CSE 344 MIDTERM PROJECT REPORT

Şehmus Acar

161044085

System Application Files Overview

client.c

Handles user input, sends various commands to a server, and processes the server's responses. Includes functionalities to send kill signals to the server, list and explain available commands, and manage interaction through a command-line interface.

directory.c

Provides functions for manipulating files within directories, such as copying files from one location to another, reading specific lines from files, and listing all files in a directory. Supports file operations crucial for server-client interactions in file management systems.

directory.h

Header file for **directory.c**, declaring function prototypes and necessary includes for managing file and directory operations. This allows other components of the system to utilize these functions for file manipulation and querying directory contents.

Queue.c

Implements a queue data structure for managing data in a first-in-first-out (FIFO) manner. Includes functions for initializing a queue, inserting items, checking if the queue is full or empty, and removing items from the queue. Typically used for buffering tasks or data within the system.

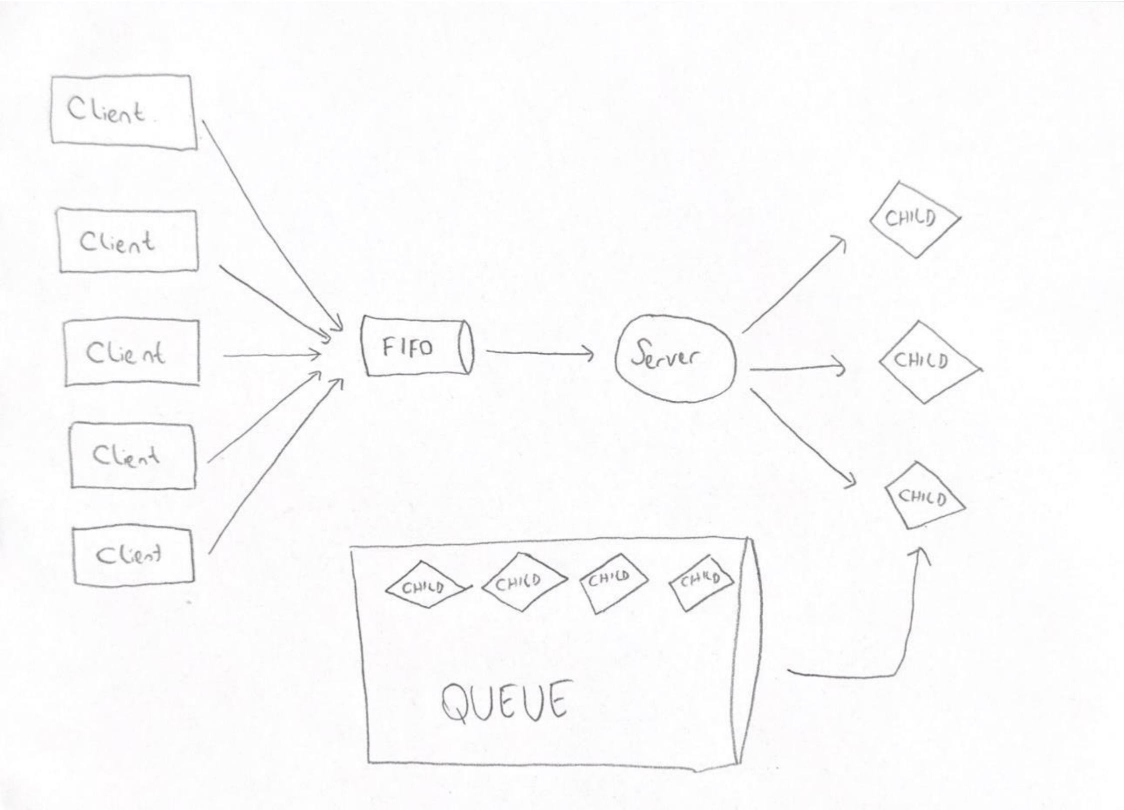
Queue.h

Header file for **Queue.c**, defining the structure of the queue and the prototypes for operations on the queue. Ensures that the queue functionality can be accessed and used correctly throughout the program.

server.c

Manages multiple client connections and processes client commands related to file operations such as reading, writing, and downloading files. Utilizes IPC mechanisms like FIFOs, semaphores, and shared memory to effectively manage client requests and synchronize access to shared resources

