CAPSTONE PROJECT REPORT

ASSIGMENT 2: CUSTOM SHELL IMPLEMENTATION (LINUX OS)

1.Project Title:

Custom shell implementation

2.Objective :

To **develop a simple custom shell in C++** that interacts with the Linux operating system by executing user commands, managing processes, and supporting advanced features such as **input/output redirection, piping, and job control**.

The shell should simulate basic functionalities of the Bash terminal, allowing users to execute commands, run processes in the foreground or background, and handle multiple tasks efficiently through process management and inter-process communication

---

3.Tools and Environment:

Programming Language: C++

Compiler: g++

Operating System: Ubuntu / WSL

Editor Used: Visual Studio Code / Nano

Build System: Makefile

Libraries Used: iostream, vector, string, unistd.h, sys/wait.h, fcntl.h, signal.h

---

4. Project Folder Structure

```
custom_shell_project/
   ├ src/
   ├ main.cpp        → try point of the shell
   ├ shell.cpp       → Handles parsing, execution, redirection, and piping
   ├ shell.h         → Header file for shell functions
   ├ jobs.cpp       → Handles background jobs and job control
   ├ jobs.h         → Header file for job management
   ├ Makefile       → Used to compile the project
   ├ README.md      → Contains project details
   ├ myshell         → Executable file generated after compilation
```

---

5. Project Description

The Custom Shell Project is an attempt to create a basic shell environment using C++.

The shell continuously waits for the user to enter a command. Once entered, it checks whether the command is a built-in one (like cd or exit) or an external program (like ls or cat).

It then executes the command either in the foreground or background, depending on whether the user adds & at the end.

The shell also supports multiple advanced features such as piping (|) and input/output redirection (<, >, and >>).

The program handles errors gracefully and provides a neat, interactive experience similar to the Linux terminal.

---

6. Features Implemented

1. Execution of standard Linux commands (ls, pwd, echo, cat, etc.)
2. Built-in commands like cd, exit, pwd, jobs, fg, bg, and history
3. Background job handling using &
4. Job listing and control with jobs, fg, and bg
5. Input and output redirection (<, >, >>)
6. Command piping using |
7. Signal handling for Ctrl+C and Ctrl+Z
8. Simple, user-friendly interface

7. Source Code

main.cpp

```
#include "shell.h"
int main(int argc, char **argv) {
    // start the shell loop
    shell_loop();
    return 0;
}
```

