

CSE 344
System Programming
Midterm Project
Report



Author

Mert Emir ŞEKER

200104004085

Date

05.05.2024

Table Of Contents

1. Makefiles	4
2. Makefile Commands	4
3. How to run the code?	4
4. Code Explanation	5
4.1. server.c	5
4.1.1. signal_handler	5
4.1.2. setup_signal_handler	5
4.1.3. semaphore manipulation	6
4.1.4. setup_server	6
4.1.5. managing connections and assigning clients	7
4.1.6. list	8
4.1.7. readF	8
4.1.8. writeT	9
4.1.9. download	10
4.1.10. upload	10
4.1.11. killServer	11
4.1.12. quit	11
4.1.13. help	12
4.1.14. error_exit	12
4.1.15. archServer	13
4.2. client.c	14
4.2.1. semaphore manipulation	14
4.2.2. Semaphore Initialization for Client-Server Synchronization.....	14
4.2.3. open_fifo	15
4.2.4. Client Identification and Initial Communication Setup	15
4.2.5. handle_client	16
5. Example Outputs	17
5.1. help	17
5.2. list	18
5.3. readF	18
5.4. writeT	19

5.5. download	20
5.6. upload	20
5.7. archServer	21
5.8. quit	21
5.9. killServer	22
5.10. Ctrl-C	22

1. Makefiles:

```
M makefile X
C: > Users > merts > OneDrive > Masaüstü > ss > client > M makefile
1  all: compile
2
3  compile:
4      gcc -o nehosClient client.c
5
6  clean:
7      rm -f *.o
8      rm -f nehosClient
9      clear
10
```

```
M makefile C:\...\server X M makefile C:\...\client
C: > Users > merts > OneDrive > Masaüstü > ss > server > M makefile
1  all: compile
2
3  compile:
4      gcc -o neHosServer server.c
5
6  clean:
7      rm -f *.o
8      rm -f neHosServer
9      clear
10
```

2. Makefile Commands:

All: Compiles the code.

Compile: Compiles the code.

Clean: Clears terminal.

3. How to run the code?

To run the program, you first need to run the server in a separate terminal. After **make all**, we start the server as **./neHosServer filename #ofclients**. Then, after **make all** for the client from another terminal, we run our program by connecting to the server as **./nehosClient connect/tryConnect server_pid**.

4. Code Explanation:

4.1. Server.c:

4.1.1. signal_handler: The function `signal_handler` is a signal handler that is triggered when it receives `SIGINT`, usually from an input of `Ctrl-C`. Cleaning is done by producing a notification message, erasing the server's FIFO (using `unlink(server_fifo)`), looping over a client PID array (`pid_array`), destroying the unique FIFO for each client, and gently terminating each active client by issuing a `SIGTERM`. To ensure a smooth shutdown, the function ends by leaving the application with a status code of 0.

```
void signal_handler(int sig)
{
    printf("Ctrl-C received, shutting down...\n");

    unlink(server_fifo);
    for (int i = 0; i < number_of_clients; i++)
    {
        if (pid_array[i] != 0)
        {
            char client_fifo[256];
            sprintf(client_fifo, "/tmp/%d", pid_array[i]);
            unlink(client_fifo);
            kill(pid_array[i], SIGTERM);
        }
    }

    exit(0);
}
```

4.1.2. setup_signal_handler: Using the `sigaction` API, the `setup_signal_handler` method sets up the `SIGINT` signal handling mechanism. It initializes `sa.sa_mask` to prevent further `SIGINT` signals during handler execution, sets `sa_handler` to `signal_handler`, and uses default settings for other flags (`sa.sa_flags`). By ensuring that the defined `signal_handler` function is run in response to `SIGINT`, this configuration enables the application to handle signals efficiently and avoid problems such as reentrancy.

```
void setup_signal_handler()
{
    struct sigaction sa;
    sa.sa_handler = signal_handler;
    sigemptyset(&sa.sa_mask); // Initialize the signal set to empty, then add signals to block.
    sigaddset(&sa.sa_mask, SIGINT); // Block SIGINT during the execution of the handler.
    sa.sa_flags = 0; // Use default settings.

    sigaction(SIGINT, &sa, NULL);
}
```

4.1.3. Semaphore manipulation: The struct `sembuf` definitions `sembuf_wait` and `sembuf_signal` manage semaphore operations for process synchronization. `sembuf_wait` decreases the semaphore's value by 1, causing the process to wait if the semaphore is zero, thereby acquiring a lock. Conversely, `sembuf_signal` increases the semaphore's value by 1, releasing the lock and allowing other processes to proceed. These operations ensure controlled access to shared resources, preventing race conditions.

```
struct sembuf sembuf_wait = {0, -1, 0};  
struct sembuf sembuf_signal = {0, 1, 0};
```

4.1.4. setup_server: In order to establish the server and set up a directory and FIFO for inter-process communication, the `setup_server` function in C is designed. Using `mkdir(dir_name, 0777)`, which gives all users full read, write, and execute rights, it first creates a directory where server files would be kept. Next, using `sprintf(server_fifo, "/tmp/%d", getpid())`, the code creates a unique path for a FIFO file by adding the server's process ID (PID) to a base path. This guarantees that the FIFO, generated by `mkfifo(server_fifo, 0666)`, is unique to this server instance and that every user has the ability to read and write it. The server successfully creates the resources required for IPC and prints messages to notify its startup and readiness to handle client connections.

```
void setup_server(char *server_fifo, char *dir_name)  
{  
    mkdir(dir_name, 0777);  
    sprintf(server_fifo, "/tmp/%d", getpid());  
    mkfifo(server_fifo, 0666);  
    printf("Server started PID %d\n", getpid());  
    printf("Server waiting for clients...\n");  
}
```

4.1.5. managing connections and assigning clients: First, a buffer is cleared, and incoming client data—including a PID—is read. After then, the server separates this PID from the related command (such as "connect" or "tryConnect"). It then starts a loop that searches an array monitoring active client PIDs for open slots, setting to zero any inactive PIDs found via a failed kill() check. The client's PID is assigned to any available slots, designating them as occupied. The server either warns of a full queue and skips assignment for further instructions, or it waits one second and retries if the command indicates a connection attempt. In order to prevent the server from handling more clients than it can handle and to ensure consistent operations, this configuration handles client connections before any intensive processing (such a fork() call for handling client-specific activities).

```
memset(buffer, 0, sizeof(buffer));
if (read(fd, buffer, sizeof(buffer)) > 0)
{
    // Extract PID from the buffer
    sprintf(pid, "%s", buffer);
    strtok(pid, " ");

    // Shift the buffer to the left to isolate the command
    memmove(buffer, &buffer[5], strlen(buffer) - 5 + 1);
    char *pid_token = strtok(buffer, " ");
    int ppid = atoi(pid_token);
    char *connect = strtok(NULL, " ");


    int client_assigned = 0;
    while (!client_assigned)
    {
        for (int i = 0; i < number_of_clients; i++)
        {
            // Check if a slot is free in the pid array (if the process is no longer active)
            if (kill(pid_array[i], 0) != 0)
                pid_array[i] = 0;

            // Assign the new client PID to the first free slot
            if (pid_array[i] == 0)
            {
                pid_array[i] = ppid;
                current = i;
                client_assigned = 1;
                break;
            }
        }
    }

    if (!client_assigned)
    {
        if (strcmp(connect, "connect", 7) == 0 || strcmp(connect, "tryConnect", 10) == 0)
        {
            // If trying to connect and no slots are available, sleep for 1 second and try again
            sleep(1);
        }
        else
        {
            // If not trying to connect explicitly, print queue full message and break
            printf("Connection request PID %d... Queue is FULL\n", ppid);
            break;
        }
    }
}

if (!client_assigned)
    continue;
```