Multi-Client Server Architecture with Queuing System



In this setup, there are numerous clients and one server. I've employed the FIFO method for communication between the server and its child processes, yet there are critical considerations at this juncture. The server has a limited number of child slots, so if there are more clients than available slots, they need to be queued due to the importance of order. For this purpose, I implemented a basic queue structure, which can be found in the queue.c and queue.h files. If all slots are occupied and there are additional clients waiting, the server generates new child processes for these clients, but these new children do not operate until an existing child terminates. For this**, I used a semaphore**. I initialized a semaphore with the value equal to the number of child slots, and a child process calls sem_wait when it starts and sem_post upon completion. For example, if the server has 4 client slots, the initial semaphore value is 4, and as 4 clients arrive, each calls sem_wait, which will reduce the semaphore value to 0 after 4 clients. When the 5th client arrives, it will also call sem_wait but will be blocked at this point because the semaphore value is 0. Once a client completes, sem_post is called, and the 5th client can begin to operate.

Semaphore-Based Synchronization in Client Queue Management

Client Slot	Semaphore Value	İndex of coming client
4	4	0
3	3	1
2	2	2
1	1	3
0	0	4
0	**0**	**5**

At this juncture, whenever a client completes its work, the semaphore is unlocked. Each client receives a pid to open a new fifo for communication with the client. Therefore, the parent server stores these pids in the queue, and each client takes and removes the first element of the queue. This arrangement prevents bugs where some clients are in the queue, a client completes, and at that moment another client enters the server and can start working because a slot is now available. **However, in my design, every client is stored in the queue, so a new client will be placed at the tail of the queue, avoiding such issues.**

Moreover, I utilized another semaphore to synchronize the addition and removal in the queue. Since the queue is in shared memory and every process can access this shared memory, a race condition occurs, and I used this semaphore to resolve this race condition.

THE WORKFLOW OF THE CONCURRENT FILE ACCESS SYSTEM

1. Client Connection and Server Initialization

- The client initiates a connection using the **neHosClient** command, specifying the server's PID and the type of connection (**Connect** or **tryConnect**) via command line arguments.
- The server is launched with the **neHosServer** command, which takes maximum client count and the directory to serve as command line parameters.
- Upon startup, the server operates in the specified directory, creates it if it doesn't exist, and opens a log file for recording client activities.

2. Client Queue Management

- The client sends a connection request to the server queue. With the **Connect** option, the client waits if the queue is full until a connection spot becomes available. With **tryConnect**, the client will exit without waiting if no spots are available.
- Once space is available, the server notifies the client of the connection acceptance and spawns a new process to handle the client's commands.

3. Command Processing

- The client can send various file operation commands to the server, including **list**, **readF**, **writeT**, **upload**, **download**, **archServer**, **killServer**, and **quit**.
- The server receives each command and invokes appropriate file operation functions to perform the requested actions. These operations are managed within a common directory on the server side.

4. File Operations

- **Listing**: Using the **list** command, the client requests a list of files from the server's directory. The server retrieves the file list and sends it back to the client.
- **Reading and Writing**: Commands like **readF** and **writeT** allow the client to read or write to a specific line of a specified file. The server opens the file, executes the requested operation, and reports the outcome to the client.
- **Uploading and Downloading**: Files are transferred between the client and the server using the **upload** and **download** commands.

5. Server Shutdown and Exit Processes

- Through the **killServer** command, the client sends a termination signal to the server. The server then sends signals to all child processes, updates the log file, and terminates the program.
- The **quit** command allows the client to send a write request to the server log file; once the logging is complete, the client disconnects from the server and terminates the client program.

HERE IS THE SCREENSHOTS OF THIS WORKFLOW

acar@acar: ~/Desktop/midterm

```
acar@acar:~/Desktop/midterm$ make
gcc -c -Wall -pedantic-errors server.c
gcc server.o -o server -Wextra Queue.c directory.c
gcc -c -Wall -pedantic-errors client.c
gcc client.o -o client -Wextra
acar@acar:~/Desktop/midterm$ ./server .. 1
Server started PID 4695 /tmp/myfifo#4695
```

acar@acar: ~/Desktop/midterm

```
acar@acar:~/Desktop/midterm$ ./client connect 4695
Client started PID 4695 /tmp/myfifo#4695 /tmp/myfifo#4700
>>Waiting for Que.. connected
Connection established:
>>Please enter a command :
```

