_type = F_WRLCK;

    /* set the lock to the entire file */
    fl->l_whence = SEEK_SET;
    fl->l_start = 0;
    fl->l_len = 0;

    /* set the lock */
    if (fcntl(fd, F_SETLKW, fl) == -1)
    {
        error_print("fcntl, write lock");
        return -1;
    }

    return 0;
}
```

- I used these functions for locking, after job is done it is unlocked. It is used for client side resources.

8) Implementation of Commands

- Commands are sent through fifo to server. Then worker threads handles the 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.

```c
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;
    }
}
```

- 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.

```c
/* 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);
}
```

```c
/* 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, task->file_name, MAX_FILENAME_LEN - strlen(file_name));

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

    /* enter critical section for read */
    if (pthread_rwlock_rdlock(&(thread_pool.rwlock)) != 0)
    {…
    }

    if (response.flag != SV_FAILURE)
    {
        if (write_line_to_shm(file_fd, shm_name,
                &response.file_size, task->arg1) == -1)
        {
            error_print_custom("readF failed");
            response.flag = SV_FAILURE;
        }
    }

    /* exit critical section */
    if (pthread_rwlock_unlock(&(thread_pool.rwlock)) != 0)
    {…
    }
```

```
/* send response */
if (send_response(sv_fifo_name, response) == -1)
{
    error_print_custom("send_response failed");
    return -1;
}

snprintf(buffer, MAX_BUF_LEN, "SERVER THREAD: >> readF comm
log_msg(log_file_name, buffer);
```

- 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, const char *shm_name)
```

upload

I similarly synchronize them with rw mutexes. Client process loads whole data to shared memory, then server process reads from here. No other threads can access the file before it is completely uploaded to server directory.

```
case CMD_UPLOAD:

    response.flag = SV_SUCCESS;

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

    /* critical section for write */
    if (pthread_rwlock_wrlock(&(thread_pool.rwlock)) != 0)
    {...
    }

    /* read from shared memory, write to file */
    if (write_shm_to_file(file_name, shm_name, task->file_size) == -1)
    {
        error_print_custom("write_shm_to_file failed");
        response.flag = SV_FAILURE;
    }

    /* exit critical section */
    if (pthread_rwlock_unlock(&(thread_pool.rwlock)) != 0)
    {...
    }

    /* send response to the client */
    if (send_response(sv_fifo_name, response) == -1)
    {
        error_print_custom("send_response failed");
        return -1;
    }

    snprintf(buffer, MAX_BUF_LEN, "SERVER THREAD: >> upload command is executed for client
    log_msg(log_file_name, buffer);
break;
```