4.1.6. list: After configuring an argument array for the ls command, which displays the contents of the current directory, it runs the command, substituting the ls program for the current process image. This technique is frequently used in server applications to give clients directory listing capabilities, enabling them to efficiently and simply examine files on the server.

```
void list()
{
    char *args[] = {"ls", NULL};
    execv("/bin/ls", args); // Execute the 'ls' command, replacing the current
} 
```

4.1.7. readF: The readF command, which is intended to read and show a file's contents in its whole or only certain lines, is handled by this code block in a server application by a client. The filename and an optional line number are first extracted from the second_buffer by parsing the input. It generates an error message in the absence of a filename. When a line number is entered and is valid (greater than zero), sed is used to create a command that prints just that line from the file. An error notice is displayed if the line number is incorrect. It creates a cat command to show the whole contents of the file if no line number is given. With the help of this feature, clients may remotely get all or a portion of file data, giving them various file reading options right through the server interface.

```
else if (strcmp(second_buffer, "readF", 5) == 0)
{
    // Parse the command to get filename and line number
    strtok(second_buffer, " ");
    char *filename = strtok(NULL, " ");
    char *lineNumberStr = strtok(NULL, "\n");

    if (filename == NULL)
    {
        printf("Filename is required.\n");
        return 0;
    }

    if (lineNumberStr != NULL)
    {
        int lineNumber = atoi(lineNumberStr);
        if (lineNumber > 0)
        {
            // Execute command to print specific line of the file
            char command[256];
            snprintf(command, sizeof(command), "sed -n '%dp' %s", lineNumber, filename);
            system(command);
        }
        else
        {
            printf("Invalid line number.\n");
        }
    }
    else
    {
        // Print the entire file if no line number is specified
        char command[256];
        snprintf(command, sizeof(command), "cat %s", filename);
        system(command);
    }
}
```


4.1.8. writeT: The writeT function allows you to add or replace text at a specific line. It does this by modifying text within a file using the sed program on Unix-based computers. The method begins by extracting the filename and expected text content from second_buffer through parsing. It checks to see if a line number is given; if not, or if the number is invalid or nonexistent (i.e., not precisely positive), it creates a sed command to add the text to the end of the file. It generates a sed script to replace the text at that particular line if a valid line number is provided. After that, execv runs the created command and makes the desired modifications to the file in place. This makes it possible for clients to specify changes remotely and allows for dynamic text editing on the server side.

```
void writeT(char *second_buffer)
{
    char *args[] = {"sed", "-i", "", "", NULL};
    strtok(second_buffer, " ");
    args[3] = strtok(NULL, " ");
    char *last_arg = strtok(NULL, "\n");
    char *copy = strdup(last_arg);
    char *number_str = strtok(last_arg, " ");
    int number = atoi(number_str);

    if (number == 0 && number_str != "0")
    {
        // Append text if no valid line number is provided
        args[2] = malloc((4 + strlen(copy)) * sizeof(char));
        sprintf(args[2], "$ a\\%", copy);
    }
    else
    {
        // Replace text at the specified line number
        args[2] = malloc((strlen(number_str) + strlen(copy)) * sizeof(char));
        copy[0] = 'c';
        copy[1] = '\\';
        sprintf(args[2], "%s%", number_str, copy); // Replacement command for 'sed'
    }

    execv("/bin/sed", args); // Execute 'sed' with the constructed command
}
```

4.1.9. download: The download function mimics a download process by making it easier to move files from the server's directory to a client's directory on the server. The paths of the source file on the server and the destination path in a different server-side directory meant for the client are extracted by processing the second buffer. These paths are set up as inputs for the cp command, which is frequently used to transfer files across Unix systems. When this command is run with execv, it moves the file from one place on the server to another, allowing clients to move and manage files inside the filesystem of the server as if they were being downloaded to their own directory space on the server. With this configuration, clients may safely and remotely store and transfer files within the server's structure.

```
void download(char *second_buffer)
{
    char *args[] = {"cp", "", "", NULL};
    strtok(second_buffer, " ");
    args[1] = strtok(NULL, " ");
    args[2] = strtok(NULL, " ");
    execv("/bin/cp", args); // Execute 'cp' to perform the download
}
```

4.1.10. upload: The upload function manages the process of uploading files from a client directory to a specified server directory. It allocates memory for the source path and sets it using the parameters of the cp command after parsing second_buffer to extract the file's source and target directories. The file is subsequently copied from the client-selected directory to the server-target directory using execv and cp, therefore controlling the server's file storage through client requests.

```
void upload(char *second_buffer)
{
    char *args[] = {"cp", "", ".", NULL};
    args[1] = malloc(256 * sizeof(char)); // Allocate memory for the source path
    strtok(second_buffer, " ");
    strcpy(args[1], strcat(strtok(NULL, " "), strtok(NULL, " ")));
    execv("/bin/cp", args); // Execute 'cp' to perform the upload
}
```

4.1.11. killServer: killServer method is intended to handle a client's request for termination. The server records a message stating that it has received a kill signal from the designated client and is terminating when it receives the command "killServer" from a client. Next, before continuing, it flushes the output to make sure all messages are written. The cleanup function iterates through the array of client PIDs, unlinking each client's unique FIFO and delivering a SIGTERM signal to gracefully terminate each child process connected to these clients. This procedure cleans up by destroying the server's primary FIFO file, which is used for communication. This effectively stops the server's activities in response to a client's request by ensuring an orderly shutdown of the server and all related processes.

```
else if (strncmp(second_buffer, "killServer", 10) == 0)
{
    printf("Kill signal from client%d... terminating...\n", current + 1);
    printf("Goodbye...\n");
    fflush(stdout);
    unlink(server_fifo);
    // Send termination signal to all child processes
    for (int i = 0; i < 3; i++)
    {
        char client_fifo[256];
        sprintf(client_fifo, "/tmp/%d", pid_array[i]);
        unlink(client_fifo);
        kill(pid_array[i], SIGTERM);
    }
}
```

4.1.12. quit: Clients that want to gently disengage from the server can use the quit command. The server logs and announces that it is ready to write to the server log file in response to a client sending this command. The client's disconnection is then confirmed by the server, which also prints a message indicating that it is awaiting the logging of this data. The client is identified by PID and sequence number (e.g., client1). By ensuring that all client-related actions are accurately recorded and wrapped up before the client disconnects, this procedure maintains a tidy and traceable session conclusion.

```
else if (strncmp(second_buffer, "quit", 4) == 0)
{
    strcpy(response, "Sending write request to server log file\nwaiting for log file...\n");
    printf("Sending write request to server log file\nwaiting for log file...\n");
    printf("Client PID %d (client%d) disconnected.\n", ppid, current + 1);
}
```

