Abdurrahman Ahmet Daştan

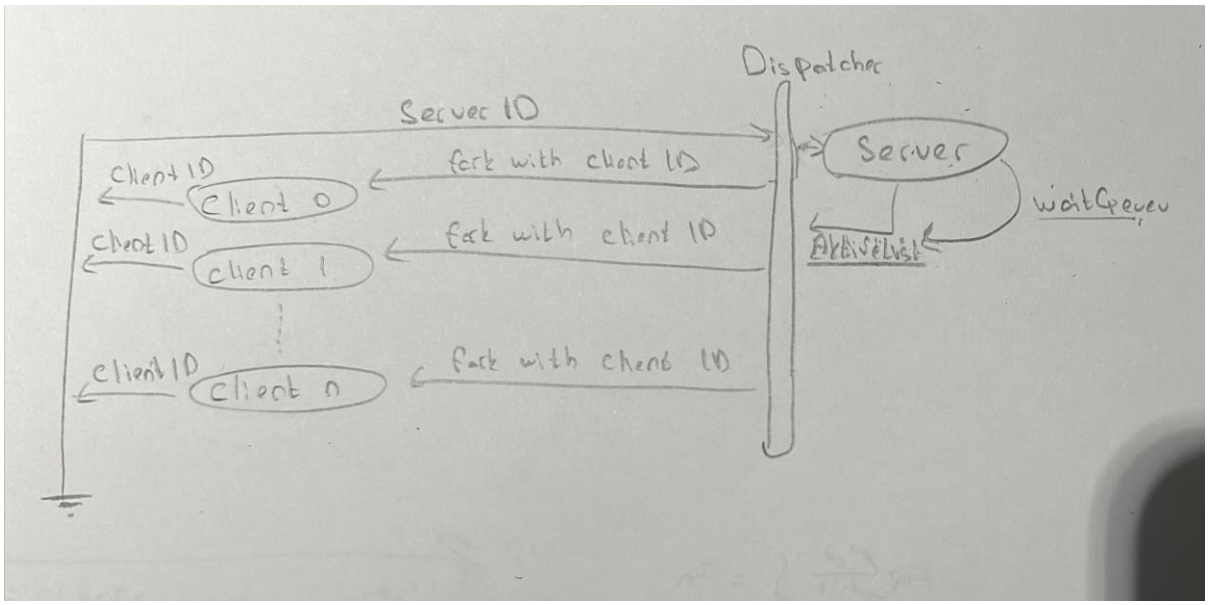
200104004127

CSE 344 MIDTERM

Project description

This project aims to develop server-client relationships with using fork to serve a lot of clients and sheared memory to communicate clients.

Project Design



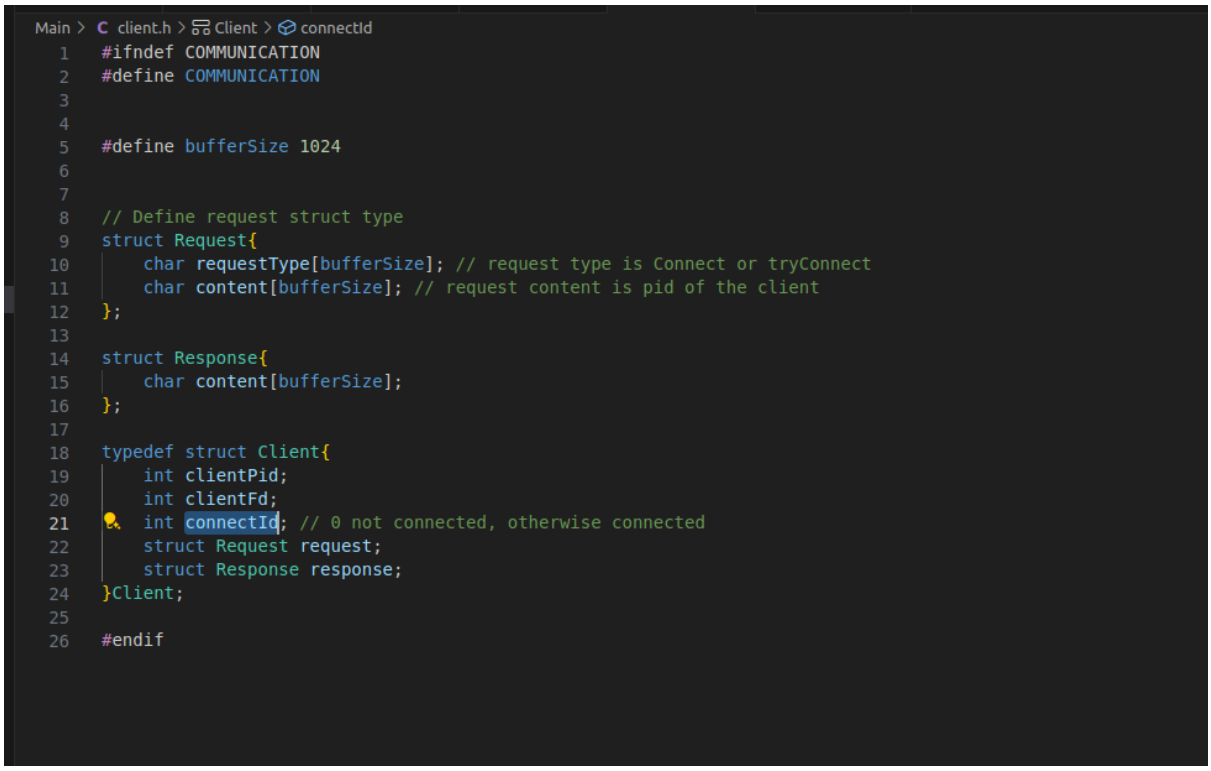
Firstly I design server side and client side with connect each other via Server PID. Wen I create a server in a terminal I take server PID and create a client with that. Server clients write own client PID to Fifo that named server PID. After that if server accept the client, server create a fork. This fork serve to spesific client through client Fifo named client PID.