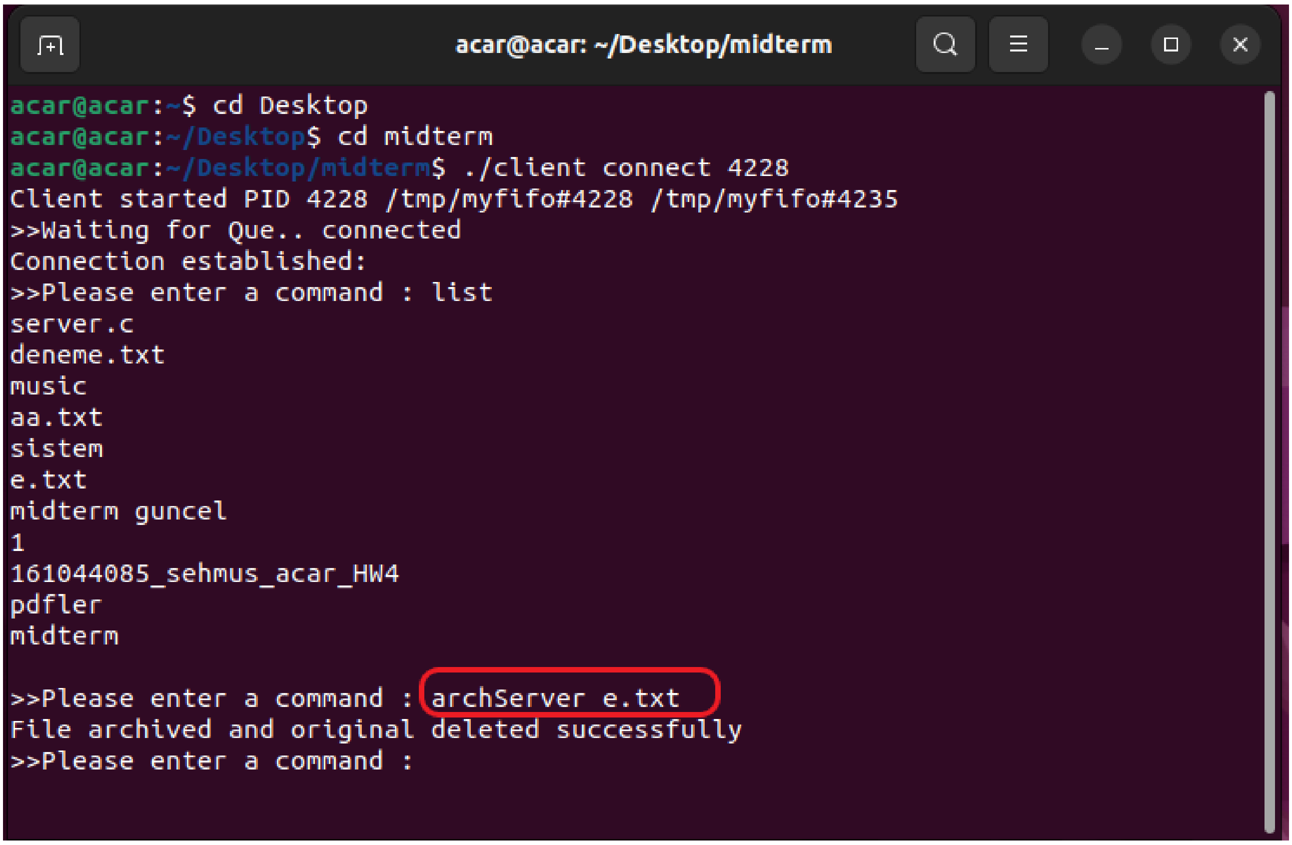
acar@acar: ~/Desktop/midterm

```
acar@acar:~/Desktop/midterm$ ./client connect 4695
Client started PID 4695 /tmp/myfifo#4695 /tmp/myfifo#4700
>>Waiting for Que.. connected
Connection established:
>>Please enter a command : list
deneme.txt
music
sistem
e.txt
midterm guncel
1
161044085_sehmus_acar_HW4
pdfler
midterm

>>Please enter a command :
```