- This is server side

```
else if (strncmp(buffer, "upload", 6) == 0)
{
    request.command = CMD_UPLOAD;

    if (strlen(buffer) <= 7)
    {…
    }

    /* get the filename */
    write_next_word(buffer, file_name);

    block_all_signals();
    /* check if file exists */
    temp_fd = open(file_name, O_RDONLY);
    if (temp_fd == -1)
    {…
    }
    unblock_all_signals();
    /* get the file size */
    request.file_size = lseek(temp_fd, 0, SEEK_END);

    if (close(temp_fd) == -1)
        error_print("close");
```

```
/* write file to the shared memory */

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(cl_fifo_name, request) == -1)
{…
}

/* get the response */
if (get_response(sv_fifo_name, &response) == -1)
{…
}

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

- This client side

download

I implemented it similar to the upload. This time things become vice versa with upload. They are synchronized with fifo, mutex(threads), file lock(client). And one writes data to shared memory, other one reads. No process can access data before it finishes downloading.

quit

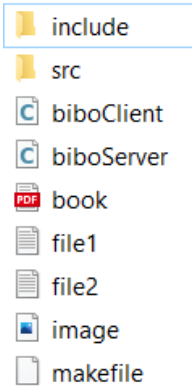
Client process sends the request, main server this time. After sending it, it terminates itself. And main server terminates child server.

killServer

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

3- Test Results

Before Compilation:



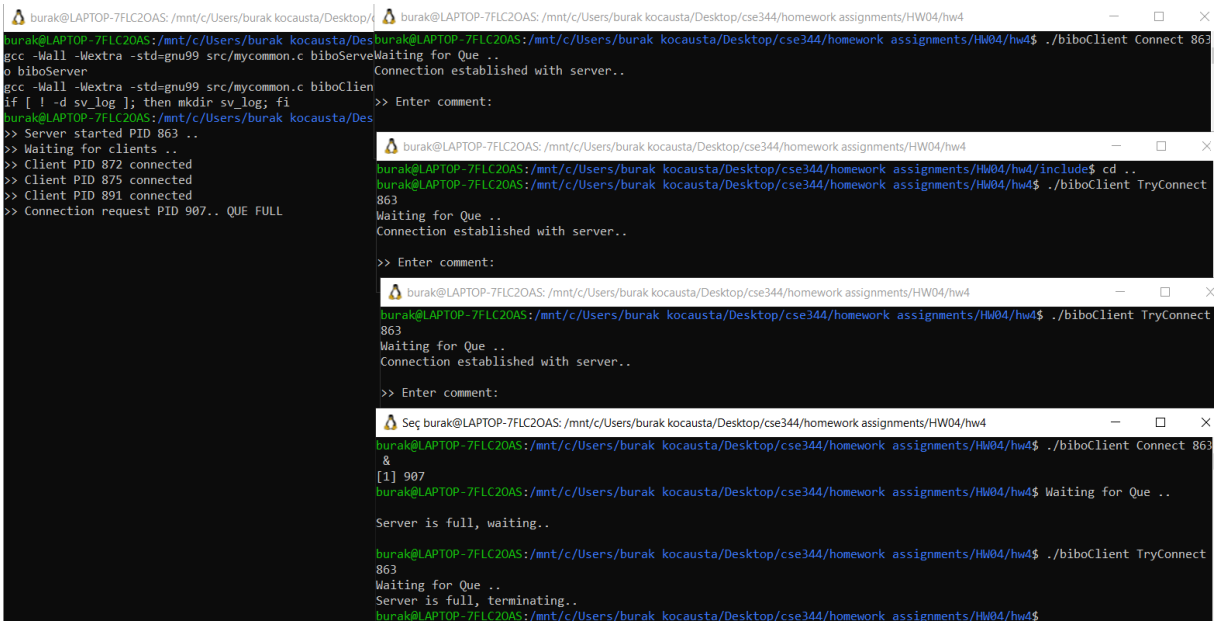
- After compiling log directory will be created.

Testing client connection, and try connect:

```
gcc -Wall -Wextra -std=gnu99 src/mycommon.c biboServer.c src/bserver.c src/task_queue.c src/thread_pool.c src/command_impl.c -lrt -pthread -
o biboServer
gcc -Wall -Wextra -std=gnu99 src/mycommon.c biboClient.c -lrt -pthread -o biboClient
if [ ! -d sv_log ]; then mkdir sv_log; fi
burak@LAPTOP-7FLC2OAS:/mnt/c/Users/burak kocausta/Desktop/cse344/homework assignments/HW04/hw4$ ./biboServer Servers 3 5
>> Server started PID 863 ..
>> Waiting for clients ..
```

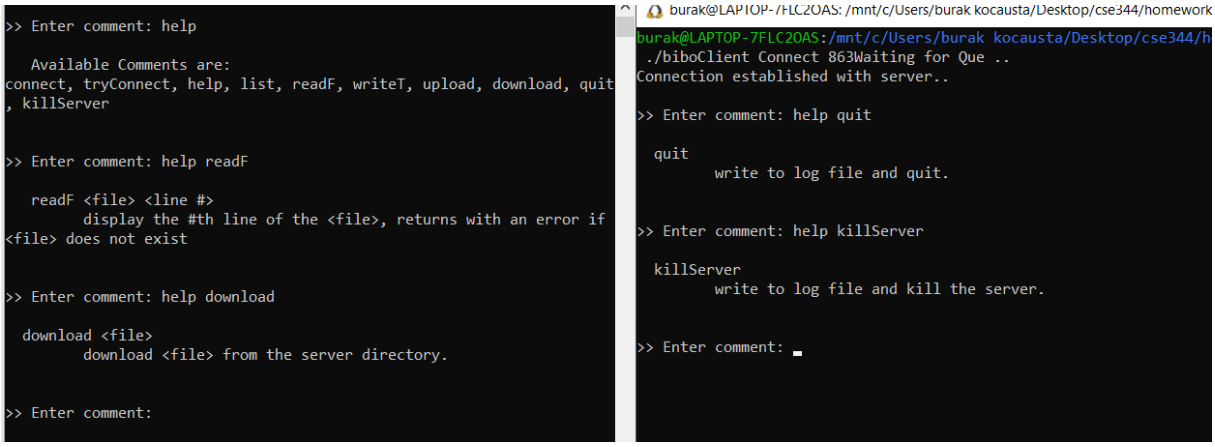
```
burak@LAPTOP-7FLC2OAS:/mnt/c/Users/burak kocausta/Desktop/cse344/homework assignments/HW04/hw4/include$ ps aux -L
USER        PID   LWP %CPU NLWP %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1     1  0.0    2  0.0   1120   676 ?        Sl   12:21   0:00 /init
root          1     6  0.0    2  0.0   1120   676 ?        Sl   12:21   0:00 /init
root          7     7  0.0    1  0.0    908    84 ?        Ss   12:21   0:00 /init
root          8     8  0.0    1  0.0    908    84 ?        S    12:21   0:00 /init
burak         9     9  0.0    1  0.0  10168  5020 pts/0    Ss   12:21   0:00 -bash
root         22    22  0.0    1  0.0    908    84 ?        Ss   12:21   0:00 /init
root         23    23  0.0    1  0.0    908    84 ?        S    12:21   0:00 /init
burak        24    24  0.0    1  0.0  10684  5632 pts/1    Ss+  12:21   0:00 -bash
root        676   676  0.0    1  0.0    908    84 ?        Ss   20:21   0:00 /init
root        677   677  0.0    1  0.0    908    84 ?        S    20:21   0:00 /init
burak       678   678  0.0    1  0.0  10168  5124 pts/2    Ss   20:21   0:00 -bash
burak       863   863  0.0    6  0.0  43692   756 pts/0    Sl+  21:28   0:00 ./biboServer Servers 3 5
burak       863   864  0.0    6  0.0  43692   756 pts/0    Sl+  21:28   0:00 ./biboServer Servers 3 5
burak       863   865  0.0    6  0.0  43692   756 pts/0    Sl+  21:28   0:00 ./biboServer Servers 3 5
burak       863   866  0.0    6  0.0  43692   756 pts/0    Sl+  21:28   0:00 ./biboServer Servers 3 5
burak       863   867  0.0    6  0.0  43692   756 pts/0    Sl+  21:28   0:00 ./biboServer Servers 3 5
burak       863   868  0.0    6  0.0  43692   756 pts/0    Sl+  21:28   0:00 ./biboServer Servers 3 5
burak       869   869  0.0    1  0.0  10616  3288 pts/2    R+   21:29   0:00 ps aux -L
```

- After process is started, threads are initalized, and waiting for commands.



- Clients are connected to server, after size is filled, one process is waiting, and after tryconnect request to server one process is terminated.

Testing help:

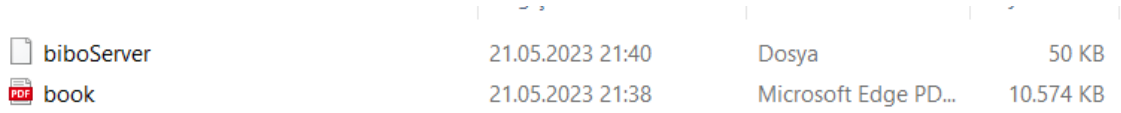


- help command and, help with arguments.

Testing upload:

```
>> Enter comment: upload book.pdf
> Sending upload request to the server..
> Writing file to shared memory..
10827197 number of bytes is loaded on shared memory
> File written to shared memory..
> Upload successful..