Therefore server add user activeUsers list. Of course this list has a limit that giving by user when server created. If activeUsers list is full, server add the client to waitQueue lists. (I use list structure instead of queue structure but its work same as queue.) I use list because client who stayed in the middle of queue can be kill itself. If it is does, we cannot erase from queue. So logic is queue but structure is list.

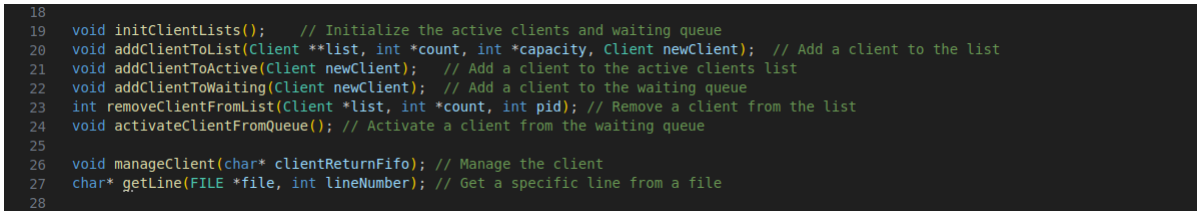
My implementation

There are some screen shot for server and clients for their communication.

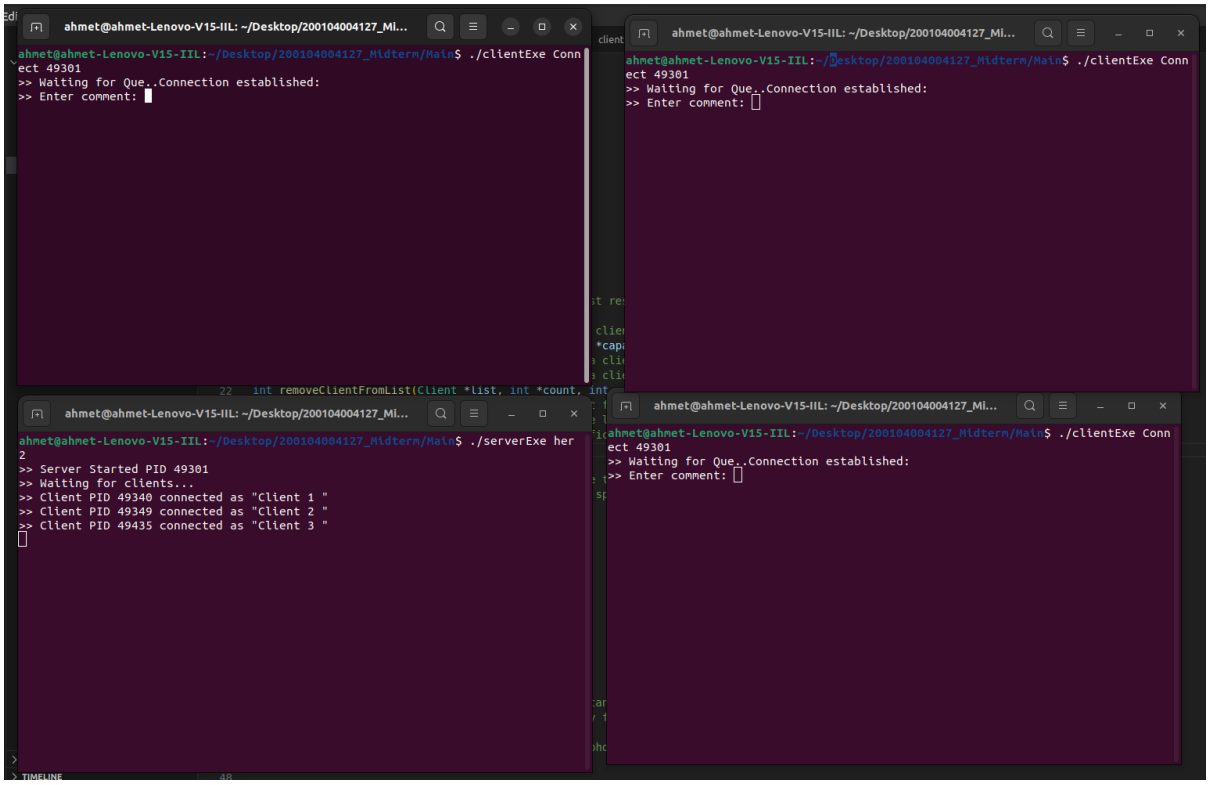
Client.h; this header file show my design basically. So I share this screenshot.