ArchServer usage

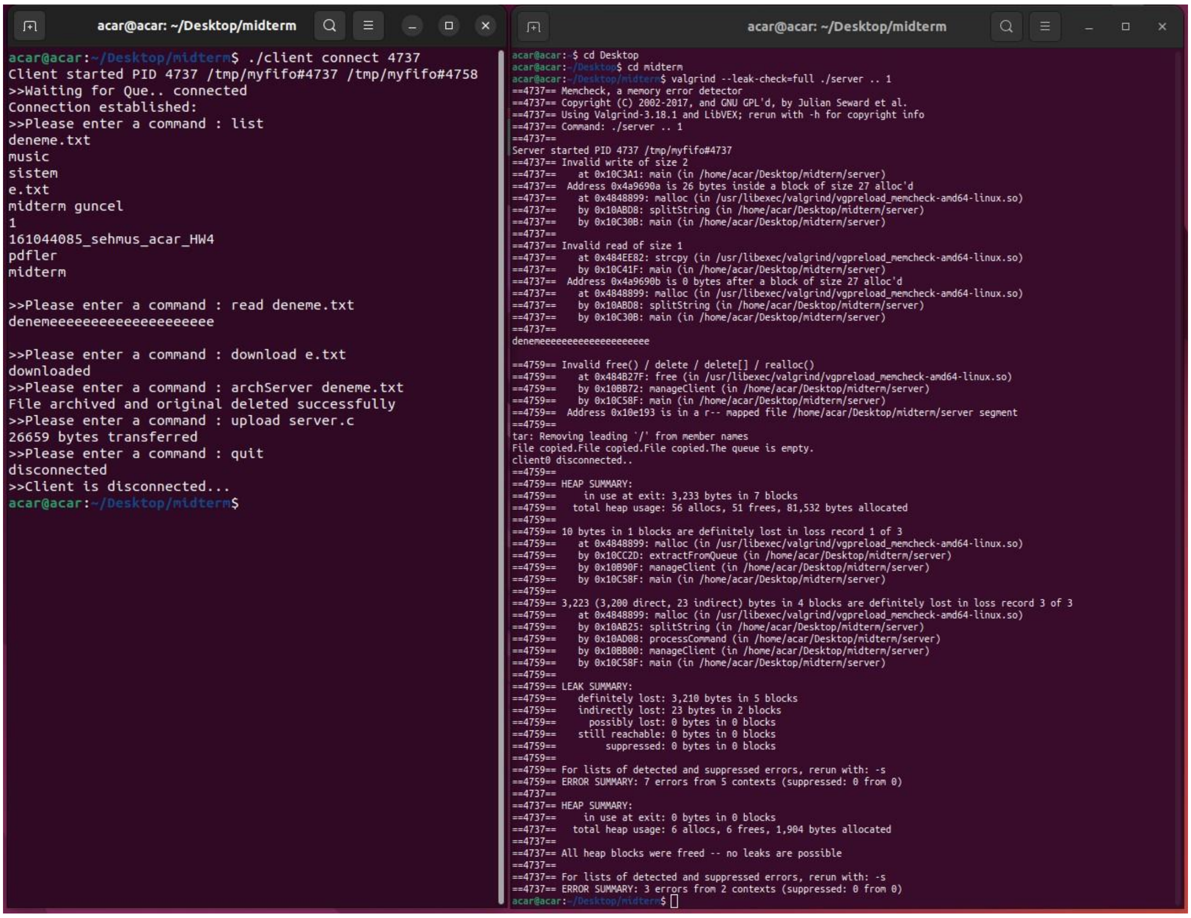
archServer comment : First downloads the file, then archives it and deletes the original file. Only the archive file remains