shell.cpp

```
#include "shell.h"
#include "jobs.h"


#include <iostream>
#include <sstream>
#include <unistd.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <signal.h>
#include <cstring>
#include <errno.h>
#include <vector>
#include <algorithm>


static pid_t shell_pgid;
static int shell_terminal;
static bool shell_interactive = false;
static std::vector<std::string> history_lines;

// Signal handlers
void sigint_handler(int signo) {
    // Send SIGINT to foreground process group
    // do nothing here; shell will forward signals to foreground processes
    std::cout << "\n";
}

void sigtstp_handler(int signo) {
    std::cout << "\n";
}

std::vector<std::string> tokenize(const std::string &line) {
    std::istringstream iss(line);
    std::vector<std::string> tokens;
    std::string tok;
    while (iss >> tok) tokens.push_back(tok);
    return tokens;
}

// split tokens by '|' pipeline into list of commands (each is vector<string>)
static std::vector<std::vector<std::string>> split_pipeline(const std::vector<std::string>&
tokens) {
    std::vector<std::vector<std::string>> cmds;
    std::vector<std::string> cur;
    for (auto &t : tokens) {
        if (t == "|") {
            cmds.push_back(cur);
            cur.clear();
        } else {
            cur.push_back(t);
        }
    }
    if (!cur.empty()) cmds.push_back(cur);
    return cmds;
}

bool is_builtin(const std::string &cmd) {
    static const std::vector<std::string> builtins = {"cd","exit","pwd","jobs","history","fg","bg"};
    return std::find(builtins.begin(), builtins.end(), cmd) != builtins.end();
}

// Run built-in commands
void run_builtin(const std::vector<std::string> &args) {
    if (args.empty()) return;
    const std::string &cmd = args[0];
    if (cmd == "cd") {
        const char* path = (args.size() > 1) ? args[1].c_str() : getenv("HOME");
        if (chdir(path) != 0) {
            perror("cd");
        }
    } else if (cmd == "exit") {
        exit(0);
    } else if (cmd == "pwd") {
        char buf[4096];
        if (getcwd(buf, sizeof(buf))) {
            std::cout << buf << "\n";
        } else {
            perror("pwd");
        }
    } else if (cmd == "jobs") {
        list_jobs();
    } else if (cmd == "history") {
        int num = 1;
        for (auto &l : history_lines) {
            std::cout << num++ << " " << l << "\n";
        }
    } else if (cmd == "fg") {
        if (args.size() < 2) {
            std::cerr << "fg: job id required\n";
            return;
        }
        int id = std::stoi(args[1]);
        Job* j = find_job_by_id(id);
        if (!j) { std::cerr << "fg: no such job\n"; return; }
        // bring to foreground
        pid_t pgid = j->pgid;
        tcsetpgrp(STDIN_FILENO, pgid);
        kill(-pgid, SIGCONT);
        mark_job_as_running(pgid);
        int status;
        waitpid(-pgid, &status, WUNTRACED);
        tcsetpgrp(STDIN_FILENO, getpid());
        if (WIFSTOPPED(status)) {
            mark_job_as_stopped(pgid);
        } else {
            remove_job_by_pgid(pgid);
        }
    } else if (cmd == "bg") {
        if (args.size() < 2) {
            std::cerr << "bg: job id required\n";
            return;
        }
        int id = std::stoi(args[1]);
        Job* j = find_job_by_id(id);
        if (!j) { std::cerr << "bg: no such job\n"; return; }
        pid_t pgid = j->pgid;
        kill(-pgid, SIGCONT);
        mark_job_as_running(pgid);
    }
}

void handle_redirection_and_exec(std::vector<std::string> args, bool background) {
    // Check for redirection tokens: > >> <
    int in_fd = -1, out_fd = -1;
    std::vector<char*> argv;
    for (size_t i=0;i<args.size();++i) {
        if (args[i] == "<" && i+1 < args.size()) {
            in_fd = open(args[i+1].c_str(), O_RDONLY);
            if (in_fd < 0) { perror("open"); return; }
            i++; // skip filename
        } else if (args[i] == ">" && i+1 < args.size()) {
            out_fd = open(args[i+1].c_str(), O_WRONLY | O_CREAT | O_TRUNC, 0644);
            if (out_fd < 0) { perror("open"); return; }
            i++;
        } else if (args[i] == ">>" && i+1 < args.size()) {
            out_fd = open(args[i+1].c_str(), O_WRONLY | O_CREAT | O_APPEND, 0644);
            if (out_fd < 0) { perror("open"); return; }
            i++;
        } else {
            argv.push_back(const_cast<char*>(args[i].c_str()));
        }
    }
```

```
argv.push_back(nullptr);


if (argv.size() <= 1) return;


pid_t pid = fork();
if (pid < 0) {
    perror("fork");
    return;
} else if (pid == 0) {
    // Child
    setpgid(0,0); // new process group
    if (!background) tcsetpgrp(STDIN_FILENO, getpid());

    if (in_fd != -1) {
        dup2(in_fd, STDIN_FILENO);