>> Enter comment:
```

```
>> Enter comment: upload biboServer
> Sending upload request to the server..
> Writing file to shared memory..
51072 number of bytes is loaded on shared memory
> File written to shared memory..
> Upload successful..
```



- Server directory. They are uploaded successfully.

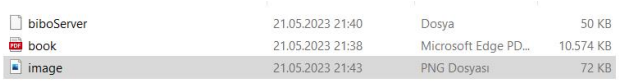
Testing list:

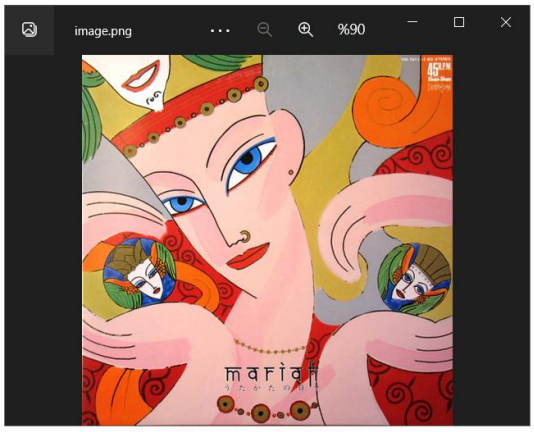
```
>> Enter comment: list
biboServer
book.pdf
```

```
>> Enter comment: upload image.png
> Sending upload request to the server..
> Writing file to shared memory..
72869 number of bytes is loaded on shared memory
> File written to shared memory..
> Upload successful..

>> Enter comment:
```

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

>> Enter comment:
```

- Upload another file which is png file, after calling list it can be seen.





- It is successfully uploaded.

Testing readF:

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

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

- Upload file1.txt, and view it.

```
>> Enter comment: readF file1.txt 3
gapgamfw
>> Enter comment: readF file1.txt 7
awgpawgaopgnpawga
>> Enter comment: readF book.pdf 50
endobj
```

- reads lines correctly.

Testing writeT:

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

>> Enter comment: writeT file1.txt 5 "5th line is here"

>> Enter comment: readF file1.txt
fagawfawawfgaw
wagawgonwgpa
gapgamfw
aa aa aa aa
5th line is here
waawfwafeaw
awgpawgaopgnpawga
awgoawgnoawgpap

