C_Shell Implementation Report: Phase 4

Introduction

This report details the upgrade of the Phase 3 server implementation of our C_Shell to incorporate scheduling capabilities. The upgraded server now simulates a scheduler by managing multiple clients' requests with a scheduling algorithm, ensuring concurrency and efficient execution of commands. The server handles two types of commands from clients: shell commands executed immediately and programs subject to scheduling based on a combined preemptive Round-Robin algorithm with Shortest Remaining Job First (SRJF) prioritization.

Usage

The upgraded C_Shell server now manages command executions based on the scheduling algorithm. Clients can connect and issue shell commands or program execution commands. The server schedules and runs the received commands accordingly.

Example Usage:

Start the upgraded server (main.c): ./server

Clients can connect to the server using the client program (client.c) and send commands: ./client

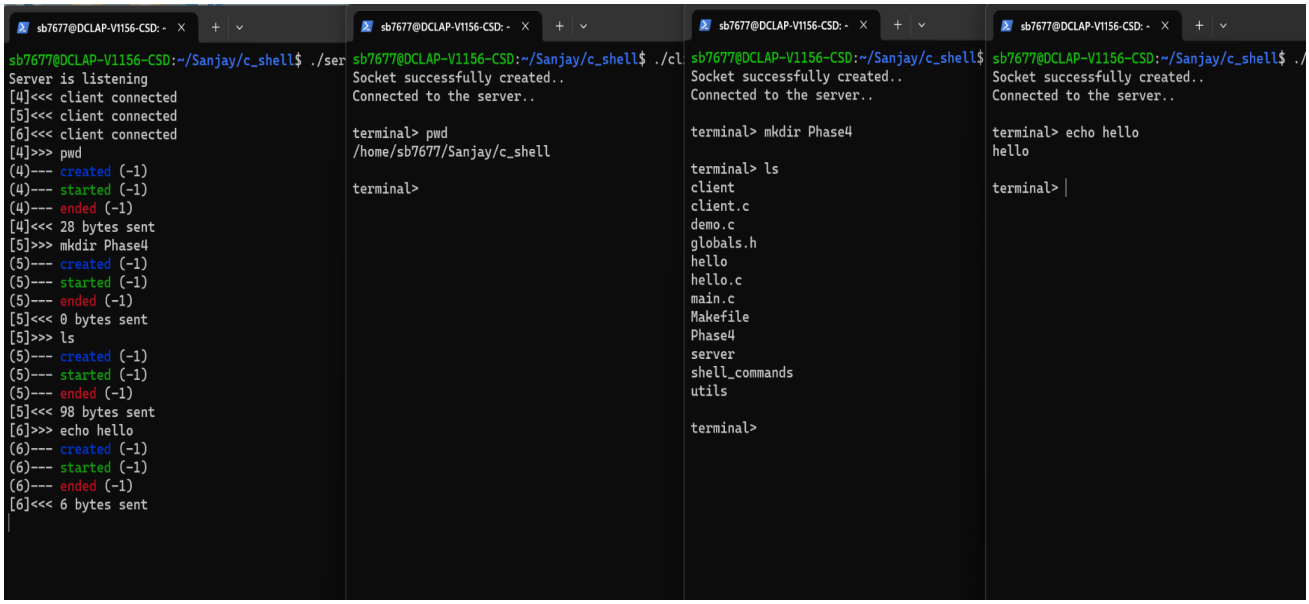
Clients can input shell commands, program execution commands with burst times, or any other program execution commands.

Test Cases and Results

Several test cases were executed to validate the functionality of the enhanced C_Shell server with scheduling capabilities:

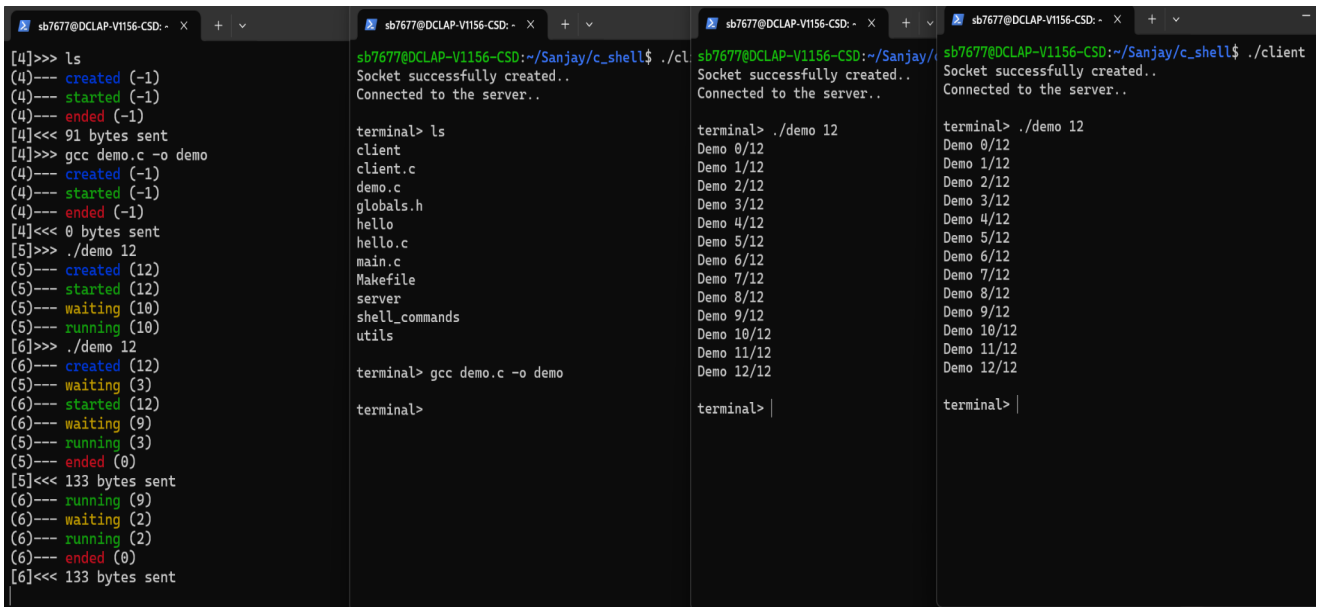
Test Scenario 1

3 concurrent clients all running single commands (since they are single commands, they should be executed without preemption)