        close(in_fd);
    }
    if (out_fd != -1) {
        dup2(out_fd, STDOUT_FILENO);
        close(out_fd);
    }

    // Restore default signals in child
    signal(SIGINT, SIG_DFL);
    signal(SIGTSTP, SIG_DFL);
```

```
        execvp(argv[0], argv.data());
        // if exec fails:
        std::cerr << argv[0] << ": command not found\n";
        _exit(127);
    } else {
        // Parent
        setpgid(pid, pid);
        if (background) {
            int jobid = add_job(pid, args[0], true);
            std::cout << "[" << jobid << "] " << pid << "\n";
        } else {
            tcsetpgrp(STDIN_FILENO, pid);
            int status;
            waitpid(pid, &status, WUNTRACED);
            tcsetpgrp(STDIN_FILENO, getpid());
            if (WIFSTOPPED(status)) {
                mark_job_as_stopped(pid);
            } else {
                remove_job_by_pgid(pid);
            }
        }
    }
}
// For a pipeline of multiple commands: create pipes and fork accordingly.
```

```
void run_pipeline(const std::vector<std::vector<std::string>> &commands, bool
background) {
    int n = commands.size();
    std::vector<int> pfd(2*(n-1));
    for (int i=0;i<n-1;++i) {
        if (pipe(&pfd[2*i]) == -1) { perror("pipe"); return; }
    }
    std::vector<pid_t> pids;
    for (int i=0;i<n;++i) {
        pid_t pid = fork();
        if (pid < 0) { perror("fork"); return; }
        else if (pid == 0) {
            // Child: set up stdin/stdout if needed
            if (i > 0) {
                dup2(pfd[2*(i-1)], STDIN_FILENO);
            }
            if (i < n-1) {
                dup2(pfd[2*i + 1], STDOUT_FILENO);
            }
            // close all pipe fds
            for (int j=0;j<2*(n-1);++j) close(pfd[j]);
            setpgid(0,0);
            if (!background) tcsetpgrp(STDIN_FILENO, getpid());
            signal(SIGINT, SIG_DFL);
            signal(SIGTSTP, SIG_DFL);
            // prepare argv
            std::vector<char*> argv;
            for (auto &s : commands[i]) argv.push_back(const_cast<char*>(s.c_str()));
            argv.push_back(nullptr);
            execvp(argv[0], argv.data());
            std::cerr << commands[i][0] << ": command not found\n";
            _exit(127);
        } else {
            // Parent
            setpgid(pid, pid); // each child in its own pgid; for better control we could use first
child's pid
            pids.push_back(pid);
        }
    }

    // parent closes pipes
    for (int j=0;j<2*(n-1);++j) close(pfd[j]);

    // put first child's pgid in jobs if background
    pid_t pgid = pids.empty() ? -1 : pids[0];
    if (background && pgid > 0) {
        int jobid = add_job(pgid, commands[0][0], true);
        std::cout << "[" << jobid << "] " << pgid << "\n";
    } else {
        // wait for all children
        for (pid_t pid : pids) {
            int status;
            tcsetpgrp(STDIN_FILENO, pgid);
            waitpid(pid, &status, WUNTRACED);
            if (WIFSTOPPED(status)) mark_job_as_stopped(pgid);
        }
        tcsetpgrp(STDIN_FILENO, getpid());
        remove_job_by_pgid(pgid);
    }
}
void execute_line(const std::string &line) {
    if (line.empty()) return;
    history_lines.push_back(line);
    // rough parsing
    auto tokens = tokenize(line);
    if (tokens.empty()) return;

    bool background = false;
    if (tokens.back() == "&") {
        background = true;
        tokens.pop_back();
    }

    // pipeline?
    auto piped = split_pipeline(tokens);
    if (piped.size() > 1) {
        // if single command in pipeline is builtin -> not supported for pipeline here
        run_pipeline(piped, background);
        return;
    }

    // no pipeline
    if (is_builtin(tokens[0])) {
        run_builtin(tokens);
        return;
    }

    handle_redirection_and_exec(tokens, background);
}

void shell_loop() {
    // initialize shell
    shell_terminal = STDIN_FILENO;
    shell_interactive = isatty(shell_terminal);
    if (shell_interactive) {
        // Put shell in its own process group
        while (tcgetpgrp(shell_terminal) != (shell_pgid = getpgrp()))
            kill(-shell_pgid, SIGTTIN);
        shell_pgid = getpid();
        setpgid(shell_pgid, shell_pgid);
        tcsetpgrp(shell_terminal, shell_pgid);


        signal(SIGINT, sigint_handler);
        signal(SIGTSTP, sigtstp_handler);
    }


    init_jobs();


    std::string line;
    while (true) {
        char cwd[4096];
        if (getcwd(cwd, sizeof(cwd)) != nullptr) {
            std::cout << cwd << " $ ";
        } else {
            std::cout << "shell $ ";
        }
        std::getline(std::cin, line);
        if (!std::cin) break;
        execute_line(line);
    }
}
```

