

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)

```
sb7677@DCLAP-V1156-CSD: - X + v
sb7677@DCLAP-V1156-CSD:~/Sanjay/c_shell$ ./server
Server is listening
[4]<<< client connected
[5]<<< client connected
[6]<<< client connected
[4]>>> pwd
(4)--- created (-1)
(4)--- started (-1)
(4)--- ended (-1)
[4]<<< 28 bytes sent
[5]>>> mkdir Phase4
(5)--- created (-1)
(5)--- started (-1)
(5)--- ended (-1)
[5]<<< 0 bytes sent
[5]>>> ls
(5)--- created (-1)
(5)--- started (-1)
(5)--- ended (-1)
[5]<<< 98 bytes sent
[6]>>> echo hello
(6)--- created (-1)
(6)--- started (-1)
(6)--- ended (-1)
[6]<<< 6 bytes sent

sb7677@DCLAP-V1156-CSD:~/Sanjay/c_shell$ ./client
Socket successfully created..
Connected to the server..

terminal> pwd
/home/sb7677/Sanjay/c_shell

terminal>

sb7677@DCLAP-V1156-CSD:~/Sanjay/c_shell$
Socket successfully created..
Connected to the server..

terminal> mkdir Phase4

terminal> ls
client
client.c
demo.c
globals.h
hello
hello.c
main.c
Makefile
Phase4
server
shell_commands
utils

terminal>

sb7677@DCLAP-V1156-CSD:~/Sanjay/c_shell$ ./client
Socket successfully created..
Connected to the server..

terminal> echo hello
hello

terminal> |
```

Test Scenario 2

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

```
sb7677@DCLAP-V1156-CSD: - X + v
[4]>>> ls
(4)--- created (-1)
(4)--- started (-1)
(4)--- ended (-1)
[4]<<< 91 bytes sent
[4]>>> gcc demo.c -o demo
(4)--- created (-1)
(4)--- started (-1)
(4)--- ended (-1)
[4]<<< 0 bytes sent
[5]>>> ./demo 12
(5)--- created (12)
(5)--- started (12)
(5)--- waiting (10)
(5)--- running (10)
[6]>>> ./demo 12
(6)--- created (12)
(6)--- waiting (3)
(6)--- started (12)
(6)--- waiting (9)
(6)--- running (3)
(6)--- ended (0)
[5]<<< 133 bytes sent
(6)--- running (9)
(6)--- waiting (2)
(6)--- running (2)
(6)--- ended (0)
[6]<<< 133 bytes sent

sb7677@DCLAP-V1156-CSD:~/Sanjay/c_shell$ ./client
Socket successfully created..
Connected to the server..

terminal> ls
client
client.c
demo.c
globals.h
hello
hello.c
main.c
Makefile
server
shell_commands
utils

terminal> gcc demo.c -o demo

terminal>

sb7677@DCLAP-V1156-CSD:~/Sanjay/c_shell$ ./client
Socket successfully created..
Connected to the server..

terminal> ./demo 12
Demo 0/12
Demo 1/12
Demo 2/12
Demo 3/12
Demo 4/12
Demo 5/12
Demo 6/12
Demo 7/12
Demo 8/12
Demo 9/12
Demo 10/12
Demo 11/12
Demo 12/12

terminal> |

sb7677@DCLAP-V1156-CSD:~/Sanjay/c_shell$ ./client
Socket successfully created..
Connected to the server..

terminal> ./demo 12
Demo 0/12
Demo 1/12
Demo 2/12
Demo 3/12
Demo 4/12
Demo 5/12
Demo 6/12
Demo 7/12
Demo 8/12
Demo 9/12
Demo 10/12
Demo 11/12
Demo 12/12

terminal> |
```