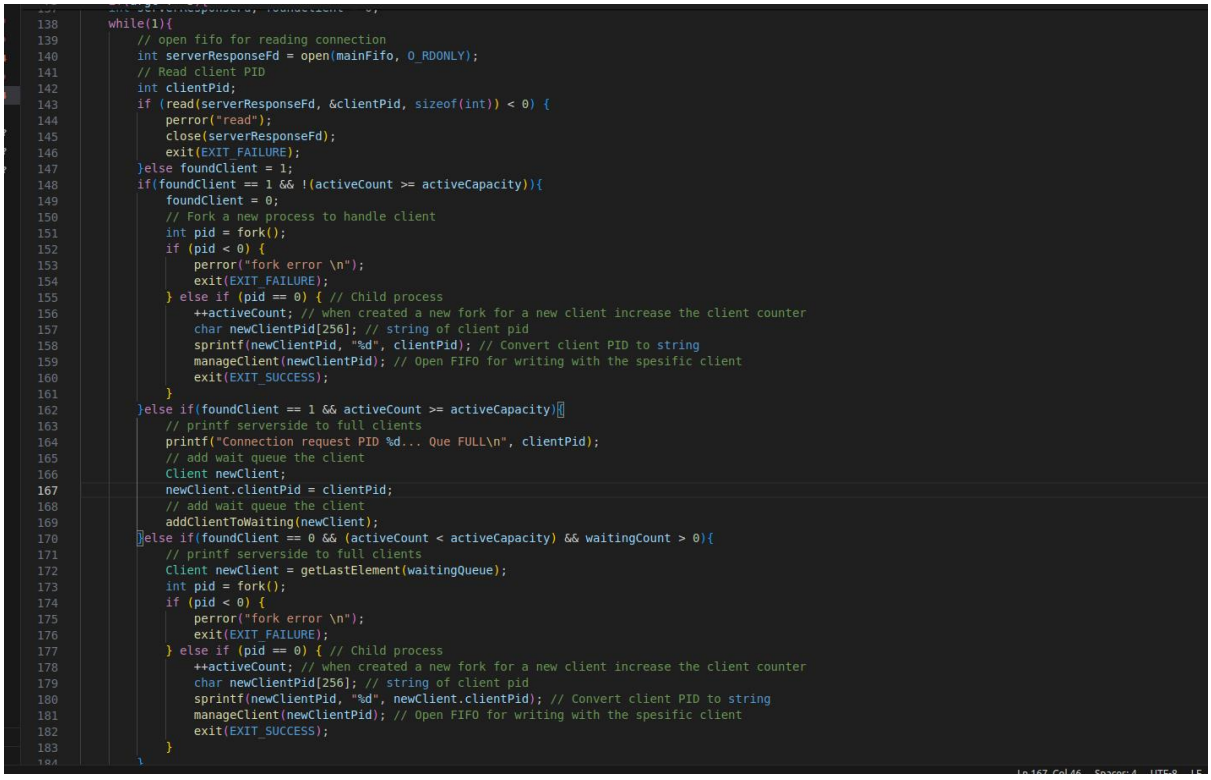
activeUser and waitQueue structure; User can be inside of the activeUser list or waitQueue. If conditions are available to adding new client to server, it add activeUser list. Otherwise client should be wait in waitQueue. There are headers of the functions to demonstrate struct of the design.