---

shell.h

```
#ifndef SHELL_H
#define SHELL_H
```

```
#include <string>
#include <vector>

void shell_loop();
std::vector<std::string> tokenize(const std::string &line);
void execute_line(const std::string &line);
bool is_builtin(const std::string &cmd);
void run_builtin(const std::vector<std::string> &args);
void handle_redirection_and_exec(std::vector<std::string> args, bool background);
void run_pipeline(const std::vector<std::vector<std::string>> &commands, bool
background);

#endif // SHELL_H
```

---

jobs.cpp

```
#include "jobs.h"
#include <vector>
#include <iostream>
#include <algorithm>

static std::vector<Job> jobs;
static int next_job_id = 1;

void init_jobs() {
    jobs.clear();
    next_job_id = 1;
```

```
}

int add_job(pid_t pgid, const std::string &cmdline, bool running) {
    Job j;
    j.id = next_job_id++;
    j.pgid = pgid;
    j.cmdline = cmdline;
    j.running = running;
    jobs.push_back(j);
    return j.id;
}

void remove_job_by_pgid(pid_t pgid) {
    jobs.erase(std::remove_if(jobs.begin(), jobs.end(),
        [pgid](const Job &j){ return j.pgid == pgid; }), jobs.end());
}

Job* find_job_by_id(int id) {
    for (auto &j : jobs) {
        if (j.id == id) return &j;
    }
    return nullptr;
}

Job* find_job_by_pgid(pid_t pgid) {
    for (auto &j : jobs) {
        if (j.pgid == pgid) return &j;
    }
    return nullptr;
}

void list_jobs() {
    for (const auto &j : jobs) {
        std::cout << '[' << j.id << "] "
                  << (j.running ? "Running " : "Stopped ")
                  << j.cmdline << " (pgid " << j.pgid << ")\n";
    }
}

void mark_job_as_stopped(pid_t pgid) {
    Job* j = find_job_by_pgid(pgid);
    if (j) j->running = false;
}

void mark_job_as_running(pid_t pgid) {
    Job* j = find_job_by_pgid(pgid);
    if (j) j->running = true;
}
```

---

jobs.h

jobs.h:

```
#ifndef JOBS_H
#define JOBS_H
#include <string>
#include <vector>
#include <sys/types.h>

struct Job {
    int id;
    pid_t pgid;
    std::string cmdline;
    bool running;
};

void init_jobs();
int add_job(pid_t pgid, const std::string &cmdline, bool running=true);
void remove_job_by_pgid(pid_t pgid);
Job* find_job_by_id(int id);
Job* find_job_by_pgid(pid_t pgid);
void list_jobs();
void mark_job_as_stopped(pid_t pgid);
void mark_job_as_running(pid_t pgid);
#endif // JOBS_H
```

Makefile:

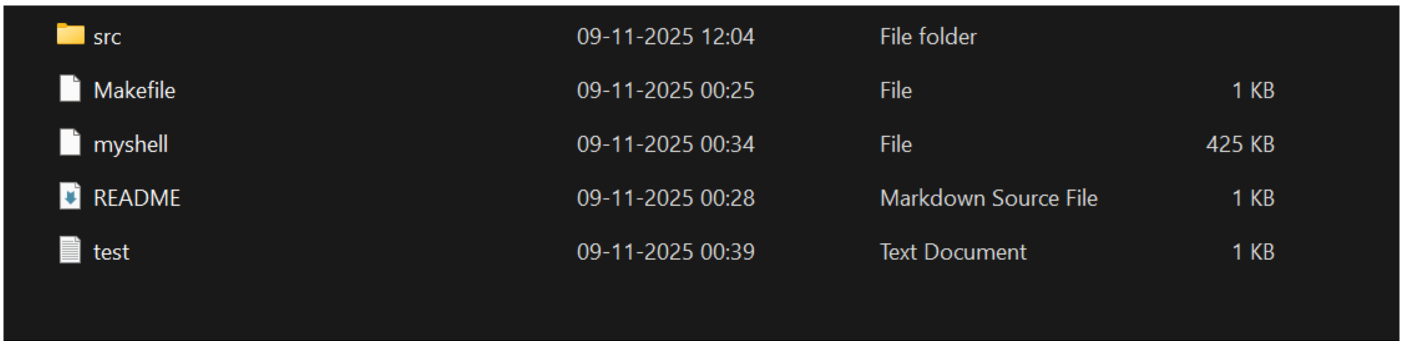
```
CXX = g++
CXXFLAGS = -std=c++17 -Wall -Wextra -g
```

```
SRC = src/main.cpp src/shell.cpp src/jobs.cpp

OBJ = $(SRC:.cpp=.o)

TARGET = myshell


all: $(TARGET)


$(TARGET): $(OBJ)

        $(CXX) $(CXXFLAGS) -o $(TARGET) $(OBJ)


%.o: %.cpp

        $(CXX) $(CXXFLAGS) -c $< -o $@


clean:

        rm -f src/*.o $(TARGET)
```