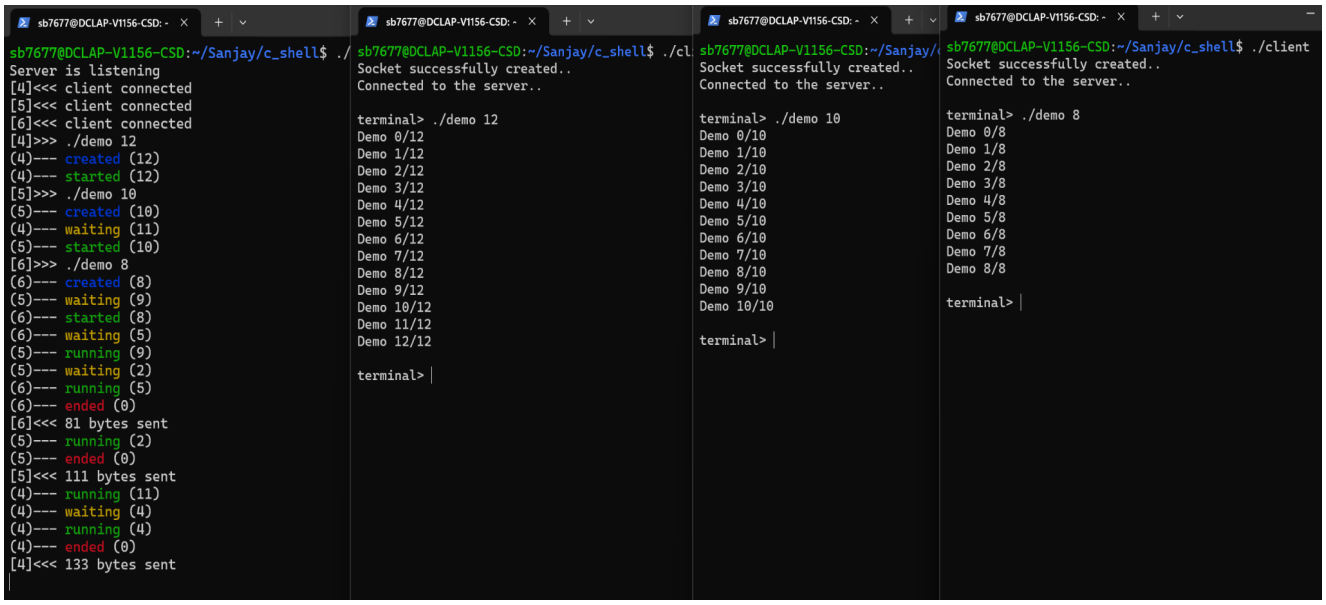
Test Scenario 2

3 concurrent clients: one client running a single command and the two others running the demo program



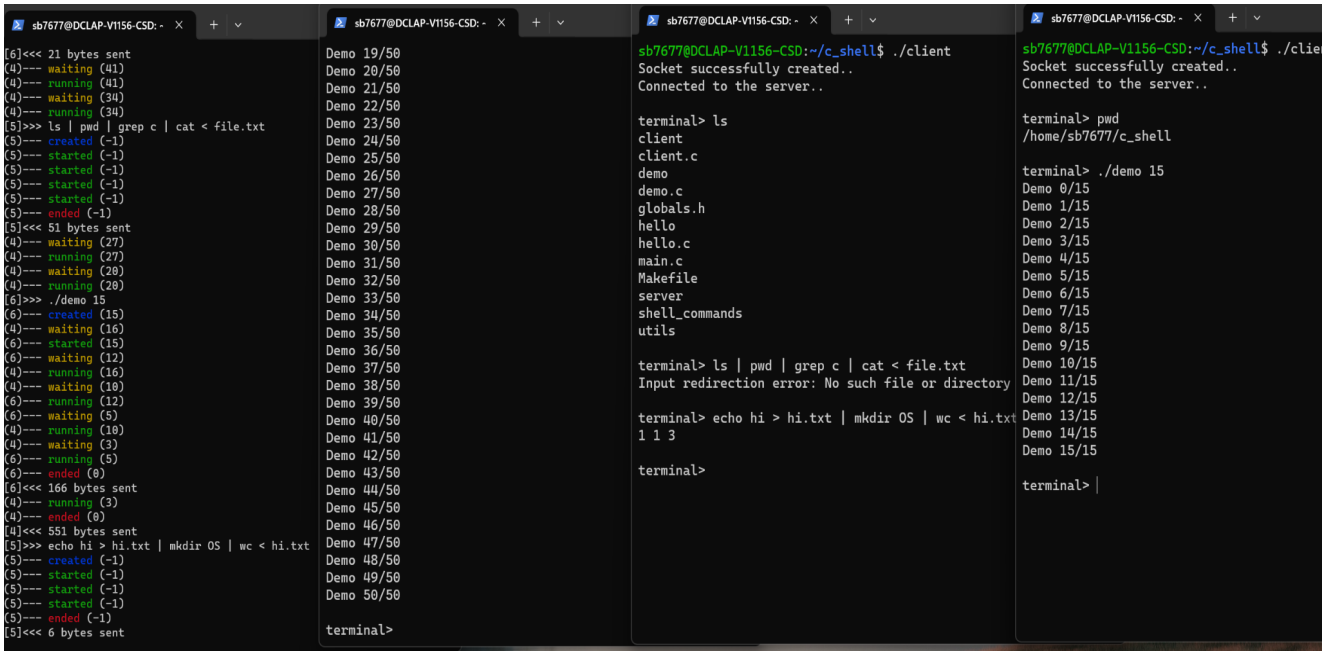
Test Scenario 3

3 concurrent clients: each client running the demo program with different values of N