Here are a lot user connect same server simultaneously;



This server side main loop; read server fifo and if detected clients try to add active users otherwise add wait queue.



```
while(1){
    // open fifo for reading connection
    int serverResponseFd = open(mainFifo, O_RDONLY);
    // Read client PID
    int clientPid;
    if (read(serverResponseFd, &clientPid, sizeof(int)) < 0) {
        perror("read");
        close(serverResponseFd);
        exit(EXIT_FAILURE);
    }else foundClient = 1;
    if(foundClient == 1 && !(activeCount >= activeCapacity)){
        foundClient = 0;
        // Fork a new process to handle client
        int pid = fork();
        if (pid < 0) {
            perror("fork error \n");
            exit(EXIT_FAILURE);
        } else if (pid == 0) { // Child process
            ++activeCount; // when created a new fork for a new client increase the client counter
            char newClientPid[256]; // string of client pid
            sprintf(newClientPid, "%d", clientPid); // Convert client PID to string
            manageClient(newClientPid); // Open FIFO for writing with the spesific client
            exit(EXIT_SUCCESS);
        }
    }else if(foundClient == 1 && activeCount >= activeCapacity){
        // printf serverside to full clients
        printf("Connection request PID %d... Que FULL\n", clientPid);
        // add wait queue the client
        Client newClient;
        newClient.clientPid = clientPid;
        // add wait queue the client
        addClientToWaiting(newClient);
    }else if(foundClient == 0 && (activeCount < activeCapacity) && waitingCount > 0){
        // printf serverside to full clients
        Client newClient = getLastElement(waitingQueue);
        int pid = fork();
        if (pid < 0) {
            perror("fork error \n");
            exit(EXIT_FAILURE);
        } else if (pid == 0) { // Child process
            ++activeCount; // when created a new fork for a new client increase the client counter
            char newClientPid[256]; // string of client pid
            sprintf(newClientPid, "%d", newClient.clientPid); // Convert client PID to string
            manageClient(newClientPid); // Open FIFO for writing with the spesific client
            exit(EXIT_SUCCESS);
        }
    }
```

This client main loop; after connecting server it wait accepted response. When client take this response it has 2 operation these sending request and taking response.

Client doesn’t know anything about management of server side. It is just use server properties and server’s allows.

```
    char request[1024], response[1024];
    while(1){
        // Read request from user
        memset(request, 0, sizeof(request));  // clear the tempMssg
        printf(">> Enter comment: ");
        fgets(request, sizeof(request), stdin);

        requestFd = open(clientFifoName, O_WRONLY); // open client fifo for request from server
        if(requestFd < 0){
            perror("client open fifo failed");
            return -1;
        }
        // Send request to server
        if (write(requestFd, &request, sizeof(request)) == -1){
            perror("Server requesting error.");
            return -1;
        }
        close(requestFd);

        // Read response from server
        responseFd = open(clientFifoName, O_RDONLY); // open client fifo for reading response from server
        if(responseFd < 0){
            perror("client open fifo failed");
            return -1;
        }
        // Read response from server for ensure connection is okay
        while (read(responseFd, &response, sizeof(response)) > 0){
            printf("%s\n", client.response.content);
            memset(client.response.content, 0, sizeof(client.response.content));  // clear the tempMssg
        }
        close(responseFd);
    }
    unlink(clientFifoName);
```

Semaphores and shared memory; We use a lot of clients in the server. Therefore they are use server properties commonly. In addition for their management server fork should know their states(active or not). Therefore we use same memory places for some variables. (all process has own memory address spaces)

