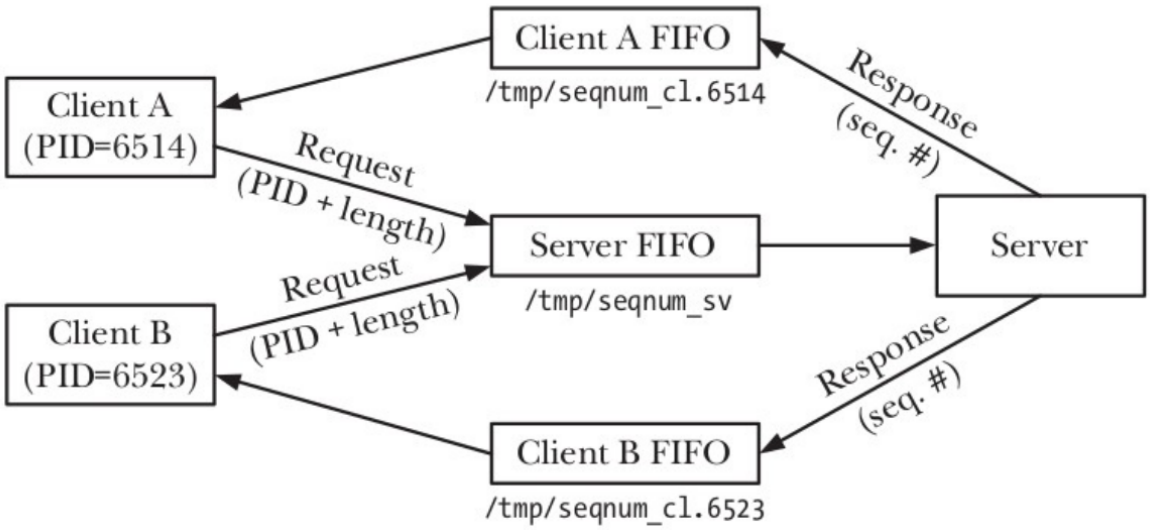
CSE 344
HW4
REPORT

Ömer Faruk SAYAR
171044038

OverAll Design

In this project I used the structure of single-server , multiple-client application:



There is a one server FIFO with named server.PID that accepts the connection requests.

```c
typedef struct connectionReq {
    pid_t client_pid;
    int try_flag;
} connectionReq;

typedef struct Request {
    request_type request;
    char filename[256];
    u_int64_t offset;
    char string[8192];
    char client_dir[256];

} Request;
```

My connection requests are consist of client_pid (needed to open client FIFO which is named client.PID) and try_flag for the determine if the request is tryConnect or connect. After the connection established. Program uses Request structure to send its requests.

When the server wake up, it generates a thread pool with the given size. Each thread in the pool starts to wait until a connection request comes to request queue in a condition wait. In the main thread server reads its FIFO in a while-true loop. If there are clients their number is less than max_number then server accepts the connection, otherwise adds request to queue if it is try_flag is 0, else rejects the request.

Server Side:

```c
init_server();

while (1) {
    connectionReq req;
    read(server_fifo_fd, &req, sizeof(connectionReq));

    if (current_clients == max_clients) {
        printf("Connection request PID %d ...  Queue is FULL\n", req.client_pid);
        if (req.try_flag){
            handle_rejected(req.client_pid);
            continue;
        }
    }

    pthread_mutex_lock(&queue_mutex);
    enqueue(&request_queue, req.client_pid);
    pthread_mutex_unlock(&queue_mutex);
    pthread_cond_signal(&queue_not_empty);
}

close_server();
```

```c
void* worker_thread(void* arg) {

    while (1) {

        if (kill_flag)
        {
            pthread_exit(NULL);
        }

        pthread_mutex_lock(&queue_mutex);

        while (is_queue_empty(&request_queue) || current_clients == max_clients) {
            if (kill_flag)
            {
                pthread_mutex_unlock(&queue_mutex);
                pthread_exit(NULL);
            }

            pthread_cond_wait(&queue_not_empty, &queue_mutex);
        }

        pid_t client_pid = dequeue(&request_queue);
        current_clients++;
        pthread_mutex_unlock(&queue_mutex);

        //Connect to client and start to serve
```

To able to provide synchronization between threads when using sharing data (request_queue), server program uses pthread_mutex to access data and pthread_cond variable to check if the data is ready or not in the queue.