Test Scenario 4

3 Concurrent clients: first client runs demo program whereas second and third will run shell commands with pipes and demo respectively



Implementation Details

The implementation of the Phase 4 server with scheduling capabilities involves the following key components and techniques:

- The server manages multiple clients and threads: a main thread for accepting new client connections, a scheduler thread for managing the scheduling algorithm, and client threads to handle individual client requests.
- A preemptive Round-Robin algorithm combined with Shortest Remaining Job First (SRJF) prioritization is implemented to schedule and prioritize commands in the waiting queue. The algorithm ensures fairness and responsiveness by allocating different quantum times for different rounds and prioritizing shorter remaining job times.
- The server handles shell commands immediately and puts program commands into the waiting queue, where they are scheduled and executed based on the defined algorithm.
- The server utilizes data structures (e.g., doubly linked list) and synchronization mechanisms such as semaphores to manage thread synchronization and handle shared resources safely.

When the server is loaded up, it listens for connections. It also starts the scheduler thread as a detached thread which continues listening for a semaphore called continue_semaphore whose value is initially 0. This semaphore is posted only when a new command (an actual program) is received from the client or when a program execution is finished. This is basically a signal indicating that the scheduler should pick a node to run execution.

The server handles multiple clients at once using threading. Once a client connects, a detached thread is created for each of them. Each node has a semaphore associated with them which determines their turn on the scheduler. By default the value is 0. Once a new node arrives when there is nothing in the queue (a doubly linked list), it parses the command, starts a child process and pauses it using kill command. Then, it waits on its semaphore to resume the child and continue the execution. It also posts the continue_semaphore so the scheduler is free to run. The scheduler just picks that node and it posts its semaphore. After the client thread consumes the semaphore, it continues execution and it again waits on a preempt_semaphore that is associated with each of the nodes. The preempt_semaphore is posted once when the timer associated with

the node (based on the quantum) is up or a new program with a smaller burst time arrives. In both these cases, the client thread uses the kill command again to stop the child process. Moreover, if the execution hasn't finished, it'll again keep waiting on its semaphore.

The detailed breakdown of the scheduling algorithm:

- The server employs a selectively preemptive Round Robin scheduling algorithm. Two different quantum times are utilized: 3 seconds for the first round and 7 seconds for subsequent rounds.
- Smaller quantum in the first round aims to prioritize shorter processes for quicker execution responses which ensures that short processes get a chance to execute in the initial rounds while also allowing time-consuming processes to run within the given quantum.
- Shortest Remaining Job First (SRJF) is utilized to prioritize processes based on the shortest remaining time to completion. The algorithm selects the process with the shortest remaining burst time to execute first. This combination allows for efficient handling of varied process lengths while optimizing overall response times.
- Shell commands are assigned a burst time of -1, ensuring they are always prioritized and executed immediately upon receipt, bypassing the queue. In cases where multiple processes have equal remaining burst times, a First-Come-First-Served (FCFS) approach is adopted, considering the order in which processes were added to the queue.
- To maintain fairness, the scheduler ensures that the same process is not selected two consecutive times if it has the shortest remaining time, except if it's the only process left in the queue.
- Program commands can be preempted if a process with a lower burst time arrives, interrupting the current running process to execute the newer, shorter burst time process. Shell commands, on the other hand, are never preempted and are run in the first round as they are received, prioritizing immediate execution.
- A round is considered complete once a process is preempted, even if the quantum is not fully utilized by the process.

C_Shell Implementation Report: Phase 3

Introduction

This report details the improvements made to the Phase 2 server implementation of our C_Shell, focusing on integrating multitasking capabilities through threads. The upgraded server can now serve multiple clients concurrently, with each thread dedicated to handling communication, computation, and requests for a specific client. This enhancement aims to improve the server's efficiency by allowing it to handle simultaneous requests from multiple clients.

Usage

The upgraded C_Shell server can now handle multiple clients simultaneously. Users can connect their client instances to the server and issue commands or requests, which the server will process independently for each client.

Example Usage:

Start the upgraded server (main.c): ./server

Clients can connect to the server using the client program (client.c) and issue commands: ./client

Multiple clients can connect and interact with the server simultaneously by issuing commands as required.