Test Scenario 3

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

```
sb7677@DCLAP-V1156-CSD: - x + v sb7677@DCLAP-V1156-CSD: - x + v sb7677@DCLAP-V1156-CSD: - x + v sb7677@DCLAP-V1156-CSD: - x + v
sb7677@DCLAP-V1156-CSD:~/Sanjay/c_shell$ ./
Server is listening
[4]<<< client connected
[5]<<< client connected
[6]<<< client connected
[4]>>> ./demo 12
(4)--- created (12)
(4)--- started (12)
[5]>>> ./demo 10
(5)--- created (10)
(4)--- waiting (11)
(5)--- started (10)
[6]>>> ./demo 8
(6)--- created (8)
(5)--- waiting (9)
(6)--- started (8)
(6)--- waiting (5)
(5)--- running (9)
(5)--- waiting (2)
(6)--- running (5)
(6)--- ended (0)
[6]<<< 81 bytes sent
(5)--- running (2)
(5)--- ended (0)
[5]<<< 111 bytes sent
(4)--- running (11)
(4)--- waiting (4)
(4)--- running (4)
(4)--- ended (0)
[4]<<< 133 bytes sent

sb7677@DCLAP-V1156-CSD:~/Sanjay/c_shell$ ./cl
Socket successfully created..
Connected to the server..

terminal> ./demo 12
Demo 0/12
Demo 1/12
Demo 2/12
Demo 3/12
Demo 4/12
Demo 5/12
Demo 6/12
Demo 7/12
Demo 8/12
Demo 9/12
Demo 10/12
Demo 11/12
Demo 12/12

terminal> |

sb7677@DCLAP-V1156-CSD:~/Sanjay/c_shell$ ./cl
Socket successfully created..
Connected to the server..

terminal> ./demo 10
Demo 0/10
Demo 1/10
Demo 2/10
Demo 3/10
Demo 4/10
Demo 5/10
Demo 6/10
Demo 7/10
Demo 8/10
Demo 9/10
Demo 10/10

terminal> |

sb7677@DCLAP-V1156-CSD:~/Sanjay/c_shell$ ./client
Socket successfully created..
Connected to the server..

terminal> ./demo 8
Demo 0/8
Demo 1/8
Demo 2/8
Demo 3/8
Demo 4/8
Demo 5/8
Demo 6/8
Demo 7/8
Demo 8/8

terminal> |
```

Test Scenario 4

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

```
sb7677@DCLAP-V1156-CSD: - x + v sb7677@DCLAP-V1156-CSD: - x + v sb7677@DCLAP-V1156-CSD: - x + v sb7677@DCLAP-V1156-CSD: - x + v
sb7677@DCLAP-V1156-CSD:~/Sanjay/c_shell$ ./
Server is listening
[4]<<< client connected
[5]<<< client connected
[6]<<< client connected
[4]>>> ls | pwd | grep c | cat < file.txt
(4)--- waiting (41)
(4)--- running (41)
(4)--- waiting (34)
(4)--- running (34)
[5]>>> ./demo 15
(5)--- created (-1)
(5)--- started (-1)
(5)--- started (-1)
(5)--- started (-1)
(5)--- started (-1)
(5)--- ended (-1)
[5]<<< 51 bytes sent
(4)--- waiting (27)
(4)--- running (27)
(4)--- waiting (20)
(4)--- running (20)
[6]>>> ./demo 15
(6)--- created (15)
(6)--- waiting (16)
(6)--- started (15)
(6)--- waiting (12)
(4)--- running (16)
(4)--- waiting (10)
(6)--- running (12)
(6)--- waiting (5)
(4)--- running (10)
(4)--- waiting (3)
(6)--- running (5)
(6)--- ended (0)
[6]<<< 166 bytes sent
(4)--- running (3)
(4)--- ended (0)
[4]<<< 551 bytes sent
[5]>>> echo hi > hi.txt | mkdir OS | wc < hi.txt
(5)--- created (-1)
(5)--- started (-1)
(5)--- started (-1)
(5)--- started (-1)
(5)--- ended (-1)
[5]<<< 6 bytes sent

sb7677@DCLAP-V1156-CSD:~/Sanjay/c_shell$ ./cl
Socket successfully created..
Connected to the server..

terminal> ls
client
client.c
demo
demo.c
globals.h
hello
hello.c
main.c
Makefile
server
shell_commands
utils

terminal> ls | pwd | grep c | cat < file.txt
Input redirection error: No such file or directory

terminal> echo hi > hi.txt | mkdir OS | wc < hi.txt
1 1 3

terminal>

sb7677@DCLAP-V1156-CSD:~/Sanjay/c_shell$ ./client
Socket successfully created..
Connected to the server..

terminal> pwd
/home/sb7677/c_shell

terminal> ./demo 15
Demo 0/15
Demo 1/15
Demo 2/15
Demo 3/15
Demo 4/15
Demo 5/15
Demo 6/15
Demo 7/15
Demo 8/15
Demo 9/15
Demo 10/15
Demo 11/15
Demo 12/15
Demo 13/15
Demo 14/15
Demo 15/15

terminal> |
```