Of course, this situation cause a race condition. For solving them we combine shared memory with semaphores. I used activeUserCounter in server side. When user in there I increase this value, otherwise decrease. For doing that I create semaphore and below some screenshot to show that.

```

///// shared memory Design with semaphoers/////////
    int shm_fd = shm_open(counterSharedMemory, O_CREAT | O_RDWR, 0666);  // define shared memory file descriptor
    if (shm_fd == -1) {
        perror("shm_open");
        return EXIT_FAILURE;
    }
    if (ftruncate(shm_fd, sizeof(int)) == -1) {  // set size of shared memory
        perror("ftruncate");
        return EXIT_FAILURE;
    }
    sharedSequentialClientCounter = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);  // map shared memory
    if (sharedSequentialClientCounter == MAP_FAILED) {
        perror("mmap");
        return EXIT_FAILURE;
    }
    *sharedSequentialClientCounter = 0;  // initialize shared memory
    // create semaphore
    sem = sem_open(semaphoreName, O_CREAT, 0666, 1);
    if (sem == SEM_FAILED) {
        perror("sem_open");
        return EXIT_FAILURE;
    }
///// shared memory Design with semaphoers/////////
```

```
41  ///////// shared memory Design with semaphoers Constants/////////
42  int *sharedSequentialClientCounter; // shared memory for sequentialClientCounter
43
44  const char *semaphoreName = "mysemaphore";  // semaphore name
45  sem_t *sem; // define semaphore
46
47  const char *counterSharedMemory = "/clientCounter";  // shared memory name
48  ///////// shared memory Design with semaphoers Constants/////////
49
50  void handleInterruptSignal(int sig) {
51      printf("SIGTSTP received, unlinking FIFO...\n");
52
53      // Unlink the FIFO
54      if (unlink(mainFifo) == -1) { // if already unlinked, it will return -1
55          perror("Failed to unlink FIFO");
56      }
57      // kill clients and child process
58      for (int i = 0; i < activeCount; i++) {
59          kill(activeClients[i].clientPid, SIGKILL);
60      }
61      // release shared memory
62      munmap(sharedSequentialClientCounter, sizeof(int));
63      shm_unlink(counterSharedMemory);
64
65      // release semaphores
66      sem_close(sem);
67      sem_unlink(semaphoreName);
68
69      kill(getpid(), SIGKILL); // kill the server process
70  }
```

Here are definition of semaphores **with signal interrupt**; Signal interrupts have critical role because when client forcely quit terminal server should know that.

When server send a kill request to **kill** server, server should kill whole child process (clients), otherwise they are hang. Also this is cause optimization problems. Client signal shown below, too.

```
15  void handleInterruptSignal(int sig) {
16      printf("SIGTSTP received, unlinking FIFO...\n");
17
18      // send quit signal to server for killing the client serverside
19      int requestFd = open(clientFifoName, O_WRONLY); // open client fifo for request from server
20      if(requestFd < 0){
21          perror("In signal client open fifo failed");
22          return;
23      }
24      // Send request to server
25      if (write(requestFd, "quit", sizeof("quit")) == -1){
26          perror("Server requesting error.");
27          return;
28      }
29      close(requestFd);
30
31      // Unlink the FIFO
32      if (unlink(clientFifoName) == -1) { // if already unlinked, it will return -1
33          perror("Failed to unlink FIFO");
34      }
35      printf("killed cleint");
36      kill(getpid(), SIGKILL); // kill the client process
37      exit(EXIT_SUCCESS);
38  }
39
```

Some utilities

Server have a lot of utilities like list directories, help about properties, upload/download, quit and kill server etc.

Here are example of **help and list** request code block and their terminal results;

Help Codeblock;

```c
    // - help; display the list of possible client request
    if (strcmp(newReq[0], "help") == 0) {
        // Help request
        if(newReq[1] == NULL){
            snprintf(tempMssg, sizeof(tempMssg),"Available comments are :\nhelp, list, readF, writeT, upload, download, archServer, quit, killServer");
            printf("burasi temp mesaj %s\n", tempMssg);
        }else if(strcmp(newReq[1], "readF") == 0){
            printf("anans");
            snprintf(tempMssg, sizeof(tempMssg), "readF <file> <line # \n> requests to display the # line of the <file>, if no line number is given \n the
            whole contents of the file is requested (and displayed on the client side)");
        }else if(strcmp(newReq[1], "writeT") == 0){
            strcpy(tempMssg,"writeT <file> <line #> <string> \n : request to write the content of “string” to the #th line the <file>, if the line # is not
            given \n writes to the end of file. If the file does not exists in Servers directory creates and edits the\n file at the same time");
        }else if(strcmp(newReq[1], "upload") == 0){
            strcpy(tempMssg,"upload <file> \nuploads the file from the current working directory of client to the Servers directory \n(beware of the cases
            no file in clients current working directory and file with the same\n name on Servers side)");
        }else if(strcmp(newReq[1], "download") == 0){
            strcpy(tempMssg,"download <file> \nrequest to receive <file> from Servers directory to client side");
        }else if(strcmp(newReq[1], "archServer") == 0){
            strcpy(tempMssg,"archServer <fileName>.tar \n Using fork, exec and tar utilities create a child process that will collect all the files
            currently \navailable on the the Server side and store them in the <filename>.tar archive");
        }else if(newReq[1] == "killServer"){
            strcpy(tempMssg,"killServer \nSends a kill request to the Server");
        }else if(newReq[1] == "quit"){
            strcpy(tempMssg,"quit \nSend write request to Server side log file and quits");
        }

        int clientFd = open(clientReturnFifo, O_WRONLY);
        if (clientFd < 0) {
            perror("server open fifo failed");
            exit(EXIT_FAILURE);
        }
        if (write(clientFd, &tempMssg, sizeof(tempMssg)) < 0) {
            perror("server write failed");
            exit(EXIT_FAILURE);
        }
        close(clientFd);
```