4.1.13. help: Clients can learn about possible server commands by using the help command. When a client gives a "help" command, the server interprets the input and, if supplied, extracts the precise command for which assistance is sought. The answer buffer and the particular command help sought are sent to the help function when the server calls it next. This feature helps clients comprehend and make efficient use of the server's capabilities by adding comprehensive use information to the response for either the specific command or the generic commands. By offering dynamic help directly through the server interface, this functionality improves the user experience.

```
void help(char *response, char *second_help)
{
    if (second_help == NULL)
        strcpy(response, "\nAvailable commands are:\nhelp, list, readF, writeF, upload, download, archServer, killServer, quit\n");
    else if (strncmp(second_help, "list", 4) == 0)
        strcpy(response, "\nlist\nDisplay the list of files in servers directory.\n");
    else if (strncmp(second_help, "readF", 5) == 0)
        strcpy(response, "\nreadF <file> <line #>\nRequests to display the # line of the <file>, if no line number is given the whole contents of the file is requested. (and displayed on the client side)\n");
    else if (strncmp(second_help, "writeF", 6) == 0)
        strcpy(response, "\nwriteF <file> <line #> <string>\nRequest to write the content of \"string\" to the #th line the <file>, if the line # is not given writes to the end of file. If the file does not exists in Servers directory creates and edits the file at the same time\n");
    else if (strncmp(second_help, "upload", 6) == 0)
        strcpy(response, "\nupload <file>\nUploads the file from the current working directory of client to the servers directory.\n");
    else if (strncmp(second_help, "download", 8) == 0)
        strcpy(response, "\ndownload <file>\nRequest to receive <file> from servers directory to client side\n");
    else if (strncmp(second_help, "archServer", 10) == 0)
        strcpy(response, "\narchServer <fileName>.tar\nUsing fork, exec and tar utilities create a child process that will collect all the files currently available on the the Server side and store them in the <fileName>.tar archive\n");
    else if (strncmp(second_help, "killServer", 10) == 0)
        strcpy(response, "\nkillServer\nSends a kill request to the Server.\n");
    else if (strncmp(second_help, "quit", 4) == 0)
        strcpy(response, "\nquit\nSend write request to Server side log file and quits.\n");
    else
        strcpy(response, "\nInvalid command. Please use 'help' to see the list of available commands.\n");
}
```

4.1.14. error_exit: Error reporting and process termination inside the server application are managed via the simple error_exit method. It passes the string error_message to the perror function after accepting it. This function is used to display the error message that has been sent to the standard error output, together with any extra data about the system fault. The function reports the problem and then calls exit(EXIT_FAILURE), which ends the process and outputs a non-zero exit code indicating that an error caused the program to stop. By using this technique, you can be confident that errors are properly recorded and that the program won't operate in an unexpected or unstable condition.

```
void error_exit(char *error_message)
{
    perror(error_message);
    exit(EXIT_FAILURE);
}
```

4.1.15. archServer: The `arch_server` function is made to archive the server directory's contents into a tar file. To guarantee proper file naming, it begins by removing any trailing spaces from the supplied filename. It instantly provides an error message if no filename is supplied. The function creates a new process to carry out the archiving procedure if a filename is supplied. The child process creates the full path for the tar command using the parent directory of the specified server directory, prints a message indicating the beginning of the archiving process, and then runs the tar command to create an archive file in the parent directory with the provided filename. The child process terminates cleanly if the tar command runs properly. The parent process, in the meantime, waits for the child to complete, verifies the exit status, and modifies the response buffer to reflect the success or failure of the archiving operation. By using this method, the server may manage asynchronously potentially lengthy archiving processes, reducing interference with other server functions.

```
void arch_server(char *response, char *dir_name, char *file_name)
{
    // Trim trailing spaces from the filename
    size_t file_name_length = strlen(file_name);
    while (file_name_length > 0 && isspace(file_name[file_name_length - 1]))
    {
        file_name[--file_name_length] = '\0';
    }

    // Check if filename is provided, if not, send error response
    if (file_name == NULL)
        strcpy(response, "Error: No filename provided.\n");
    else
    {
        // Fork a new process to handle the archiving
        int archive_child = fork();
        if (archive_child == 0)
        {
            // Child process: Executes the tar command to archive the server's content
            printf("Archiving the current contents of the server...\n");
            char tarCommand[1024];
            char parent_dir[1024];

            // Construct the path to the parent directory and the tar command
            sprintf(parent_dir, "%s/..", dir_name);
            sprintf(tarCommand, "%s/%s.tar", parent_dir, file_name);
            printf("%s\n", dir_name);
            printf("%s\n", file_name);

            // Execute the tar command to create an archive
            execlp("tar", "tar", "cf", tarCommand, "-C", parent_dir, ".", NULL);
            perror("Failed to create archive!!!");
            exit(EXIT_FAILURE);
        }
        else
        {
            // Parent process waits for the child to complete
            int status;
            waitpid(archive_child, &status, 0);
            if (WIFEXITED(status) && WEXITSTATUS(status) == 0)
            {
                // If child process exits successfully, update response to indicate success
                printf("Calling tar utility .. child PID %d\n", archive_child);
                printf("Child process completed successfully.\n");
                printf("Copying the archive file..\n");
                sprintf(response, "SUCCESS: Server side files are archived in \"%s.tar\"\n", file_name);
            }
            else
            {
                // If child process fails, update response to indicate error
                strcpy(response, "ERROR in archiving process.\n");
            }
        }
    }
}
```

4.2. Client.c:

4.2.1. Semaphore manipulation: The struct sembuf definitions sembuf_wait and sembuf_signal manage semaphore operations for process synchronization. sembuf_wait decreases the semaphore's value by 1, causing the process to wait if the semaphore is zero, thereby acquiring a lock. Conversely, sembuf_signal increases the semaphore's value by 1, releasing the lock and allowing other processes to proceed. These operations ensure controlled access to shared resources, preventing race conditions.

```
struct sembuf sembuf_wait = {0, -1, 0};  
struct sembuf sembuf_signal = {0, 1, 0};
```

4.2.2. Semaphore Initialization for Client-Server Synchronization: Each semaphore is uniquely identified by the process's PID and its incremented value. Semid, the initial semaphore, is probably used to synchronize the primary communication operations and ensure orderly data transfers. Semid is generated with the current process ID and a beginning value of zero. Semid2, the second semaphore, might be used for other coordination duties like controlling access to other shared resources or notifying the end of data processing. Semid2 is started with the process ID plus one and starts at zero.

```
// Initialize semaphores for synchronization  
int semid = initialize_semaphore(getpid(), 0);  
int semid2 = initialize_semaphore(getpid() + 1, 0);
```

4.2.3. open_fifo: The client's open_fifo function. The c code is in charge of opening a FIFO (called pipe) with the access modes given in flags, as indicated by the fifo_name argument. Using the open system call, it tries to open the FIFO and verifies that the operation was successful. The function handles an error promptly by running the error_exit function with a particular error message ("open fifo error!!!") and ending the operation with an error status if the open call fails, indicating a problem with accessing or constructing the FIFO. This configuration offers strong error handling to stop the process from operating under unfavorable conditions by ensuring that the required communication channels via FIFOs are correctly established before moving on with more IPC activities.

```
int open_fifo(const char *fifo_name, int flags)
{
    int fd = open(fifo_name, flags);
    if (fd < 0)
    {
        error_exit("open fifo error!!!\n");
    }
    return fd;
}
```