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.

```
sb7677@DCLAP-V1156-CSD: ~ - c_shell$ ./server
Server is listening
Client is connected
Client is connected
Client is connected

sb7677@DCLAP-V1156-CSD: ~ - c_shell$ ./client
Socket successfully created..
Connected to the server..

terminal> ls
client
client.c
hello.c
main.c
Makefile
server
shell_commands
utils

terminal>

sb7677@DCLAP-V1156-CSD: ~ - c_shell$ ./client
Socket successfully created..
Connected to the server..

terminal> ls -al
total 84
drwxrwxr-x 5 sb7677 sb7677 4096 Nov 16 23:39 .
drwx----- 6 sb7677 sb7677 4096 Oct 28 22:26 ..
-rwxrwxr-x 1 sb7677 sb7677 17344 Nov 16 23:39 client
-rw-rw-r-- 1 sb7677 sb7677 2149 Nov 16 23:24 client.c
drwxrwxr-x 8 sb7677 sb7677 4096 Nov 16 23:25 .git
-rw-rw-r-- 1 sb7677 sb7677 13 Oct 28 20:40 .gitignore
-rw-rw-r-- 1 sb7677 sb7677 69 Oct 28 21:48 hello.c
-rw-rw-r-- 1 sb7677 sb7677 3494 Nov 16 23:24 main.c
-rw-rw-r-- 1 sb7677 sb7677 406 Nov 8 16:20 Makefile
-rwxrwxr-x 1 sb7677 sb7677 22376 Nov 16 23:39 server
drwxrwxr-x 2 sb7677 sb7677 4096 Oct 28 20:40 shell_commands
drwxrwxr-x 2 sb7677 sb7677 4096 Oct 28 20:40 utils

terminal>

sb7677@DCLAP-V1156-CSD: ~ - c_shell$ ./client
Socket successfully created..
Connected to the server..

terminal> ls -l
total 68
-rwxrwxr-x 1 sb7677 sb7677 17344 Nov 16 23:39 client
-rw-rw-r-- 1 sb7677 sb7677 2149 Nov 16 23:24 client.c
-rw-rw-r-- 1 sb7677 sb7677 69 Oct 28 21:48 hello.c
-rw-rw-r-- 1 sb7677 sb7677 3494 Nov 16 23:24 main.c
-rw-rw-r-- 1 sb7677 sb7677 406 Nov 8 16:20 Makefile
-rwxrwxr-x 1 sb7677 sb7677 22376 Nov 16 23:39 server
drwxrwxr-x 2 sb7677 sb7677 4096 Oct 28 20:40 shell_commands
drwxrwxr-x 2 sb7677 sb7677 4096 Oct 28 20:40 utils

terminal>
```