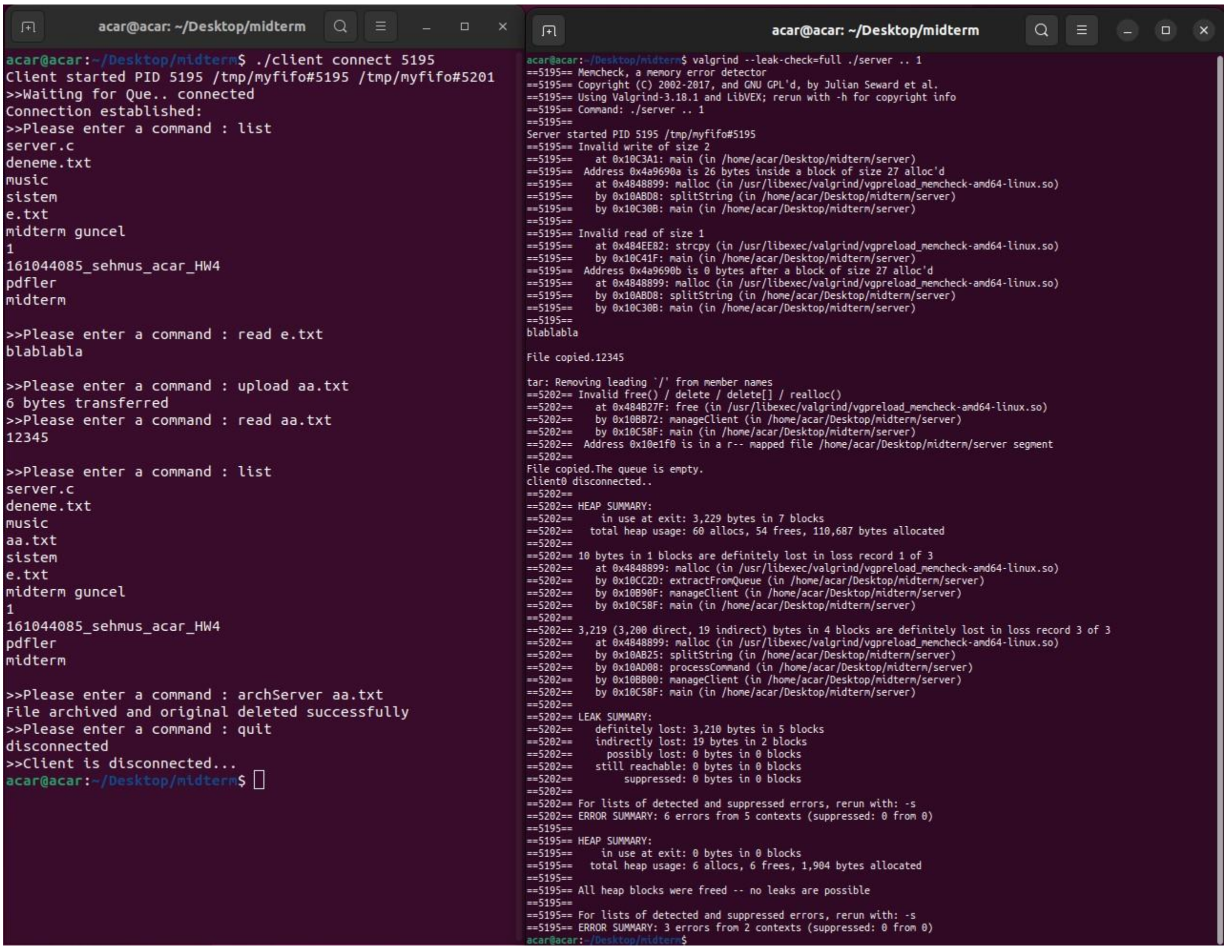


MEMORY LEAK

In my implementation, I used dynamic memory allocation extensively to manage various tasks such as tokenizing strings, executing requests, and processing files. I have also ensured that these allocations are properly handled to prevent memory leaks.

You can review the Valgrind report in the screenshots for detailed analysis.