4.2.4. Client Identification and Initial Communication Setup: It begins by formatting a string pid, which creates a unique identification like "/tmp/[PID] [command]" by combining the client's process ID (obtained via getpid()) with a command or identifier from the command-line arguments (argv[1]). The write function is then used to send this identification to the server across a file descriptor (fd), which is usually attached to a socket or FIFO. Following transmission, the code tokenizes the pid string by extracting the first segment before a space using strtok. Next, it realigns this token at the beginning of the pid buffer using memmove, guaranteeing that the buffer is properly ordered and ended for any further actions. Effectively starting the communication channel between the client and server, this procedure signals the start of their exchange.

```
// Send client identification to server
char pid[100];
sprintf(pid, "/tmp/%d %s", getpid(), argv[1]);
write(fd, pid, strlen(pid));
char *token = strtok(pid, " ");
if (token != NULL)
{
    memmove(pid, token, strlen(token) + 1); // Include null terminator
}
```

4.2.5. handle_client: The `handle_client` function continually asks the user for instructions, confirms their correctness, and processes them to enable interactive communication between a client and a server. To precisely guide file operations, commands like "upload" and "download" are added to the current working directory. A FIFO is used to send valid orders to the server, and semaphore signaling is used to maintain synchronization. After that, the function shows the user the response after waiting for it from the server. In the event that a "quit" command is input, the function exits, signals via semaphore, and transmits this to the server, therefore ending the session. For different command executions, this procedure guarantees dynamic and coordinated communication between the client and server.

```
void handle_client(int client_fd, int semid, int semid2)
{
    char buffer[65536], line[65536], cwd[100];
    int bytes_read;

    printf("Connection established!\n");

    while (true)
    {
        memset(buffer, 0, sizeof(buffer)); // Clear buffer before use
        printf("\nEnter command: ");
        if (fgets(line, sizeof(line), stdin) == NULL)
        {
            printf("Error reading input, please try again.\n");
            continue;
        }
        line[strlen(line) - 1] = '\0'; // Proper null termination after removing newline

        if (strcmp(line, "quit") == 0)
        {
            strncpy(buffer, line, sizeof(buffer) - 1);
            write(client_fd, buffer, strlen(buffer));
            semaphore_signal(semid2);
            break;
        }

        if (!is_valid_command(line))
        {
            printf("Invalid command. Please try a valid command.\n");
            continue;
        }

        // Prepare command for upload or download with current working directory
        if (strcmp(line, "upload", 6) == 0 || strcmp(line, "download", 8) == 0)
        {
            if (!getcwd(cwd, sizeof(cwd)))
            {
                printf("Failed to get current working directory.\n");
                continue;
            }
            snprintf(buffer, sizeof(buffer), "%s %s/", line, cwd); // Safer buffer handling with snprintf
        }
        else
        {
            strncpy(buffer, line, sizeof(buffer) - 1); // Use safer string copy
            buffer[sizeof(buffer) - 1] = '\0'; // Ensure null termination
        }

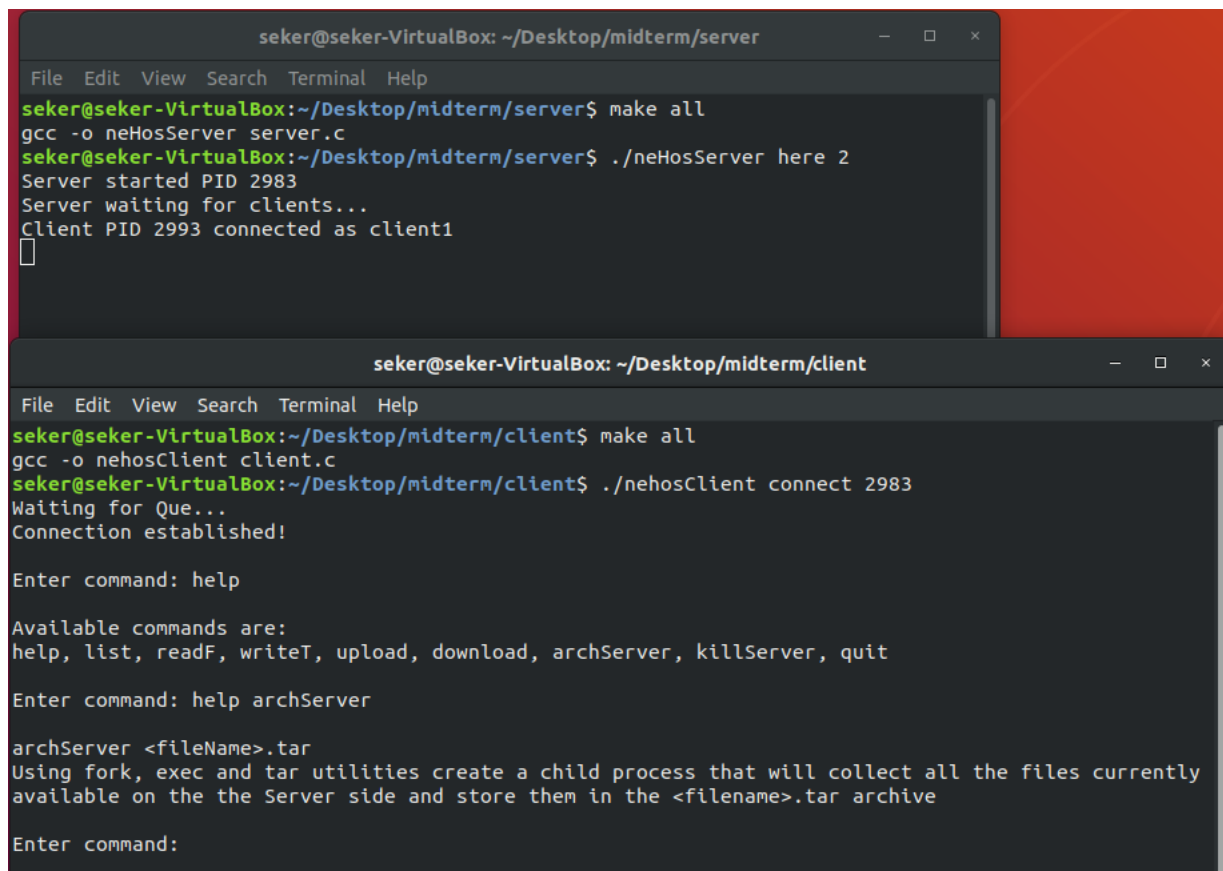
        // Communicate with server
        write(client_fd, buffer, strlen(buffer));
        semaphore_signal(semid2);
        semaphore_wait(semid2);

        // Read and display server response
        while ((bytes_read = read(client_fd, buffer, sizeof(buffer) - 1)) > 0)
        {
            buffer[bytes_read] = '\0'; // Null-terminate the buffer
            printf("%s", buffer);

            if (bytes_read < sizeof(buffer) - 1)
                break; // Break if we read less than the buffer size, likely means we've reached the end
        }
    }
}
```


5. Example Outputs:

5.1. help:



The image shows two terminal windows from a VirtualBox environment. The top window is titled 'seker@seker-VirtualBox: ~/Desktop/midterm/server' and shows the compilation of 'server.c' into 'neHosServer' and its execution. The server starts at PID 2983 and waits for clients. A client connects at PID 2993 as 'client1'. The bottom window is titled 'seker@seker-VirtualBox: ~/Desktop/midterm/client' and shows the compilation of 'client.c' into 'nehosClient' and its execution. The client connects to the server at PID 2983. It then enters a command 'help' and receives a list of available commands: 'help', 'list', 'readF', 'writeT', 'upload', 'download', 'archServer', 'killServer', and 'quit'. Finally, it enters 'help archServer' and receives a description of the 'archServer' command.

```
seker@seker-VirtualBox: ~/Desktop/midterm/server
File Edit View Search Terminal Help
seker@seker-VirtualBox:~/Desktop/midterm/server$ make all
gcc -o neHosServer server.c
seker@seker-VirtualBox:~/Desktop/midterm/server$ ./neHosServer here 2
Server started PID 2983
Server waiting for clients...
Client PID 2993 connected as client1
█

seker@seker-VirtualBox: ~/Desktop/midterm/client
File Edit View Search Terminal Help
seker@seker-VirtualBox:~/Desktop/midterm/client$ make all
gcc -o nehosClient client.c
seker@seker-VirtualBox:~/Desktop/midterm/client$ ./nehosClient connect 2983
Waiting for Que...
Connection established!

Enter command: help

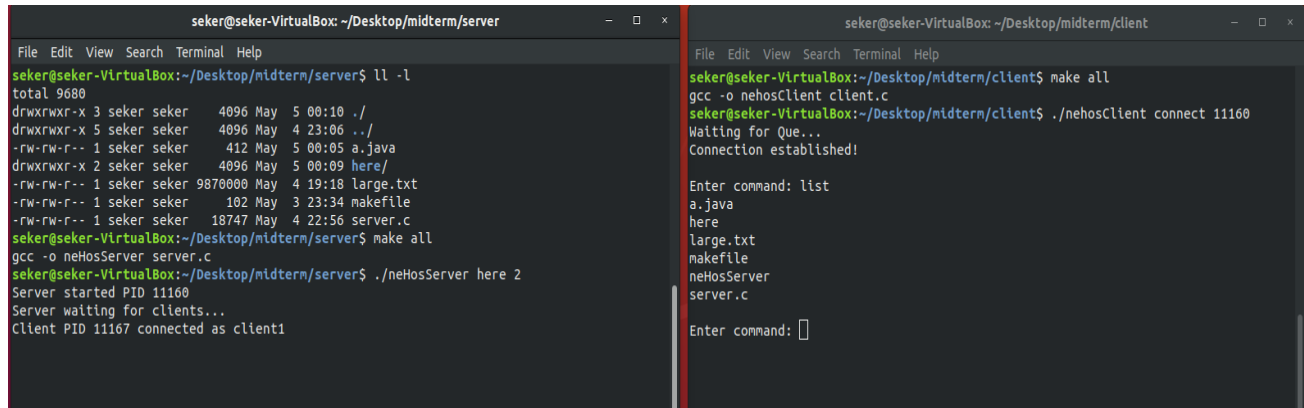
Available commands are:
help, list, readF, writeT, upload, download, archServer, killServer, quit

Enter command: help archServer

archServer <fileName>.tar
Using fork, exec and tar utilities create a child process that will collect all the files currently
available on the the Server side and store them in the <filename>.tar archive

Enter command:
```

5.2. list



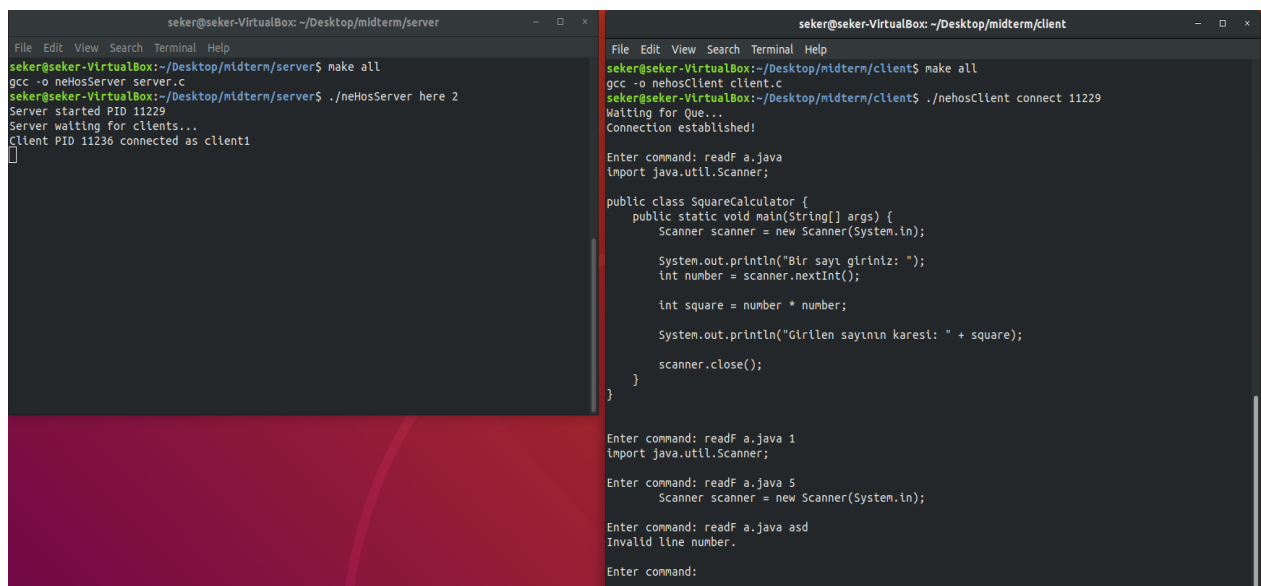
```
seker@seker-VirtualBox: ~/Desktop/midterm/server
File Edit View Search Terminal Help
seker@seker-VirtualBox:~/Desktop/midterm/server$ ll -l
total 9680
drwxrwxr-x 3 seker seker 4096 May 5 00:10 ./
drwxrwxr-x 5 seker seker 4096 May 4 23:06 ../
-rw-rw-r-- 1 seker seker 412 May 5 00:05 a.java
drwxrwxr-x 2 seker seker 4096 May 5 00:09 here/
-rw-rw-r-- 1 seker seker 9870000 May 4 19:18 large.txt
-rw-rw-r-- 1 seker seker 102 May 3 23:34 makefile
-rw-rw-r-- 1 seker seker 18747 May 4 22:56 server.c
seker@seker-VirtualBox:~/Desktop/midterm/server$ make all
gcc -o neHosServer server.c
seker@seker-VirtualBox:~/Desktop/midterm/server$ ./neHosServer here 2
Server started PID 11160
Server waiting for clients...
Client PID 11167 connected as client1

seker@seker-VirtualBox: ~/Desktop/midterm/client
File Edit View Search Terminal Help
seker@seker-VirtualBox:~/Desktop/midterm/client$ make all
gcc -o nehosClient client.c
seker@seker-VirtualBox:~/Desktop/midterm/client$ ./nehosClient connect 11160
Waiting for Que...
Connection established!

Enter command: list
a.java
here
large.txt
makefile
neHosServer
server.c

Enter command: 
```