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.

```
sb7677@DCLAP-V1156-CSD: - X + v
sb7677@DCLAP-V1156-CSD:~/c_shell$ ./server
Server is listening
Client is connected
Client is connected
Client is connected

sb7677@DCLAP-V1156-CSD:~/c_shell$ ./client
Socket successfully created..
Connected to the server..

terminal> ls
client
client.c
hello.c
main.c
Makefile
server
shell_commands
utils

terminal> pwd
/home/sb7677/c_shell

terminal> echo "Hi" > hi.txt

terminal> cat < hi.txt
"Hi"

terminal>

sb7677@DCLAP-V1156-CSD: - X + v
sb7677@DCLAP-V1156-CSD:~/c_shell$ ./server
Server is listening
Client is connected
Client is connected
Client is connected

sb7677@DCLAP-V1156-CSD:~/c_shell$ ./client
Socket successfully created..
Connected to the server..

terminal> ls -al
total 84
drwxrwxr-x 5 sb7677 sb7677 4096 Nov 16 23:39 .
drwx----- 6 sb7677 sb7677 4096 Oct 28 22:26 ..
-rwxrwxr-x 1 sb7677 sb7677 17344 Nov 16 23:39 client
-rw-rw-r-- 1 sb7677 sb7677 2149 Nov 16 23:24 client.c
drwxrwxr-x 8 sb7677 sb7677 4096 Nov 16 23:25 .git
-rw-rw-r-- 1 sb7677 sb7677 13 Oct 28 20:48 .gitignore
-rw-rw-r-- 1 sb7677 sb7677 69 Oct 28 21:48 hello.c
-rw-rw-r-- 1 sb7677 sb7677 3494 Nov 16 23:24 main.c
-rw-rw-r-- 1 sb7677 sb7677 406 Nov 8 16:20 Makefile
-rwxrwxr-x 1 sb7677 sb7677 22376 Nov 16 23:39 server
drwxrwxr-x 2 sb7677 sb7677 4096 Oct 28 20:48 shell_commands
drwxrwxr-x 2 sb7677 sb7677 4096 Oct 28 20:48 utils

terminal> pwd
/home/sb7677/c_shell

terminal> gcc hello.c -o hello

terminal> ./hello
Hlo World

sb7677@DCLAP-V1156-CSD: - X + v
sb7677@DCLAP-V1156-CSD:~/c_shell$ ./client
Socket successfully created..
Connected to the server..

terminal> ls -l
total 68
-rwxrwxr-x 1 sb7677 sb7677 17344 Nov 16 23:39 client
-rw-rw-r-- 1 sb7677 sb7677 2149 Nov 16 23:24 client.c
-rw-rw-r-- 1 sb7677 sb7677 69 Oct 28 21:48 hello.c
-rw-rw-r-- 1 sb7677 sb7677 3494 Nov 16 23:24 main.c
-rw-rw-r-- 1 sb7677 sb7677 406 Nov 8 16:20 Makefile
-rwxrwxr-x 1 sb7677 sb7677 22376 Nov 16 23:39 server
drwxrwxr-x 2 sb7677 sb7677 4096 Oct 28 20:48 shell_commands
drwxrwxr-x 2 sb7677 sb7677 4096 Oct 28 20:48 utils

terminal> pwd
/home/sb7677/c_shell

terminal> echo hello
hello

terminal>
```

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.


```
sb7677@DCLAP-V1156-CSD: ~ - x + v
sb7677@DCLAP-V1156-CSD:~/c_shell$ ./server
Server is listening
Client is connected
Client is connected
Client is connected
Disconnecting Client
...Client is connected

sb7677@DCLAP-V1156-CSD: ~ - x + v
sb7677@DCLAP-V1156-CSD:~/c_shell$ ./client
Socket successfully created..
Connected to the server..

terminal> echo "Hello" > hello.txt | mkdir OS | wc < hello.txt
1 1 8

terminal> cat < hello.txt
"Hello"

terminal> exit
Client Exit...

sb7677@DCLAP-V1156-CSD:~/c_shell$ ./client
Socket successfully created..
Connected to the server..

terminal>

sb7677@DCLAP-V1156-CSD: ~ - x + v
sb7677@DCLAP-V1156-CSD:~/c_shell$ ./client
Socket successfully created..
Connected to the server..

terminal> pwd | ls | grep c | cat < file.txt
Input redirection error: No such file or directory

terminal> ls asjhdjhsagdsag
ls: cannot access 'asjhdjhsagdsag': No such file or directory

terminal> ls
client
client.c
hello
hello.c
hello.txt
hi.txt
main.c
Makefile
OS
server
shell_commands
utils

terminal> rm hi.txt | rm hello.txt | rmdir OS | rm hello

terminal>

sb7677@DCLAP-V1156-CSD: ~ - x + v
sb7677@DCLAP-V1156-CSD:~/c_shell$ ./client
Socket successfully created..
Connected to the server..

terminal> gcc hello.c -o hello

terminal> ./hello
Hlo World

terminal> cat < hello.c | grep h
#include <stdio.h>

terminal> ls -l |
Invalid input

terminal> ls
client
client.c
hello.c
main.c
Makefile
server
shell_commands
utils

terminal> |
```