Test Cases and Results

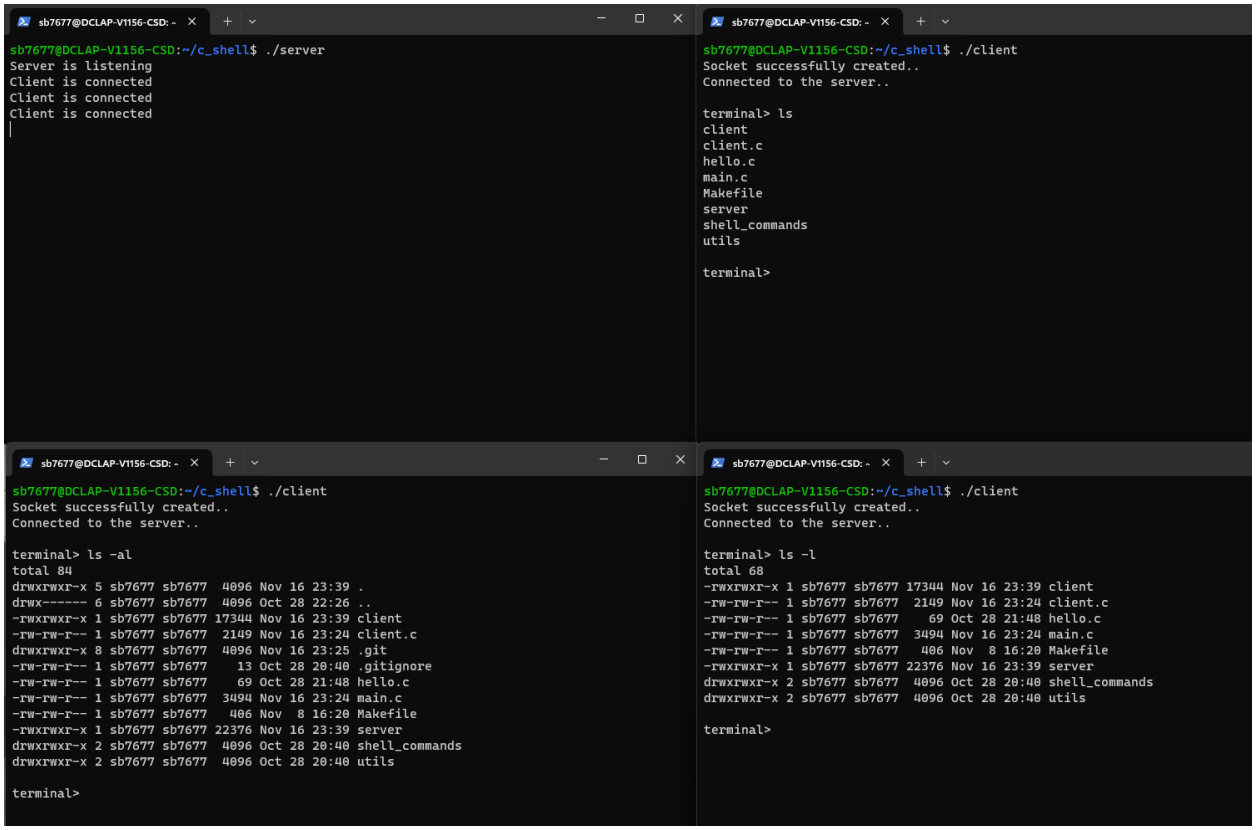
Test cases were executed to verify the server's multitasking capabilities:

Simultaneous Client Requests

Multiple clients connect to the server concurrently and issue commands simultaneously.

Expected Result: Each client's requests should be handled independently by different threads on the server.

Actual Result: The server successfully handles commands from each client simultaneously without interfering with other clients' operations.

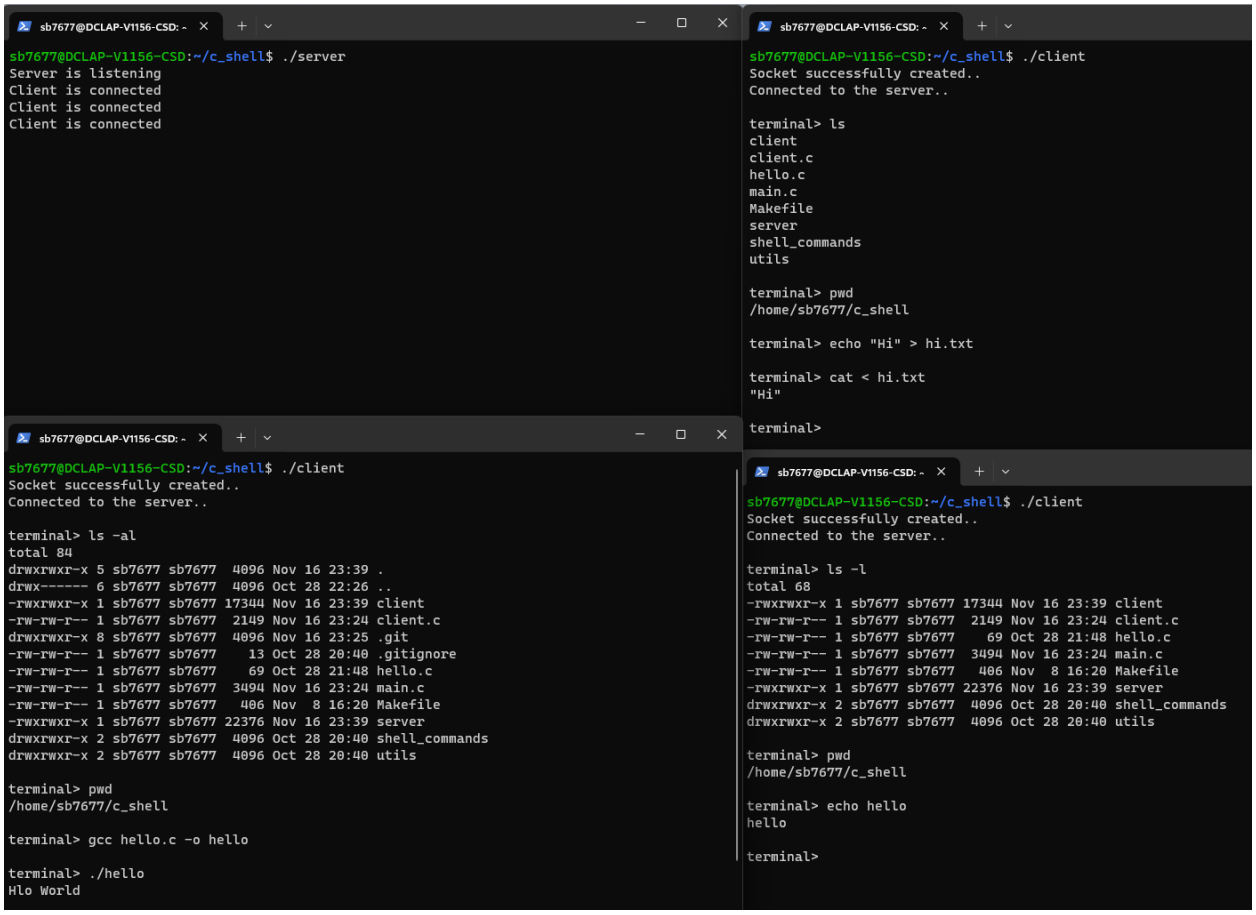


Load Testing

Multiple clients connect to the server and bombard it with multiple commands concurrently.