5.3. readF



```
seker@seker-VirtualBox: ~/Desktop/midterm/server
File Edit View Search Terminal Help
seker@seker-VirtualBox:~/Desktop/midterm/server$ make all
gcc -o neHosServer server.c
seker@seker-VirtualBox:~/Desktop/midterm/server$ ./neHosServer here 2
Server started PID 11229
Server waiting for clients...
Client PID 11236 connected as client1
[]

seker@seker-VirtualBox: ~/Desktop/midterm/client
File Edit View Search Terminal Help
seker@seker-VirtualBox:~/Desktop/midterm/client$ make all
gcc -o nehosClient client.c
seker@seker-VirtualBox:~/Desktop/midterm/client$ ./nehosClient connect 11229
Waiting for Que...
Connection established!

Enter command: readF a.java
import java.util.Scanner;

public class SquareCalculator {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.println("Bir sayı giriniz: ");
        int number = scanner.nextInt();

        int square = number * number;

        System.out.println("Girilen sayının karesi: " + square);

        scanner.close();
    }
}

Enter command: readF a.java 1
import java.util.Scanner;

Enter command: readF a.java 5
Scanner scanner = new Scanner(System.in);

Enter command: readF a.java asd
Invalid line number.

Enter command: 
```

Unfortunately, when trying to read very large files, it writes to fifo, but it can't print it to the terminal because it can't read it. :(

5.4. writeT

```
seker@seker-VirtualBox: ~/Desktop/midterm/server
File Edit View Search Terminal Help
seker@seker-VirtualBox:~/Desktop/midterm/server$ make all
gcc -o neHosServer server.c
seker@seker-VirtualBox:~/Desktop/midterm/server$ ./neHosServer here 2
Server started PID 11369
Server waiting for clients...
Client PID 11376 connected as client1
█

seker@seker-VirtualBox: ~/Desktop/midterm/client
File Edit View Search Terminal Help
seker@seker-VirtualBox:~/Desktop/midterm/client$ make all
gcc -o neHosClient client.c
seker@seker-VirtualBox:~/Desktop/midterm/client$ ./neHosClient connect 11369
Waiting for Que...
Connection established!

Enter command: readF b.c
#include <stdio.h>

int main() {
    int number;
    printf("Bir sayı giriniz: ");
    scanf("%d", &number);
    printf("%d", number);
    return 0;
}

Enter command: writeT b.c 2 ERKAN ZERGEROGLU
Task completed successfully.

Enter command: readF b.c
#include <stdio.h>
ERKAN ZERGEROGLU
int main() {
    int number;
    printf("Bir sayı giriniz: ");
    scanf("%d", &number);
    printf("%d", number);
    return 0;
}
```

```
seker@seker-VirtualBox: ~/Desktop/midterm/server
File Edit View Search Terminal Help
seker@seker-VirtualBox:~/Desktop/midterm/server$ make all
gcc -o neHosServer server.c
seker@seker-VirtualBox:~/Desktop/midterm/server$ ./neHosServer here 2
Server started PID 11369
Server waiting for clients...
Client PID 11376 connected as client1
█

seker@seker-VirtualBox: ~/Desktop/midterm/client
File Edit View Search Terminal Help
}

Enter command: writeT b.c 2 ERKAN ZERGEROGLU
Task completed successfully.

Enter command: readF b.c
#include <stdio.h>
ERKAN ZERGEROGLU
int main() {
    int number;
    printf("Bir sayı giriniz: ");
    scanf("%d", &number);
    printf("%d", number);
    return 0;
}

Enter command: readF b.c
#include <stdio.h>
ERKAN ZERGEROGLU
int main() {
    int number;
    printf("Bir sayı giriniz: ");
    scanf("%d", &number);
    printf("%d", number);
    return 0;
}

Enter command: writeT b.c YOU HAVE 1 NEW MESSAGE
Task completed successfully.

Enter command: readF b.c
#include <stdio.h>
ERKAN ZERGEROGLU
int main() {
    int number;
    printf("Bir sayı giriniz: ");
    scanf("%d", &number);
    printf("%d", number);
    return 0;
}

YOU HAVE 1 NEW MESSAGE

Enter command:
```

5.5. download

```
seker@seker-VirtualBox: ~/Desktop/midterm/server
File Edit View Search Terminal Help
seker@seker-VirtualBox:~/Desktop/midterm/server$ ll -l
total 9684
drwxrwxr-x 3 seker seker 4096 May 5 00:46 ./
drwxrwxr-x 5 seker seker 4096 May 4 23:06 ../
-rw-rw-r-- 1 seker seker 434 May 5 00:19 a.java
-rw-rw-r-- 1 seker seker 192 May 5 00:23 b.c
drwxrwxr-x 2 seker seker 4096 May 5 00:19 here/
-rw-rw-r-- 1 seker seker 9870000 May 4 19:18 large.txt
-rw-rw-r-- 1 seker seker 102 May 3 23:34 makefile
-rw-rw-r-- 1 seker seker 18747 May 5 00:38 server.c
seker@seker-VirtualBox:~/Desktop/midterm/server$ make all
gcc -o nehosServer server.c
seker@seker-VirtualBox:~/Desktop/midterm/server$ ./nehosServer here 2
Server started PID 12090
Server waiting for clients...
Client PID 12099 connected as client1
Kill signal from client1... terminating...
Goodbye...
Terminated
seker@seker-VirtualBox:~/Desktop/midterm/server$

seker@seker-VirtualBox: ~/Desktop/midterm/client
File Edit View Search Terminal Help
seker@seker-VirtualBox:~/Desktop/midterm/client$ ll -l
total 216
drwxrwxr-x 2 seker seker 4096 May 5 00:47 ./
drwxrwxr-x 5 seker seker 4096 May 4 23:06 ../
-rw-rw-r-- 1 seker seker 222 May 5 00:05 a.c
-rw-rw-r-- 1 seker seker 5049 May 4 23:02 client.c
-rw-rw-r-- 1 seker seker 110 Nis 28 22:40 makefile
-rw-rw-r-- 1 seker seker 194879 May 4 09:35 ne.jpg
seker@seker-VirtualBox:~/Desktop/midterm/client$ make all
gcc -o nehosClient client.c
seker@seker-VirtualBox:~/Desktop/midterm/client$ ./nehosClient connect 12090
Waiting for Que...
Connection established!

Enter command: list
a.java
b.c
here
large.txt
makefile
nehosServer
server.c

Enter command: download a.java
Task completed successfully.

Enter command: download large.txt
Task completed successfully.

Enter command: killServer
Terminated
seker@seker-VirtualBox:~/Desktop/midterm/client$ ll -l
total 9876
drwxrwxr-x 2 seker seker 4096 May 5 00:48 ./
drwxrwxr-x 5 seker seker 4096 May 4 23:06 ../
-rw-rw-r-- 1 seker seker 222 May 5 00:05 a.c
-rw-rw-r-- 1 seker seker 434 May 5 00:48 a.java
-rw-rw-r-- 1 seker seker 5049 May 4 23:02 client.c
-rw-rw-r-- 1 seker seker 9870000 May 5 00:48 large.txt
-rw-rw-r-- 1 seker seker 110 Nis 28 22:40 makefile
-rw-rw-r-- 1 seker seker 194879 May 4 09:35 ne.jpg
-rwxrwxr-x 1 seker seker 13952 May 5 00:47 nehosClient*
seker@seker-VirtualBox:~/Desktop/midterm/client$
```