Client Side:

```c
// create connection request struct
connectionReq req;
req.client_pid = getpid();
if (strcmp(argv[1], "connect") == 0)
    req.try_flag = 0; // client will wait in the queue

else if (strcmp(argv[1], "tryConnect") == 0)
    req.try_flag = 1; // client will not wait in the queue

else {…

snprintf(client_fifo, sizeof(client_fifo), "/tmp/client.%d", getpid());
if (mkfifo(client_fifo ,0666) == -1) {…

char server_fifo[256];
snprintf(server_fifo, sizeof(server_fifo), "/tmp/server.%d", server_pid);
// send connection request to server
server_fifo_fd = open(server_fifo, O_WRONLY);
if (server_fifo_fd == -1) {
    perror("open server fifo");
    unlink(client_fifo);
    exit(1);
}
if (write(server_fifo_fd, &req, sizeof(connectionReq)) == -1) {
    perror("write");
    close(server_fifo_fd);
    unlink(client_fifo);
    exit(1);
}
```

To be able to keep any number of requests, queue was implemented as linked-list. Queue only keeps client pid as data.
There is also a log file for the server that logs the client requests. To provide any data corruption on the log file I used unnamed semaphore as mutex and server initialize the mutex with pshared arguments as 0 to be able to use between threads.

```c
typedef struct node {
    int data;
    struct node* next;
} node_t;

typedef struct {
    node_t* head;
    node_t* tail;
} queue_t;
```

```c
void log_message(const char *message) {

    time_t now = time(NULL);
    struct tm *tm_info = localtime(&now);
    char timestamp[20];
    strftime(timestamp, 20, "%Y-%m-%d %H:%M:%S", tm_info);
    char log_msg[1024];
    snprintf(log_msg, sizeof(log_msg), "[%s] %s\n", timestamp, message);
    sem_wait(log_mutex);
    write(log_fd, log_msg, strlen(log_msg));
    sem_post(log_mutex);
}
```

After the connection established, client and server threads use the client FIFO respectively. First one side opens the FIFO for writing other side opens for reading then, writing side closes and opens for reading, reading side closes and opens for writing. Before client sends any commands to server it waits the user input.

```c
110     while(running)
111     {
112         // open client fifo for writing
113         client_fifo_fd = open(client_fifo, O_WRONLY | O_CREAT, 0666);
114 >       if (client_fifo_fd == -1) {…
120
121         Request req;
122         printf("Enter a command: ");
123         char input[BUFSIZE];
124         fgets(input, sizeof(input), stdin);
125         fflush(stdin);
126         input[strlen(input) - 1] = '\0';
127         //parse command according to space character
128         char *token = strtok(input, " ");
129 >       if (token == NULL){…
139
140         char* command = malloc(strlen(token) + 1);
141         strcpy(command, token);
142         command[strlen(token)] = '\0';
143
144 >       if (strcmp(command, "help") == 0)…
184
185 >       else if(strcmp(command,"list") == 0){…
230
231 >       else if(strcmp(command,"readF") == 0){…
326
327 >       else if(strcmp(command, "writeT") == 0)…
456
457 >       else if(strcmp(command, "quit") == 0)…
482
483 >       else if(strcmp(command, "killServer") == 0){…
507
508 >       else if(strcmp(command, "download") == 0){…
581
582 >       else if(strcmp(command, "upload")== 0){…
655
656         else
657             printf("Invalid command!\n");
658
```

To provide synchronization between the server’s threads when access to files on the server directory, named semaphores was used to avoid any race condition and data corruption.

Semaphores are named with file names on the server to be able to sure correct semaphore(mutex) is locked and opened. Before the termination of the server program unlink them using the files names on the server.