Expected Result: The server should efficiently handle the increased load and respond to each client without experiencing significant delays or crashing.

Actual Result: The server manages the increased load well, responding to each client's requests without noticeable performance degradation or failure.

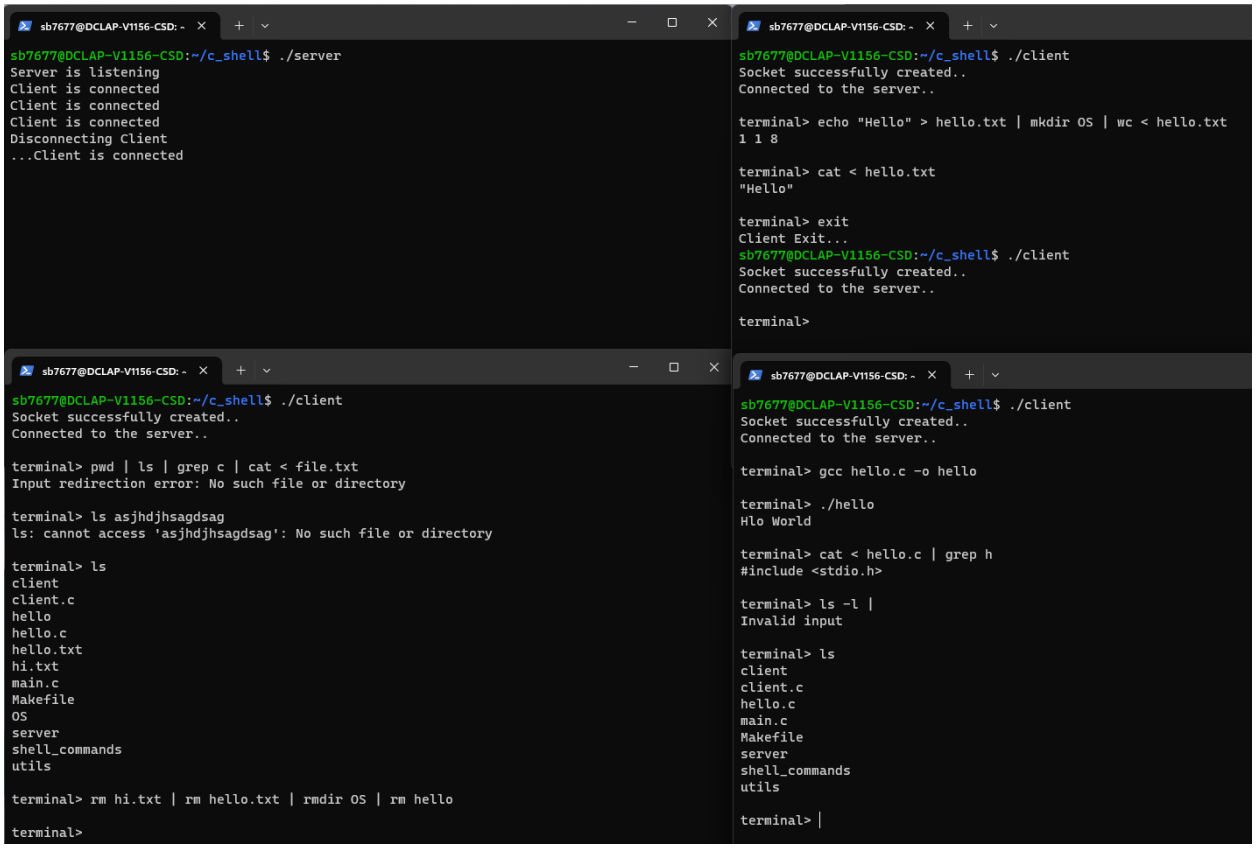


Varying Client Requests

Clients issue different types of commands, including basic commands, composed commands, file operations, pipe operations and redirection transmission.

Expected Result: The server should handle a variety of commands from different clients concurrently without conflicts or data corruption.

Actual Result: The server successfully processes diverse commands from multiple clients simultaneously.



Implementation Details

The implementation of the Phase 3 server with multitasking capabilities involves the following key components and techniques:

- The server uses threads to handle multiple clients concurrently. Upon each client's connection, a new thread is created to manage communication and computation exclusively for that client.
- The server maintains socket communication with each client, handling commands, execution, and result transmission individually for every connected client.
- Similar techniques from Phase 1 and 2 are used to execute commands received from clients, ensuring proper handling of fork(), exec(), and I/O redirection within each thread.

C_Shell Implementation Report: Phase 2

Introduction

In this report, we discuss the implementation and upgrade of our C_Shell from Phase 1 to Phase 2, which introduces remote access capabilities through Socket communication. The C_Shell now consists of a client-server architecture, where the client takes user inputs and sends requests to the server for execution. The server executes the requested commands and returns the results to the client. We have created a client.c file for the client-side implementation and kept the original main.c as the server.

Usage

The upgraded C_Shell now supports remote access, allowing users to interact with the server through the client. The user can input commands, composed commands, or programs to execute via the client. The server processes these requests and sends back the output or result, which is displayed on the client's screen.

Test Cases and Results

We conducted test cases to validate the functionality of the C_Shell with remote access. Below are some key test cases and their results:

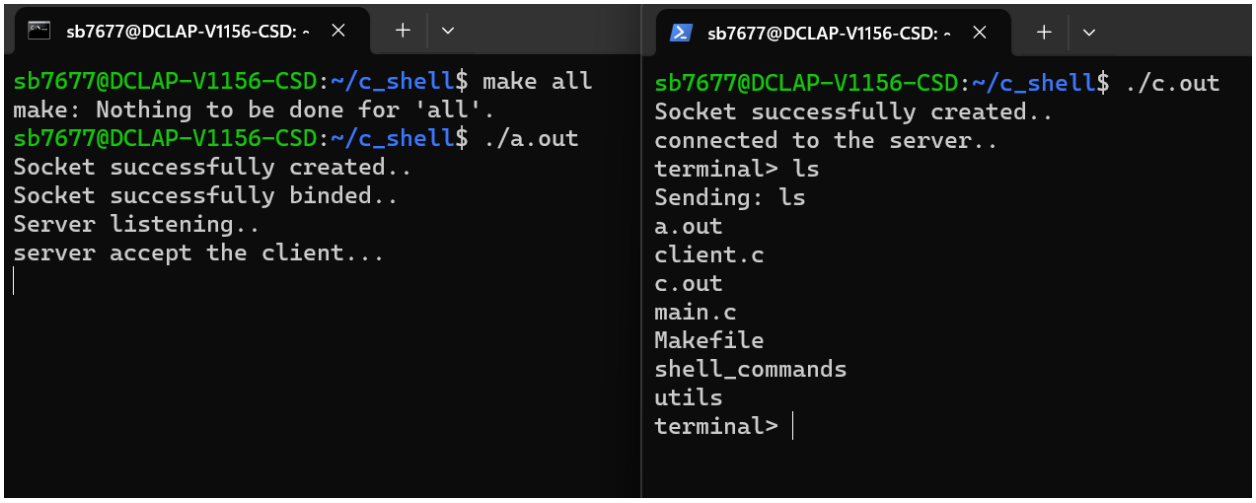
- On the server (main.c), execute the server program: ./a.out
- On the client (client.c), connect to the server: ./c.out

Basic Command Execution

Client Input: ls

Expected Result: The client should send the 'ls' command to the server, and the server should execute it. The client should receive and display the directory listing.

Actual Result: The command executed on the server, and the client received and displayed the directory listing.

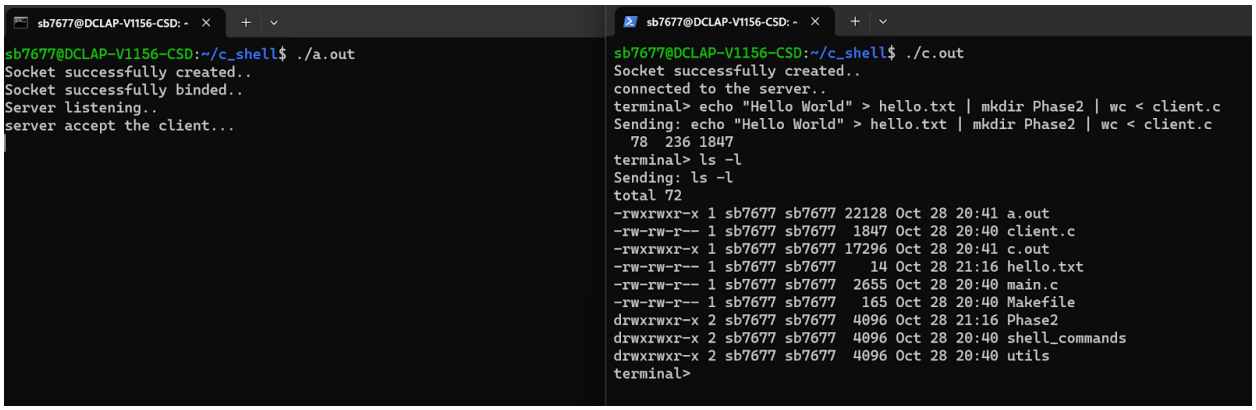


Redirection and Pipe Operators

Client Input: *composed commands*

Expected Result: The client should send the composed commands to the server, and the server should execute it. The client should receive and display the last command in a pipe.

Actual Result: The command executed on the server, and the client received and displayed the results.

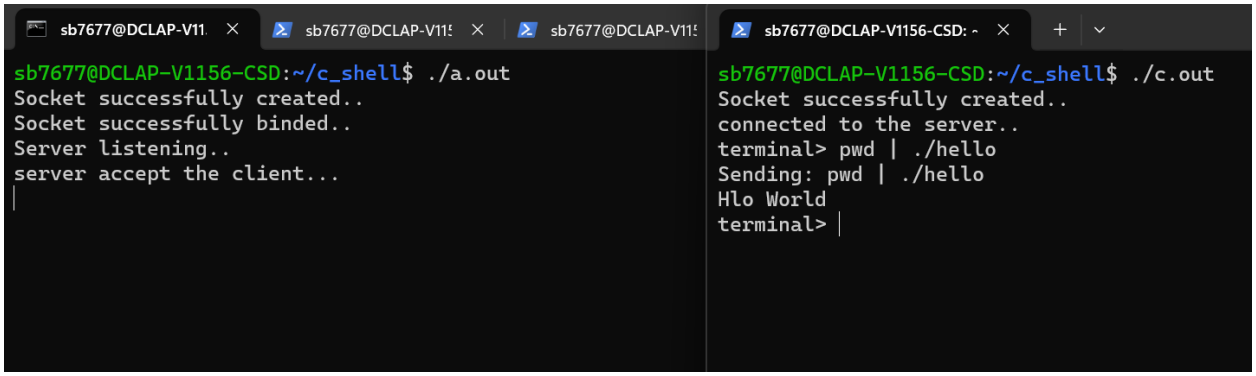


Executable File

Client Input: execute executable/output file

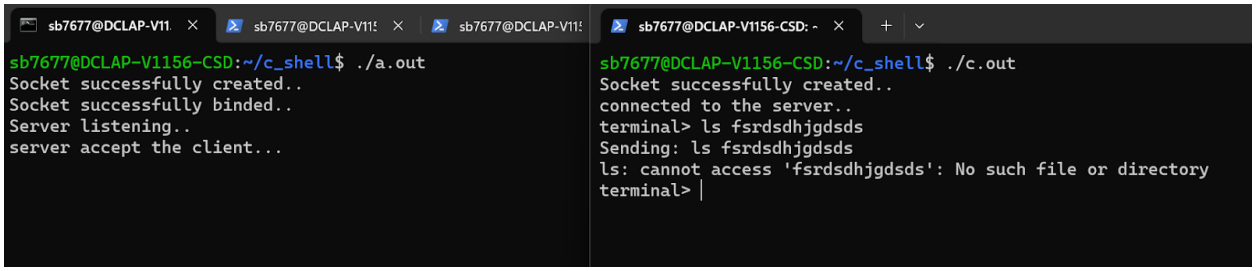
Expected Result: Print the output of the file

Actual Result: The command executed on the server, and the client received and displayed the output of the file.