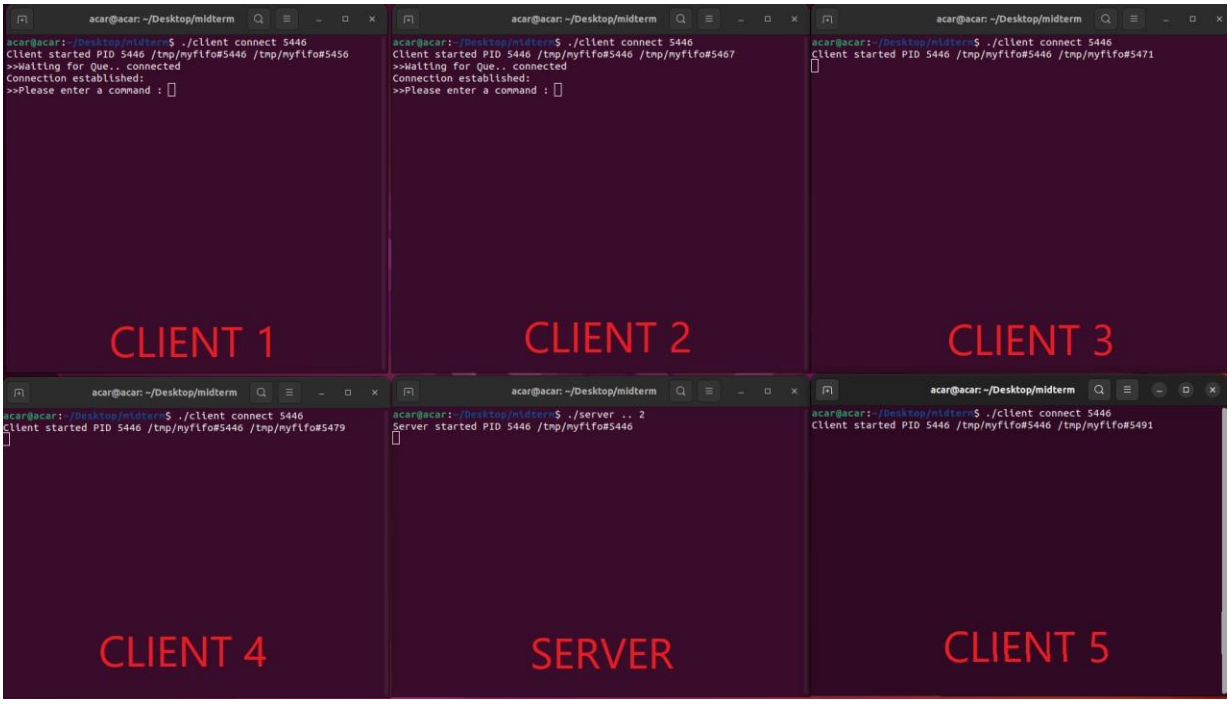



acar@acar: ~/Desktop/midterm
acar@acar:~/Desktop/midterm$ ./client connect 5195
Client started PID 5195 /tmp/myfifo#5195 /tmp/myfifo#5201
>>Waiting for Que.. connected
Connection established:
>>Please enter a command : list
server.c
deneme.txt
music
sistem
e.txt
midterm guncel
1
161044085_sehmus_acar_HW4
pdfler
midterm
>>Please enter a command : read e.txt
blablabla
>>Please enter a command : upload aa.txt
6 bytes transferred
>>Please enter a command : read aa.txt
12345
>>Please enter a command : list
server.c
deneme.txt
music
aa.txt
sistem
e.txt
midterm guncel
1
161044085_sehmus_acar_HW4
pdfler
midterm
>>Please enter a command : archServer aa.txt
File archived and original deleted successfully
>>Please enter a command : quit
disconnected
>>Client is disconnected...
acar@acar:~/Desktop/midterm$
acar@acar: ~/Desktop/midterm
acar@acar:~/Desktop/midterm$ valgrind --leak-check=full ./server .. 1
==5195== Memcheck, a memory error detector
==5195== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5195== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==5195== Command: ./server .. 1
==5195==
Server started PID 5195 /tmp/myfifo#5195
==5195== Invalid write of size 2
==5195== at 0x10C3A1: main (in /home/acar/Desktop/midterm/server)
==5195== Address 0x4a9690a is 26 bytes inside a block of size 27 alloc'd
==5195== at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==5195== by 0x10ABD8: splitString (in /home/acar/Desktop/midterm/server)
==5195== by 0x10C30B: main (in /home/acar/Desktop/midterm/server)
==5195==
==5195== Invalid read of size 1
==5195== at 0x484EE82: strcpy (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==5195== by 0x10C41F: main (in /home/acar/Desktop/midterm/server)
==5195== Address 0x4a9690b is 0 bytes after a block of size 27 alloc'd
==5195== at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==5195== by 0x10ABD8: splitString (in /home/acar/Desktop/midterm/server)
==5195== by 0x10C30B: main (in /home/acar/Desktop/midterm/server)
==5195==
blablabla
File copied.12345
tar: Removing leading `/' from member names
==5202== Invalid free() / delete / delete[] / realloc()