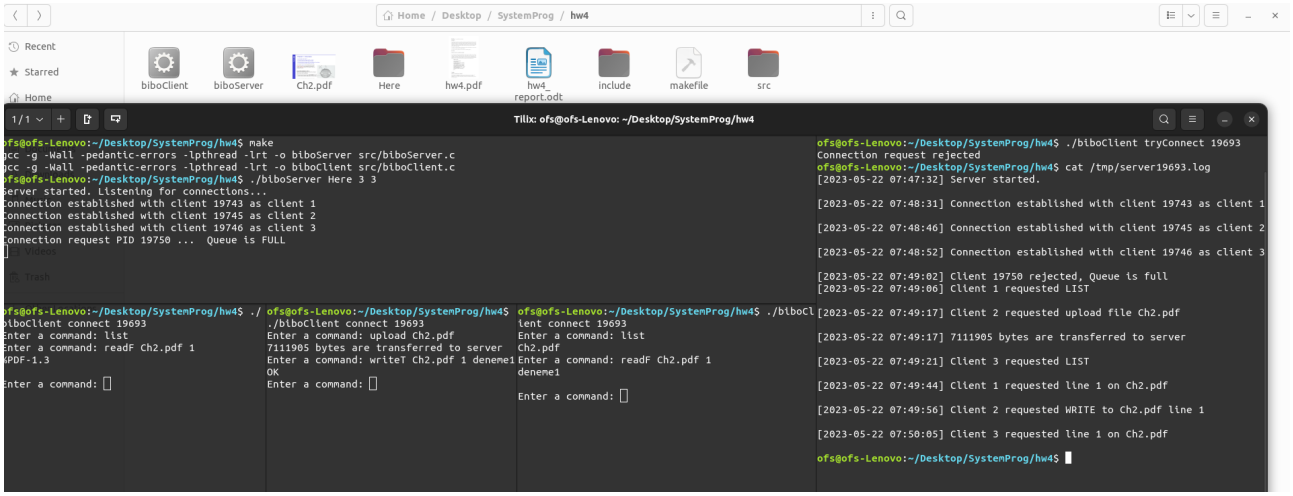
To be able to cleanup threads when client sends killServer commands to server or any SIGINT, SIGTERM, SIGQUIT signal arrives the server, Program keeps the array of the client file descriptors and closes them after cancel the threads in the pool.

```c
void close_server(){

    kill_flag = 1;
    close(server_fifo_fd);
    close(log_fd);
    unlink(server_fifo_path);

    // Cancel threads
    for (int i = 0; i < thread_pool_size; i++) {
        pthread_cancel(thread_pool[i]);
    }

    for (int i = 0; i < max_clients; i++)
        close(client_fds[i]);

    DIR* dir = opendir(working_dir);
    if (dir == NULL) {
        perror("opendir");
        exit(1);
    }
    struct dirent* entry;
    while ((entry = readdir(dir)) != NULL) {
        if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0) {
            continue;
        }
        char semaphore_name[256];
        strcpy(semaphore_name, "/");
        strcat(semaphore_name, entry->d_name);
        sem_unlink(semaphore_name);
    }

    free(client_fds);
    free(thread_pool);
    sem_destroy(log_mutex);
    pthread_mutex_destroy(&queue_mutex);
    pthread_cond_destroy(&queue_cond);
    empty_queue(&request_queue);
    exit(0);
}
```

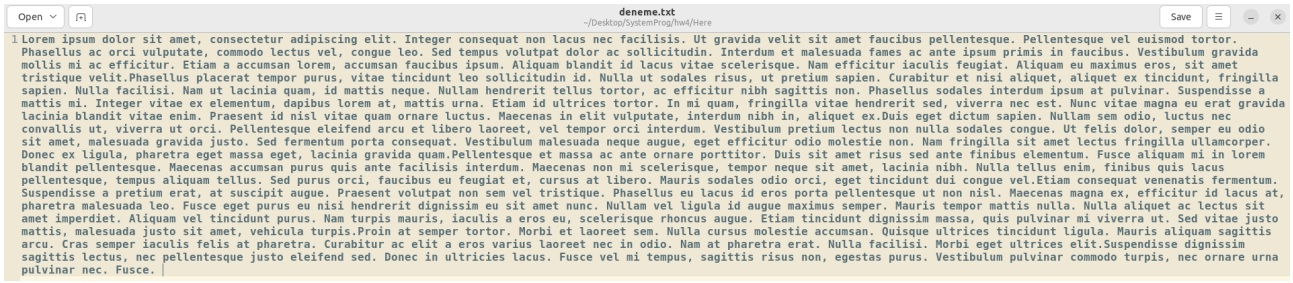
```c
void handle_server_kill(int sig){
    close_server();
}

struct sigaction sigact;
sigact.sa_handler = handle_server_kill;
sigemptyset(&sigact.sa_mask);
sigact.sa_flags = 0;
sigaction(SIGINT, &sigact, NULL);
sigaction(SIGTERM, &sigact, NULL);
sigaction(SIGQUIT, &sigact, NULL);
```