5.6. upload

```
seker@seker-VirtualBox: ~/Desktop/midterm/server
File Edit View Search Terminal Help
seker@seker-VirtualBox:~/Desktop/midterm/server$ ll -l
total 9684
drwxrwxr-x 3 seker seker 4096 May 5 00:26 ./
drwxrwxr-x 5 seker seker 4096 May 4 23:06 ../
-rw-rw-r-- 1 seker seker 434 May 5 00:19 a.java
-rw-rw-r-- 1 seker seker 192 May 5 00:23 b.c
drwxrwxr-x 2 seker seker 4096 May 5 00:25 here/
-rw-rw-r-- 1 seker seker 9870000 May 4 19:18 large.txt
-rw-rw-r-- 1 seker seker 102 May 3 23:34 makefile
-rw-rw-r-- 1 seker seker 18747 May 4 22:56 server.c
seker@seker-VirtualBox:~/Desktop/midterm/server$ make all
gcc -o nehosServer server.c
seker@seker-VirtualBox:~/Desktop/midterm/server$ ./nehosServer here 2
Server started PID 11555
Server waiting for clients...
Client PID 11562 connected as client1
seker@seker-VirtualBox:~/Desktop/midterm/server$

seker@seker-VirtualBox: ~/Desktop/midterm/client
File Edit View Search Terminal Help
seker@seker-VirtualBox:~/Desktop/midterm/client$ ll -l
total 216
drwxrwxr-x 2 seker seker 4096 May 5 00:25 ./
drwxrwxr-x 5 seker seker 4096 May 4 23:06 ../
-rw-rw-r-- 1 seker seker 222 May 5 00:05 a.c
-rw-rw-r-- 1 seker seker 5049 May 4 23:02 client.c
-rw-rw-r-- 1 seker seker 110 Nis 28 22:40 makefile
-rw-rw-r-- 1 seker seker 194879 May 4 09:35 ne.jpg
seker@seker-VirtualBox:~/Desktop/midterm/client$ make all
gcc -o nehosClient client.c
seker@seker-VirtualBox:~/Desktop/midterm/client$ ./nehosClient connect 11555
Waiting for Que...
Connection established!

Enter command: list
a.java
b.c
here
large.txt
makefile
nehosServer
server.c

Enter command: upload a.c
Task completed successfully.

Enter command: list
a.c
a.java
b.c
here
large.txt
makefile
nehosServer
server.c

Enter command:
```

5.7. archServer

The screenshot shows two terminal windows and a file explorer. The left terminal window, titled 'seker@seker-VirtualBox: ~/Desktop/midterm/server', shows the execution of 'make all' and 'makefile' for 'neHosServer'. It then starts the server with PID 12905, which waits for clients. A client with PID 12912 connects, and the server archives the current contents of the 'here' directory into 'arsiv.tar'. The file explorer, titled 'arsiv.tar', shows the contents of the archive: 'here' (Folder), 'a.java' (Java source code), 'b.c' (C source code), 'large.txt' (plain text document), 'makefile' (Makefile), 'neHosServer' (unknown), and 'server.c' (C source code). The right terminal window, titled 'seker@seker-VirtualBox: ~/Desktop/midterm/client', shows the execution of 'make all' and 'makefile' for 'neHosClient'. It then connects to the server with PID 12905. The client sends the 'list' command, and the server responds with the contents of the 'here' directory: 'a.java', 'arsiv.tar', 'b.c', 'here', 'large.txt', 'makefile', 'neHosServer', and 'server.c'.

```
seker@seker-VirtualBox: ~/Desktop/midterm/server
File Edit View Search Terminal Help
seker@seker-VirtualBox:~/Desktop/midterm/server$ make all
gcc -o neHosServer server.c
seker@seker-VirtualBox:~/Desktop/midterm/server$ ./neHosServer here 2
Server started PID 12905
Server waiting for clients...
Client PID 12912 connected as client1
Archiving the current contents of the server...
here
arsiv
tar: ./arsiv.tar: file is the archive; not dumped
Calling tar utility .. child PID 12916
Child process completed successfully.
Copying the archive file..
[]

seker@seker-VirtualBox: ~/Desktop/midterm/client
File Edit View Search Terminal Help
seker@seker-VirtualBox:~/Desktop/midterm/client$ make all
gcc -o neHosClient client.c
seker@seker-VirtualBox:~/Desktop/midterm/client$ ./neHosClient connect 12905
Waiting for Que...
Connection established!

Enter command: list
a.java
b.c
here
large.txt
makefile
neHosServer
server.c

Enter command: archServer arsv
SUCCESS: Server side files are archived in "arsiv.tar"

Enter command: list
a.java
arsiv.tar
b.c
here
large.txt
makefile
neHosServer
server.c

Enter command: []
```

5.8. quit

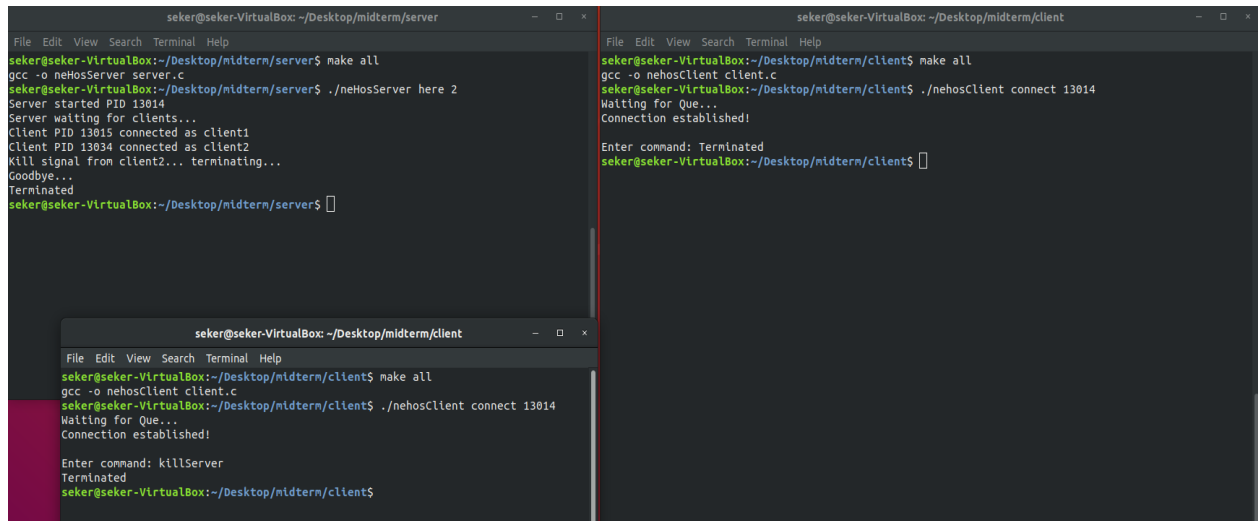
The screenshot shows two terminal windows. The left terminal window, titled 'seker@seker-VirtualBox: ~/Desktop/midterm/server', shows the execution of 'make all' and 'makefile' for 'neHosServer'. It then starts the server with PID 12974, which waits for clients. A client with PID 12986 connects, and the server sends a write request to the server log file. The client then disconnects. The right terminal window, titled 'seker@seker-VirtualBox: ~/Desktop/midterm/client', shows the execution of 'make all' and 'makefile' for 'neHosClient'. It then connects to the server with PID 12974. The client sends the 'quit' command, and the server responds with the contents of the 'here' directory: 'a.java', 'arsiv.tar', 'b.c', 'here', 'large.txt', 'makefile', 'neHosServer', and 'server.c'.

```
seker@seker-VirtualBox: ~/Desktop/midterm/server
File Edit View Search Terminal Help
seker@seker-VirtualBox:~/Desktop/midterm/server$ make all
gcc -o neHosServer server.c
seker@seker-VirtualBox:~/Desktop/midterm/server$ ./neHosServer here 2
Server started PID 12974
Server waiting for clients...
Client PID 12986 connected as client1
Sending write request to server log file
waiting for log file...
Client PID 12986 (client1) disconnected.
[]

seker@seker-VirtualBox: ~/Desktop/midterm/client
File Edit View Search Terminal Help
seker@seker-VirtualBox:~/Desktop/midterm/client$ make all
gcc -o neHosClient client.c
seker@seker-VirtualBox:~/Desktop/midterm/client$ ./neHosClient connect 12974
Waiting for Que...
Connection established!

Enter command: quit
seker@seker-VirtualBox:~/Desktop/midterm/client$
```