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.

```
sb7677@DCLAP-V1156-CSD: ~ X + v
sb7677@DCLAP-V1156-CSD:~/c_shell$ make all
make: Nothing to be done for 'all'.
sb7677@DCLAP-V1156-CSD:~/c_shell$ ./a.out
Socket successfully created..
Socket successfully binded..
Server listening..
server accept the client...

sb7677@DCLAP-V1156-CSD: ~ X + v
sb7677@DCLAP-V1156-CSD:~/c_shell$ ./c.out
Socket successfully created..
connected to the server..
terminal> ls
Sending: ls
a.out
client.c
c.out
main.c
Makefile
shell_commands
utils
terminal> |
```

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.

```
sb7677@DCLAP-V1156-CSD: ~ X + v
sb7677@DCLAP-V1156-CSD:~/c_shell$ ./a.out
Socket successfully created..
Socket successfully binded..
Server listening..
server accept the client...

sb7677@DCLAP-V1156-CSD: ~ X + v
sb7677@DCLAP-V1156-CSD:~/c_shell$ ./c.out
Socket successfully created..
connected to the server..
terminal> echo "Hello World" > hello.txt | mkdir Phase2 | wc < client.c
Sending: echo "Hello World" > hello.txt | mkdir Phase2 | wc < client.c
78 236 1847
terminal> ls -l
Sending: ls -l
total 72
-rwxrwxr-x 1 sb7677 sb7677 22128 Oct 28 20:41 a.out
-rw-rw-r-- 1 sb7677 sb7677 1847 Oct 28 20:40 client.c
-rwxrwxr-x 1 sb7677 sb7677 17296 Oct 28 20:41 c.out
-rw-rw-r-- 1 sb7677 sb7677 14 Oct 28 21:16 hello.txt
-rw-rw-r-- 1 sb7677 sb7677 2655 Oct 28 20:40 main.c
-rw-rw-r-- 1 sb7677 sb7677 165 Oct 28 20:40 Makefile
drwxrwxr-x 2 sb7677 sb7677 4096 Oct 28 21:16 Phase2
drwxrwxr-x 2 sb7677 sb7677 4096 Oct 28 20:40 shell_commands
drwxrwxr-x 2 sb7677 sb7677 4096 Oct 28 20:40 utils
terminal>
```

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.

```
sb7677@DCLAP-V11: X sb7677@DCLAP-V11: X sb7677@DCLAP-V11: X sb7677@DCLAP-V1156-CSD: - X + v
sb7677@DCLAP-V1156-CSD:~/c_shell$ ./a.out
Socket successfully created..
Socket successfully binded..
Server listening..
server accept the client...
|

sb7677@DCLAP-V1156-CSD:~/c_shell$ ./c.out
Socket successfully created..
connected to the server..
terminal> pwd | ./hello
Sending: pwd | ./hello
Hlo World
terminal> |
```

Error Fixation

```
sb7677@DCLAP-V11: X sb7677@DCLAP-V11: X sb7677@DCLAP-V11: X sb7677@DCLAP-V1156-CSD: - X + v
sb7677@DCLAP-V1156-CSD:~/c_shell$ ./a.out
Socket successfully created..
Socket successfully binded..
Server listening..
server accept the client...
|

sb7677@DCLAP-V1156-CSD:~/c_shell$ ./c.out
Socket successfully created..
connected to the server..
terminal> cat client.c | grep a |
Invalid input
terminal> |
```

Stderr transmission

```
sb7677@DCLAP-V11: X sb7677@DCLAP-V11: X sb7677@DCLAP-V11: X sb7677@DCLAP-V1156-CSD: - X + v
sb7677@DCLAP-V1156-CSD:~/c_shell$ ./a.out
Socket successfully created..
Socket successfully binded..
Server listening..
server accept the client...
|

sb7677@DCLAP-V1156-CSD:~/c_shell$ ./c.out
Socket successfully created..
connected to the server..
terminal> ls fsrdsdhjgdsds
Sending: ls fsrdsdhjgdsds
ls: cannot access 'fsrdsdhjgdsds': No such file or directory
terminal> |
```