>> Enter comment:
```

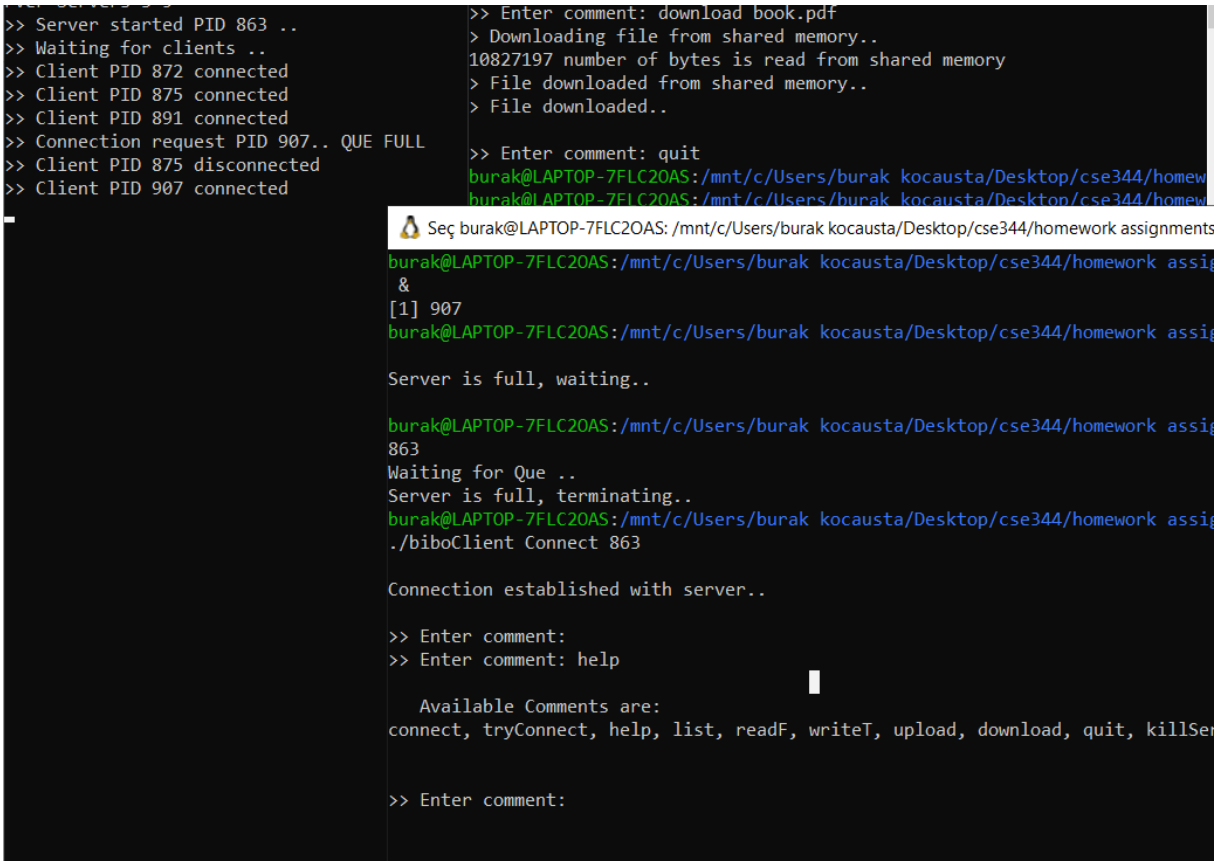
- It successfully replaced.

Testing download:

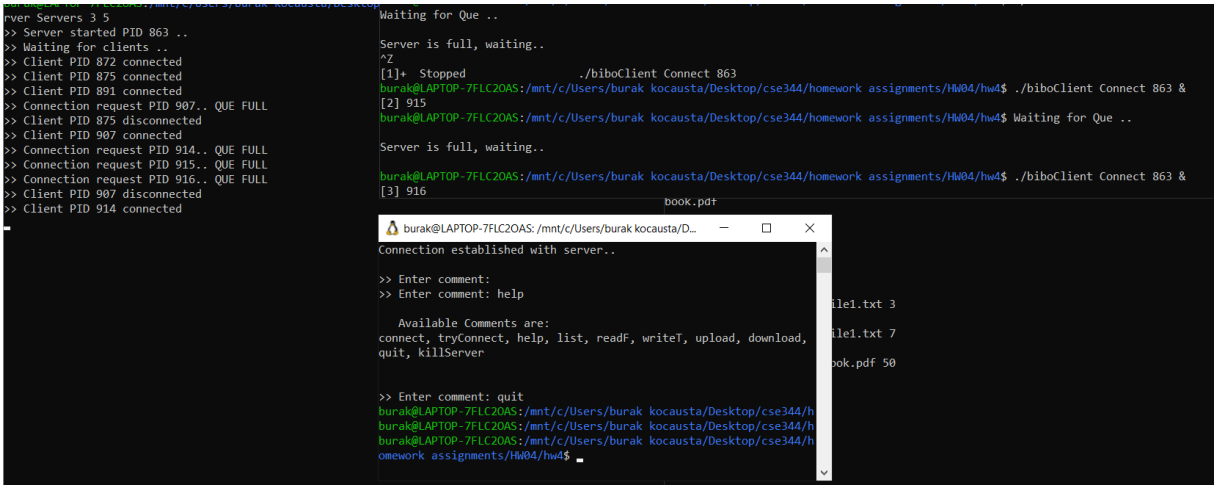
```
>> Enter comment: download book.pdf
> Downloading file from shared memory..
10827197 number of bytes is read from shared memory
> File downloaded from shared memory..
> File downloaded..
```

- download output, it is downloaded successfully.

Testing quit:

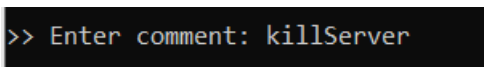


- One of the client sends quit command it disconnects, and one of the waiting clients connects.



- More clients try to connect, and added to waiting list. One of the clients quits, and one of them is connects instead of exited client.

Testing killServer:



```
./biboClient Connect 863
Connection established with server..

>> Enter comment: > SIGINT received. Terminating..
> SIGINT received. Terminating..
> SIGINT received. Terminating..
[2]   Done                    ./biboClient Connect 863
[3]+  Done                    ./biboClient Connect 863
```

Seç burak@LAPTOP-7FLC2OAS: /mnt/c/Users/burak kocausta/Desktop/cse344/homework a

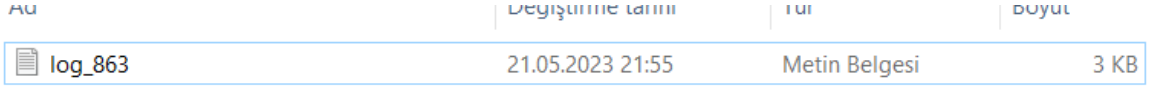
```
863
Waiting for Que ..
Connection established with server..

>> Enter comment: > SIGINT received. Terminating..
```

- Running and waiting clients are terminated.

```
>> Server started PID 863 ..
>> Waiting for clients ..
>> Client PID 872 connected
>> Client PID 875 connected
>> Client PID 891 connected
>> Connection request PID 907.. QUE FULL
>> Client PID 875 disconnected
>> Client PID 907 connected
>> Connection request PID 914.. QUE FULL
>> Connection request PID 915.. QUE FULL
>> Connection request PID 916.. QUE FULL
>> Client PID 907 disconnected
>> Client PID 914 connected
>> Kill signal from client PID 891, terminating..
>> bye..
burak@LAPTOP-7FLC2OAS:/mnt/c/Users/burak kocausta/Desktop/cse344/homework assignments/HW04/hw4$ ps aux -L
USER       PID   LWP %CPU NLWP %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1     1  0.0    2  0.0   1120   660 ?        Sl   12:21   0:00 /init
root         1     6  0.0    2  0.0   1120   660 ?        Sl   12:21   0:00 /init
root         7     7  0.0    1  0.0    908    84 ?        Ss   12:21   0:00 /init
root         8     8  0.0    1  0.0    908    84 ?        R    12:21   0:00 /init
burak        9     9  0.0    1  0.0  10168  5020 pts/0    Ss   12:21   0:00 -bash
root        22    22  0.0    1  0.0    908    84 ?        Ss   12:21   0:00 /init
root        23    23  0.0    1  0.0    908    84 ?        S    12:21   0:00 /init
burak       24    24  0.0    1  0.0  10684  5632 pts/1    Ss+  12:21   0:00 -bash
root       676   676  0.0    1  0.0    908    84 ?        Ss   20:21   0:00 /init
root       677   677  0.0    1  0.0    908    84 ?        S    20:21   0:00 /init
burak      678   678  0.0    1  0.0  10684  5656 pts/2    Ss+  20:21   0:00 -bash
root       876   876  0.0    1  0.0   1120   240 ?        Ss   21:32   0:00 /init
root       877   877  0.0    1  0.0   1120   240 ?        S    21:32   0:00 /init
burak      878   878  0.0    1  0.0  10168  5096 pts/3    Ss+  21:32   0:00 -bash
root       892   892  0.0    1  0.0   1120   240 ?        Ss   21:32   0:00 /init
root       893   893  0.0    1  0.0   1120   240 ?        S    21:32   0:00 /init
burak      894   894  0.0    1  0.0  10168  5036 pts/4    Ss+  21:32   0:00 -bash
burak      917   917  0.0    1  0.0  10616  3256 pts/0    R+   21:56   0:00 ps aux -L
burak@LAPTOP-7FLC2OAS:/mnt/c/Users/burak kocausta/Desktop/cse344/homework assignments/HW04/hw4$
```

- Server is terminated. There aren't any zombie or orphan processes. All threads are terminated.

Testing Log:



- Inside sv_log directory.

Contents of log file:

```
MAIN SERVER: >> Server started PID 863 ..
MAIN SERVER: 863 >> Client PID 872 connected
MAIN SERVER: 863 >> Client PID 875 connected
MAIN SERVER: 863 >> Client PID 891 connected
MAIN SERVER: 863 >> Connection request PID 907.. QUE FULL
SERVER THREAD: >> Thread for client 875 is activated
SERVER THREAD: >> help command is executed for client 875
SERVER THREAD: >> Thread for client 875 is activated
SERVER THREAD: >> help command is executed for client 875
SERVER THREAD: >> Thread for client 875 is activated
SERVER THREAD: >> help command is executed for client 875
SERVER THREAD: >> Thread for client 891 is activated
SERVER THREAD: >> help command is executed for client 891
SERVER THREAD: >> Thread for client 891 is activated
SERVER THREAD: >> help command is executed for client 891
SERVER THREAD: >> Thread for client 891 is activated
SERVER THREAD: >> upload command is executed for client 891
SERVER THREAD: >> Thread for client 875 is activated
SERVER THREAD: >> upload command is executed for client 875
SERVER THREAD: >> Thread for client 891 is activated
SERVER THREAD: >> list command is executed for client 891
SERVER THREAD: >> Thread for client 875 is activated
SERVER THREAD: >> upload command is executed for client 875
SERVER THREAD: >> Thread for client 891 is activated
SERVER THREAD: >> list command is executed for client 891
SERVER THREAD: >> Thread for client 875 is activated
SERVER THREAD: >> upload command is executed for client 875
SERVER THREAD: >> Thread for client 875 is activated
SERVER THREAD: >> readF command is executed for client 875
SERVER THREAD: >> Thread for client 891 is activated
SERVER THREAD: >> readF command is executed for client 891
SERVER THREAD: >> Thread for client 891 is activated
SERVER THREAD: >> readF command is executed for client 891
SERVER THREAD: >> Thread for client 891 is activated
SERVER THREAD: >> readF command is executed for client 891
SERVER THREAD: >> Thread for client 875 is activated
SERVER THREAD: >> writeT command is executed for client 875
SERVER THREAD: >> Thread for client 875 is activated
SERVER THREAD: >> readF command is executed for client 875
SERVER THREAD: >> Thread for client 875 is activated
SERVER THREAD: >> download command is executed for client 875
MAIN SERVER: 863 >> Client PID 875 disconnected
MAIN SERVER: 863 >> Client PID 907 connected
SERVER THREAD: >> Thread for client 907 is activated
SERVER THREAD: >> help command is executed for client 907
MAIN SERVER: 863 >> Connection request PID 914.. QUE FULL
MAIN SERVER: 863 >> Connection request PID 915.. QUE FULL
MAIN SERVER: 863 >> Connection request PID 916.. QUE FULL
MAIN SERVER: 863 >> Client PID 907 disconnected
MAIN SERVER: 863 >> Client PID 914 connected
MAIN SERVER: 863 >> Kill signal from client PID 891, terminating..
```

Other:

- I tested with other cases like, uploading a large file, during that upload another client tries to read it, or write a line to it etc.. For the client part it can be tested like more than one client try to download a file, or one client tries to download same file, other one tries to upload that file. Mutual exclusion, and prevention of data race worked correctly for those cases.