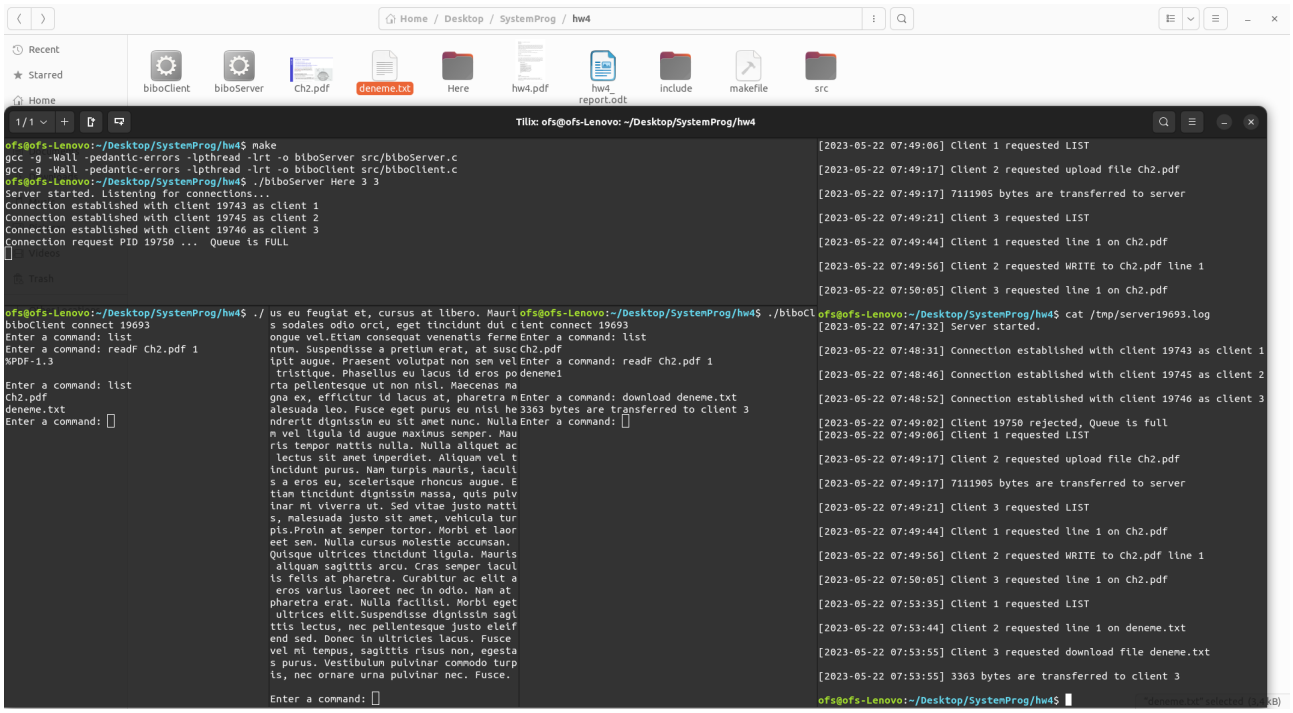
TEST CASES:

1- Open server for existing directory, shows all commands are working.