==5202== at 0x484B27F: free (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==5202== by 0x10BB72: manageClient (in /home/acar/Desktop/midterm/server)
==5202== by 0x10C58F: main (in /home/acar/Desktop/midterm/server)
==5202== Address 0x10e1f0 is in a r-- mapped file /home/acar/Desktop/midterm/server segment
==5202==
File copied.The queue is empty.
client0 disconnected..
==5202==
==5202== HEAP SUMMARY:
==5202== in use at exit: 3,229 bytes in 7 blocks
==5202== total heap usage: 60 allocs, 54 frees, 110,687 bytes allocated
==5202==
==5202== 10 bytes in 1 blocks are definitely lost in loss record 1 of 3
==5202== at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==5202== by 0x10CC2D: extractFromQueue (in /home/acar/Desktop/midterm/server)
==5202== by 0x10B90F: manageClient (in /home/acar/Desktop/midterm/server)
==5202== by 0x10C58F: main (in /home/acar/Desktop/midterm/server)
==5202==
==5202== 3,219 (3,200 direct, 19 indirect) bytes in 4 blocks are definitely lost in loss record 3 of 3
==5202== at 0x4848899: malloc (in /usr/libexec/valgrind/vgpreload_memcheck-amd64-linux.so)
==5202== by 0x10AB25: splitString (in /home/acar/Desktop/midterm/server)
==5202== by 0x10AD08: processCommand (in /home/acar/Desktop/midterm/server)
==5202== by 0x10BB00: manageClient (in /home/acar/Desktop/midterm/server)
==5202== by 0x10C58F: main (in /home/acar/Desktop/midterm/server)
==5202==
==5202== LEAK SUMMARY:
==5202== definitely lost: 3,210 bytes in 5 blocks
==5202== indirectly lost: 19 bytes in 2 blocks
==5202== possibly lost: 0 bytes in 0 blocks
==5202== still reachable: 0 bytes in 0 blocks
==5202== suppressed: 0 bytes in 0 blocks
==5202==
==5202== For lists of detected and suppressed errors, rerun with: -s
==5202== ERROR SUMMARY: 6 errors from 5 contexts (suppressed: 0 from 0)
==5195==
==5195== HEAP SUMMARY:
==5195== in use at exit: 0 bytes in 0 blocks
==5195== total heap usage: 6 allocs, 6 frees, 1,904 bytes allocated
==5195==
==5195== All heap blocks were freed -- no leaks are possible
==5195==
==5195== For lists of detected and suppressed errors, rerun with: -s
==5195== ERROR SUMMARY: 3 errors from 2 contexts (suppressed: 0 from 0)
acar@acar:~/Desktop/midterm$