5.9. killServer

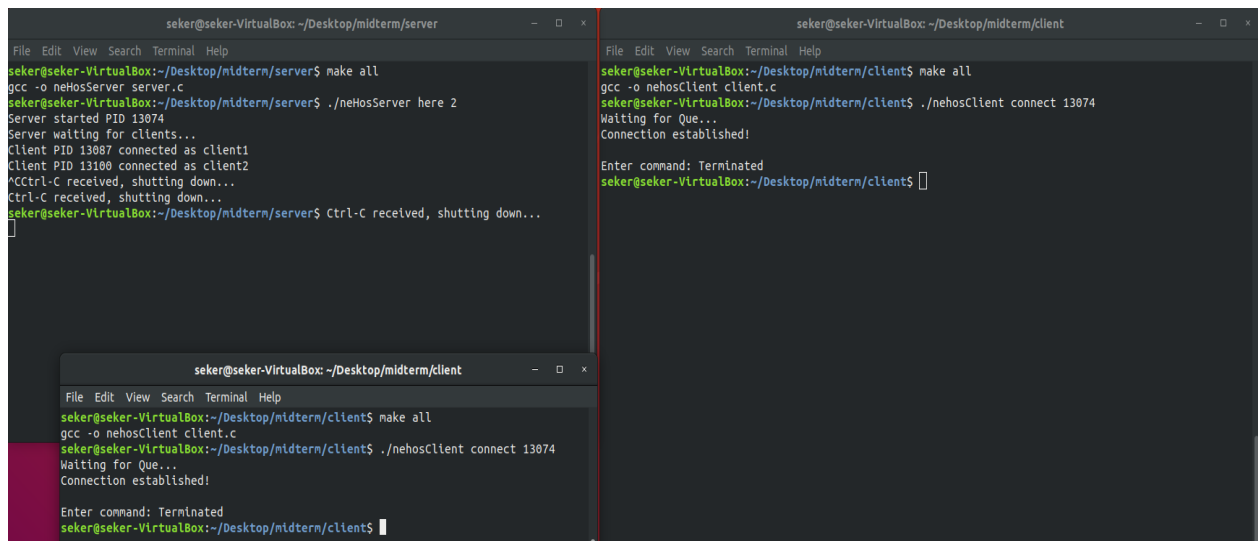


The screenshot shows two terminal windows side-by-side. The left window is titled 'seker@seker-VirtualBox: ~/Desktop/midterm/server' and the right window is titled 'seker@seker-VirtualBox: ~/Desktop/midterm/client'. Both windows show the execution of a 'make all' command, which compiles 'nehosServer' and 'nehosClient' respectively. The server window then shows the execution of './nehosServer here 2', which starts the server on PID 13014. The client window shows the execution of './nehosClient connect 13014', which establishes a connection. The server window then shows the execution of 'killServer', which terminates the server. The client window shows the execution of 'killServer', which terminates the client.

```
seker@seker-VirtualBox: ~/Desktop/midterm/server
File Edit View Search Terminal Help
seker@seker-VirtualBox:~/Desktop/midterm/server$ make all
gcc -o nehosServer server.c
seker@seker-VirtualBox:~/Desktop/midterm/server$ ./nehosServer here 2
Server started PID 13014
Server waiting for clients...
Client PID 13015 connected as client1
Client PID 13034 connected as client2
Kill signal from client2... terminating...
Goodbye...
Terminated
seker@seker-VirtualBox:~/Desktop/midterm/server$

seker@seker-VirtualBox: ~/Desktop/midterm/client
File Edit View Search Terminal Help
seker@seker-VirtualBox:~/Desktop/midterm/client$ make all
gcc -o nehosClient client.c
seker@seker-VirtualBox:~/Desktop/midterm/client$ ./nehosClient connect 13014
Waiting for Que...
Connection established!
Enter command: Terminated
seker@seker-VirtualBox:~/Desktop/midterm/client$
```

5.10. Ctrl-C



The screenshot shows two terminal windows side-by-side. The left window is titled 'seker@seker-VirtualBox: ~/Desktop/midterm/server' and the right window is titled 'seker@seker-VirtualBox: ~/Desktop/midterm/client'. Both windows show the execution of a 'make all' command, which compiles 'nehosServer' and 'nehosClient' respectively. The server window then shows the execution of './nehosServer here 2', which starts the server on PID 13074. The client window shows the execution of './nehosClient connect 13074', which establishes a connection. The server window then shows the execution of 'Ctrl-C', which terminates the server. The client window shows the execution of 'Ctrl-C', which terminates the client.

```
seker@seker-VirtualBox: ~/Desktop/midterm/server
File Edit View Search Terminal Help
seker@seker-VirtualBox:~/Desktop/midterm/server$ make all
gcc -o nehosServer server.c
seker@seker-VirtualBox:~/Desktop/midterm/server$ ./nehosServer here 2
Server started PID 13074
Server waiting for clients...
Client PID 13087 connected as client1
Client PID 13100 connected as client2
^C Ctrl-C received, shutting down...
Ctrl-C received, shutting down...
seker@seker-VirtualBox:~/Desktop/midterm/server$ Ctrl-C received, shutting down...
^C

seker@seker-VirtualBox: ~/Desktop/midterm/client
File Edit View Search Terminal Help
seker@seker-VirtualBox:~/Desktop/midterm/client$ make all
gcc -o nehosClient client.c
seker@seker-VirtualBox:~/Desktop/midterm/client$ ./nehosClient connect 13074
Waiting for Que...
Connection established!
Enter command: Terminated
seker@seker-VirtualBox:~/Desktop/midterm/client$
```