Error Fixation

Stderr transmission



Implementation Details

The implementation of the C_Shell with remote access involves the following components and techniques:

- We have implemented socket communication using the socket(), bind(), listen(), and accept() functions on the server-side and socket(), connect(), and send()/recv() functions on the client-side.

- The server processes user commands using techniques similar to those in Phase 1. It uses fork(), exec(), wait(), and dup2() to execute commands and handle redirection.
- The client accepts user input and sends it to the server for execution. The server sends the results back to the client, which then displays them on the screen. A command 'exit' is implemented for termination.

C_Shell Implementation Report: Phase 1

Introduction

In this report, we describe the implementation of a custom C_Shell program. The C_Shell is designed to take Linux commands as input from the user and execute them. We have employed various system calls and techniques, such as fork(), exec(), wait(), dup2(), and pipe(), to create a functional shell. This report provides details about the program's usage, test cases, and an overview of the implementation.

Usage

The C_Shell provides a command-line interface that allows users to enter Linux commands. The shell supports basic command execution and redirection, as well as pipes for connecting multiple commands in a pipeline. To execute a command, users simply input it and press Enter. The shell will then execute the command and return the result.

Test Cases and Results

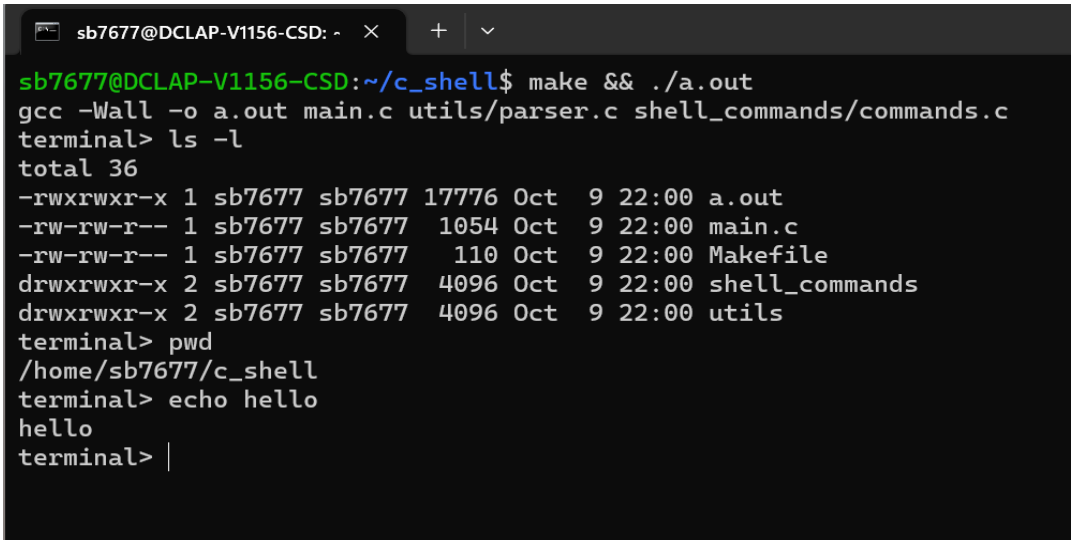
Below are some test cases to verify the functionality of our C_Shell.

Basic Command Execution

Input: ls -l

Expected Result: The shell should execute the 'ls -l' command and display the directory listing.

Actual Result: The command executed successfully, and the directory listing was displayed.

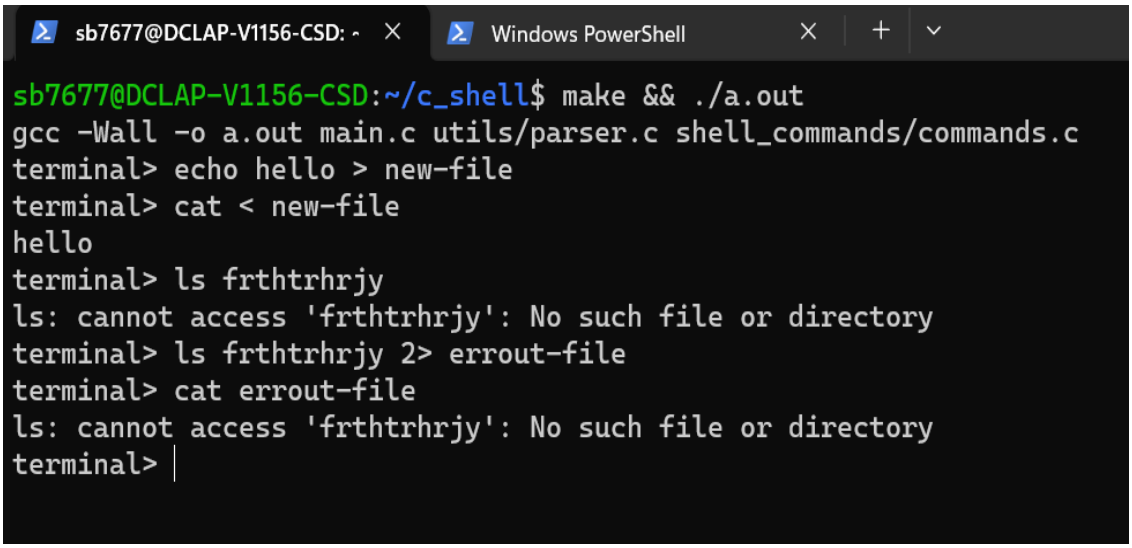


Redirection

Input: cat file.txt > output.txt

Expected Result: The shell should redirect the contents of 'file.txt' to 'output.txt'.