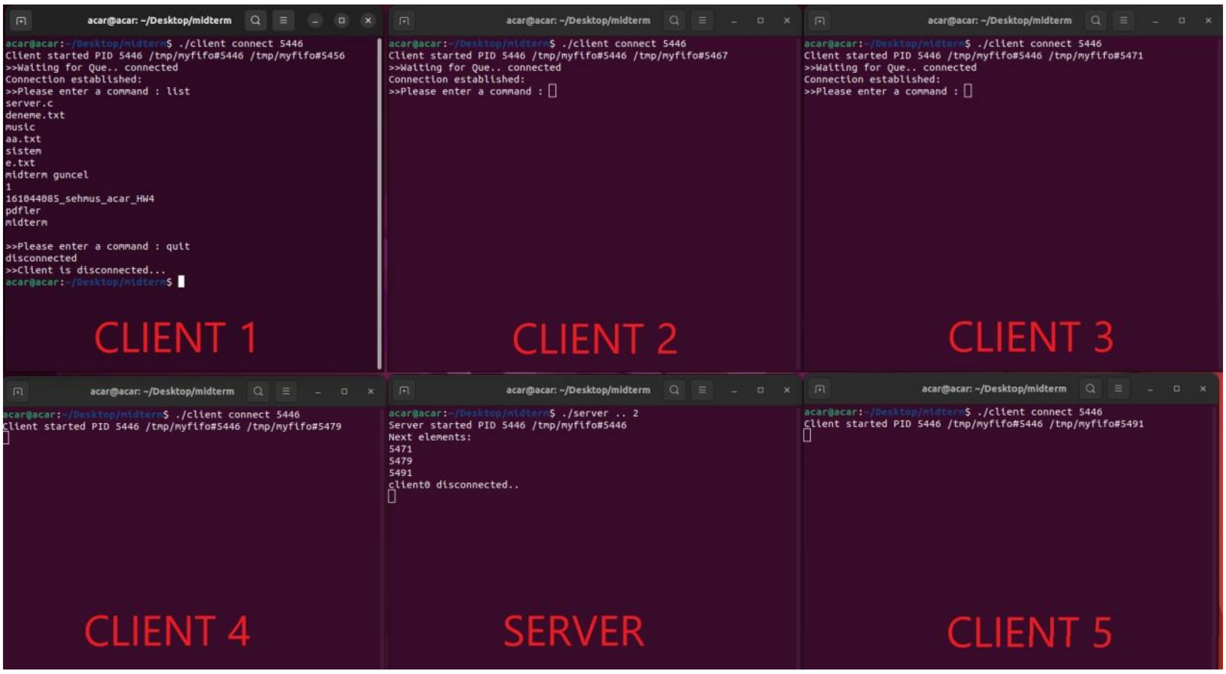
MULTIPLE CLIENTS

This app supports multiple customers and customers can wait in line.
Here is the screenshot tested with 5 customers and having only 2 customer slots.

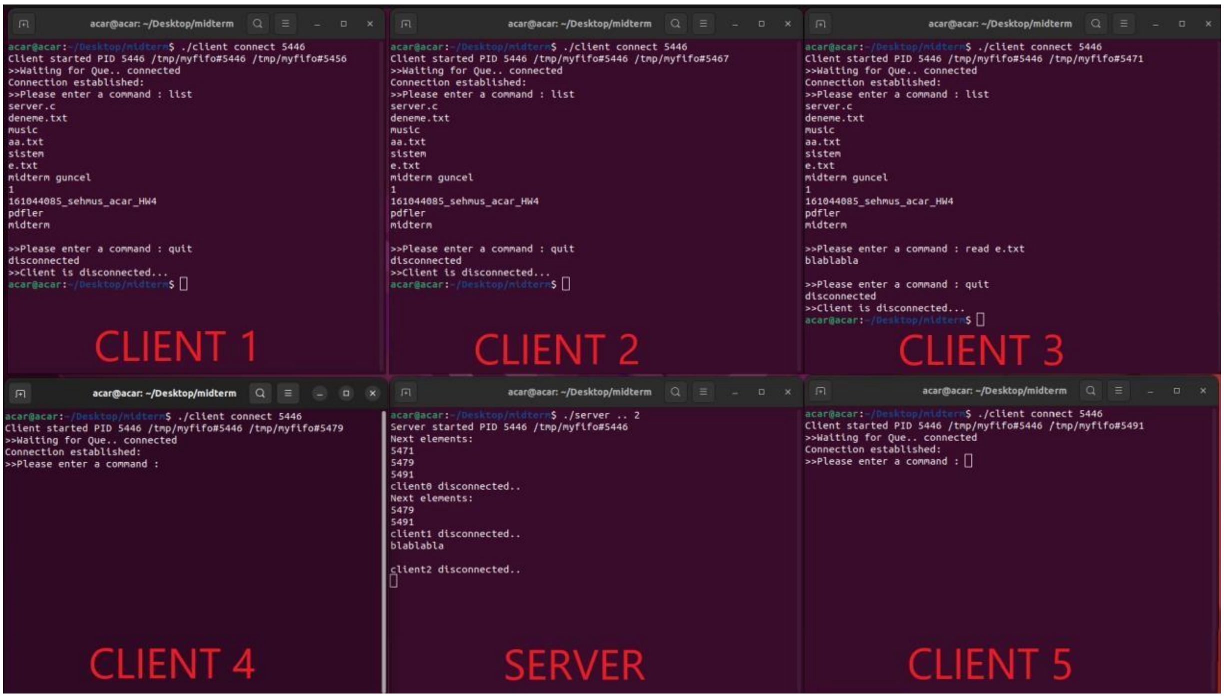
Screenshot 1



Screenshot 2



Screenshot 3



Screenshot 4