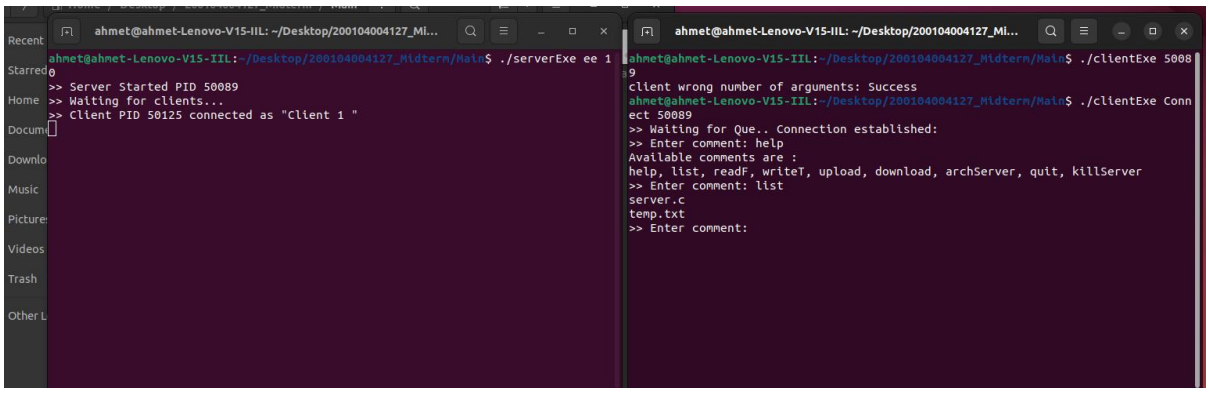
List Codeblock;

```c
        close(clientFd);
    }else if(strcmp(newReq[0], "list") == 0){
        // list all server directory elemetns
        DIR *dir;
        struct dirent *ent;
        int clientFd = open(clientReturnFifo, O_WRONLY, 0666);
        if (clientFd < 0) {
            perror("server open fifo failed");
            exit(EXIT_FAILURE);
        }
        // Open the current directory
        if ((dir = opendir("./server")) != NULL) {

            // Iterate over entries in the directory
            while ((ent = readdir(dir)) != NULL) {

                //snprintf(tempMssg, sizeof(tempMssg), "%s\n", ent->d_name);   // Send serponse to client
                if (strcmp(ent->d_name, ".") == 0 || strcmp(ent->d_name, "..") == 0) {
                    continue;
                }
                snprintf(tempMssg, sizeof(tempMssg), "%s", ent->d_name);
                if (write(clientFd, tempMssg, sizeof(tempMssg)) < 0) {
                    perror("server write failed");
                    close(clientFd);
                    closedir(dir);
                    exit(EXIT_FAILURE);
                }
                memset(tempMssg, 0, sizeof(tempMssg));  // clear the tempMssg
            }
        } else {
            perror("");
            return;
        }
        close(clientFd);
        closedir(dir);
    }else if(strcmp(newReq[0], "readF") == 0){
```

Their Terminal Results;



Release, free and kill; These process save us to leak memory

```c

    // release shared memory
    munmap(sharedSequentialClientCounter, sizeof(int));
    shm_unlink(counterSharedMemory);

    // release semaphores
    sem_close(sem);
    sem_unlink(semaphoreName);

    // Close the FIFO
    close(serverResponseFd);
    unlink(mainFifo);
    // Free the allocated memory
    free(activeClients);
    free(waitingQueue);

    kill(getpid(), SIGKILL); // kill the server process
    return 0;
}

```

For managing whole operation in a same root directory I create **makefile.**