Actual Result: Redirection worked as expected, and the contents were copied to 'output.txt'.

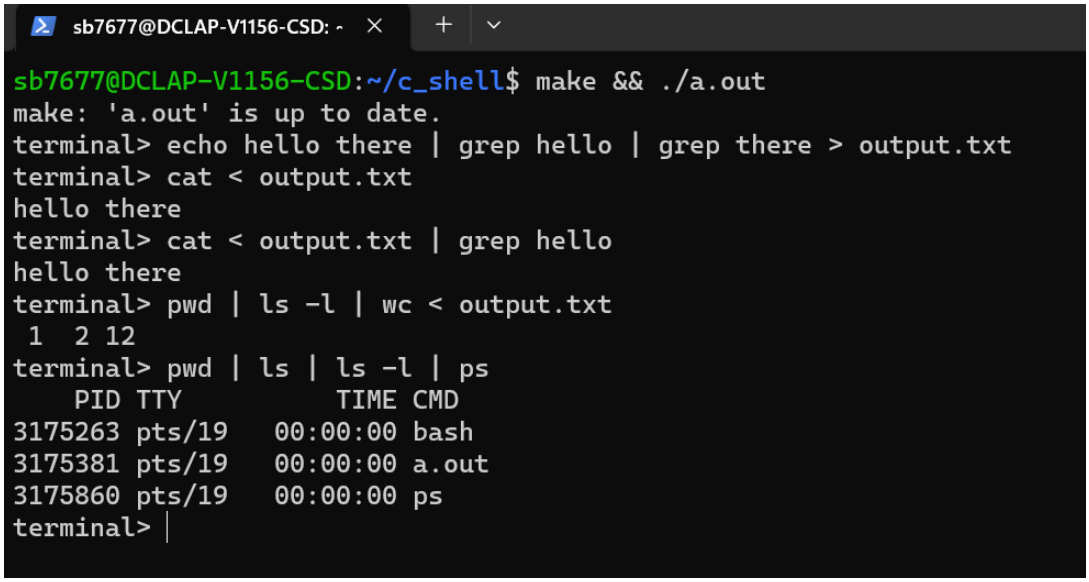


Pipe Operator

Input: cat output.txt | grep hello | grep there

Expected Result: The shell should pipe the output of 'cat output.txt' to 'grep hello'.

Actual Result: The pipe operator correctly connected the two commands, and 'grep' filtered the lines containing the specified keyword.

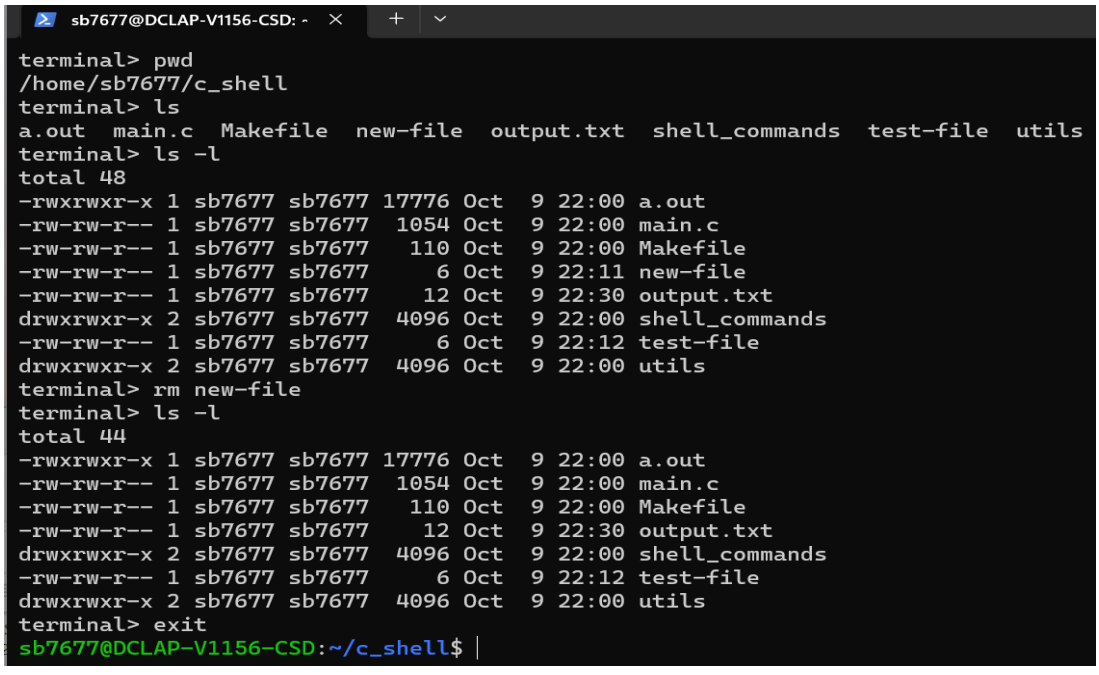


Exit Command

Input: exit

Expected Result: The shell should exit gracefully.

Actual Result: The shell terminated as expected, returning control to the system.



Implementation Details

The C_Shell is implemented as follows:

- Each command is executed in an individual child process spawned from the main process using the fork() system call.
- The execvp() system call is used to execute the user-entered command with its arguments.
- Input and output are redirected to files using the < and > operators, achieved through the dup2() system call.
- The pipe() system call is used to connect the standard output of one command to the standard input of another when the pipe symbol | is used. The shell ensures that the next process/command waits for the last process in the pipeline to terminate before proceeding.
- The shell employs a while loop to continuously prompt the user for input until the 'exit' command is entered.

List of fifteen commands:

- pwd
- ls (all arguments)
- ps (all arguments)
- clear
- cat
- touch
- rm (all arguments)
- echo
- mkdir
- grep
- wc
- curl (www.google.com)
- nslookup (www.google.com)
- chmod
- env
- whoami
- tty