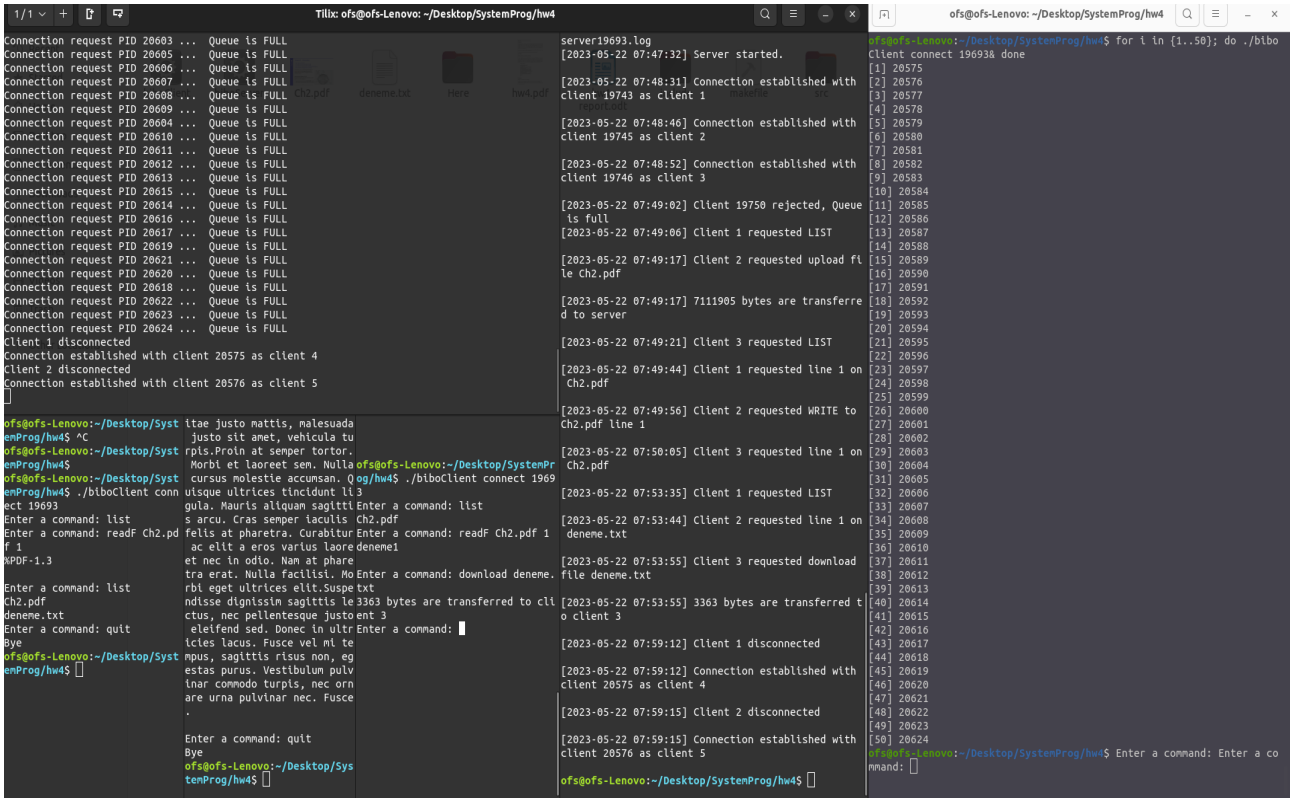


Now adding deneme.txt to Here directory manually to test download option.

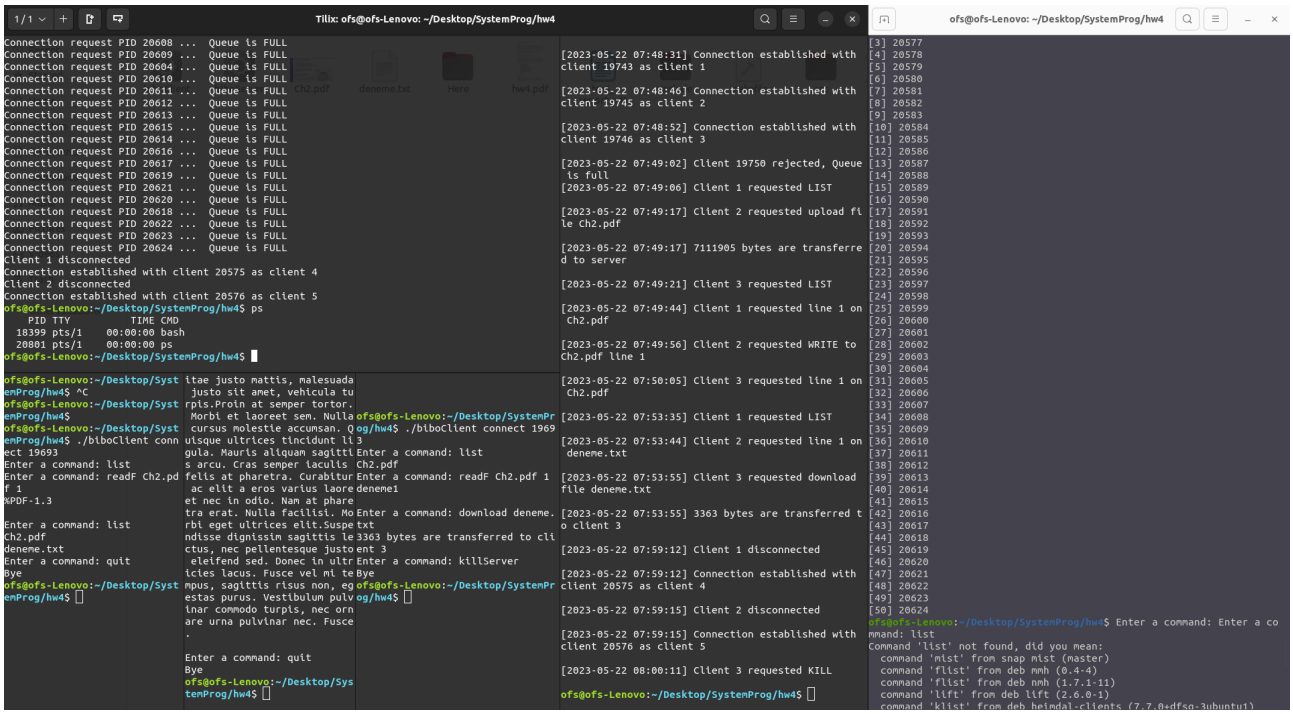




2- Sending 50 connect requests and test if they are in the queue



3-Sending killServer request



4- Invalid file name