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.

```
sb7677@DCLAP-V1156-CSD: ~ X + v
sb7677@DCLAP-V1156-CSD:~/c_shell$ make && ./a.out
gcc -Wall -o a.out main.c utils/parser.c shell_commands/commands.c
terminal> ls -l
total 36
-rwxrwxr-x 1 sb7677 sb7677 17776 Oct  9 22:00 a.out
-rw-rw-r-- 1 sb7677 sb7677 1054 Oct  9 22:00 main.c
-rw-rw-r-- 1 sb7677 sb7677 110 Oct  9 22:00 Makefile
drwxrwxr-x 2 sb7677 sb7677 4096 Oct  9 22:00 shell_commands
drwxrwxr-x 2 sb7677 sb7677 4096 Oct  9 22:00 utils
terminal> pwd
/home/sb7677/c_shell
terminal> echo hello
hello
terminal> |
```

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'.

```
sb7677@DCLAP-V1156-CSD: ~ X Windows PowerShell
sb7677@DCLAP-V1156-CSD:~/c_shell$ make && ./a.out
gcc -Wall -o a.out main.c utils/parser.c shell_commands/commands.c
terminal> echo hello > new-file
terminal> cat < new-file
hello
terminal> ls frthtrhrjy
ls: cannot access 'frthtrhrjy': No such file or directory
terminal> ls frthtrhrjy 2> errout-file
terminal> cat errout-file
ls: cannot access 'frthtrhrjy': No such file or directory
terminal> |
```

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.

```
sb7677@DCLAP-V1156-CSD: ~ X + v
sb7677@DCLAP-V1156-CSD:~/c_shell$ make && ./a.out
make: 'a.out' is up to date.
terminal> echo hello there | grep hello | grep there > output.txt
terminal> cat < output.txt
hello there
terminal> cat < output.txt | grep hello
hello there
terminal> pwd | ls -l | wc < output.txt
1 2 12
terminal> pwd | ls | ls -l | ps
      PID TTY          TIME CMD
3175263 pts/19    00:00:00 bash
3175381 pts/19    00:00:00 a.out
3175860 pts/19    00:00:00 ps
terminal> |
```

Exit Command

Input: exit

Expected Result: The shell should exit gracefully.

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

```
sb7677@DCLAP-V1156-CSD: ~ X + v
terminal> pwd
/home/sb7677/c_shell
terminal> ls
a.out main.c Makefile new-file output.txt shell_commands test-file utils
terminal> ls -l
total 48
-rwxrwxr-x 1 sb7677 sb7677 17776 Oct  9 22:00 a.out
-rw-rw-r-- 1 sb7677 sb7677 1054 Oct  9 22:00 main.c
-rw-rw-r-- 1 sb7677 sb7677 110 Oct  9 22:00 Makefile
-rw-rw-r-- 1 sb7677 sb7677 6 Oct  9 22:11 new-file
-rw-rw-r-- 1 sb7677 sb7677 12 Oct  9 22:30 output.txt
drwxrwxr-x 2 sb7677 sb7677 4096 Oct  9 22:00 shell_commands
-rw-rw-r-- 1 sb7677 sb7677 6 Oct  9 22:12 test-file
drwxrwxr-x 2 sb7677 sb7677 4096 Oct  9 22:00 utils
terminal> rm new-file
terminal> ls -l
total 44
-rwxrwxr-x 1 sb7677 sb7677 17776 Oct  9 22:00 a.out
-rw-rw-r-- 1 sb7677 sb7677 1054 Oct  9 22:00 main.c
-rw-rw-r-- 1 sb7677 sb7677 110 Oct  9 22:00 Makefile
-rw-rw-r-- 1 sb7677 sb7677 12 Oct  9 22:30 output.txt
drwxrwxr-x 2 sb7677 sb7677 4096 Oct  9 22:00 shell_commands
-rw-rw-r-- 1 sb7677 sb7677 6 Oct  9 22:12 test-file
drwxrwxr-x 2 sb7677 sb7677 4096 Oct  9 22:00 utils
terminal> exit
sb7677@DCLAP-V1156-CSD:~/c_shell$ |
```

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