---

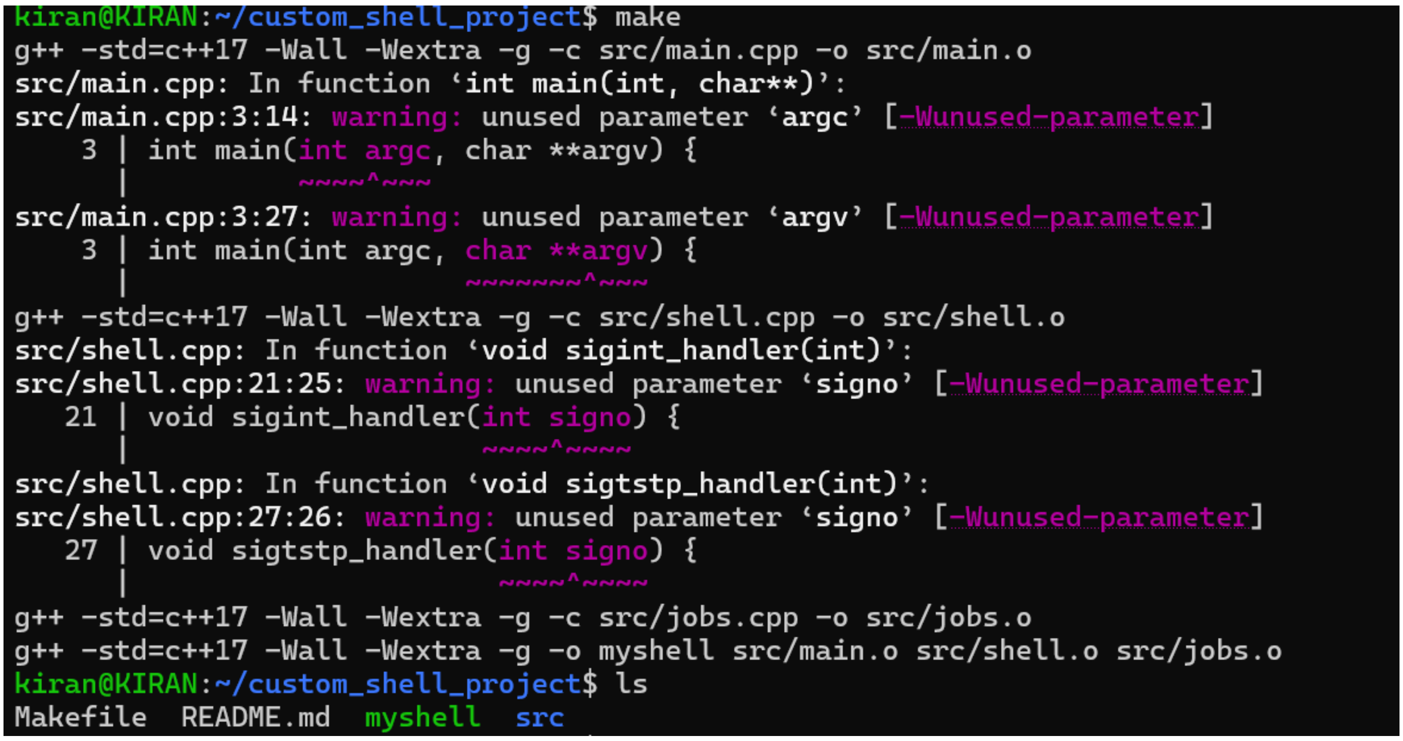
8. Output Screenshots

Screenshot 1: Folder Structure



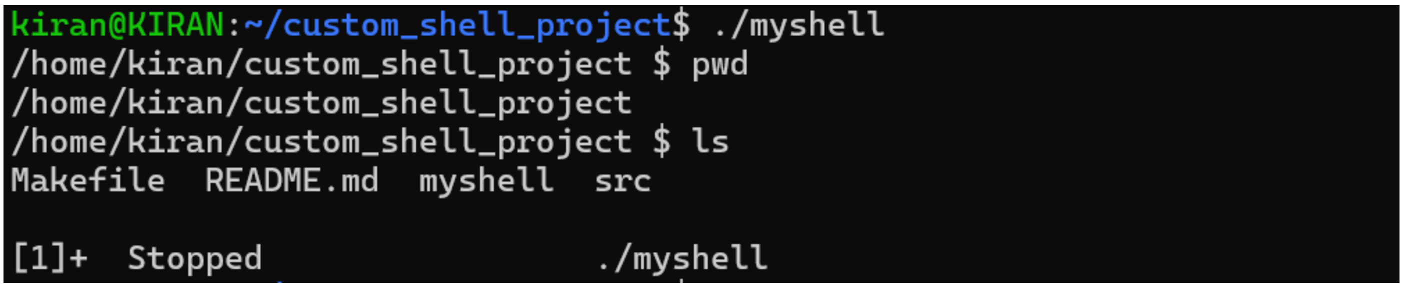
src	09-11-2025 12:04	File folder	
Makefile	09-11-2025 00:25	File	1 KB
myshell	09-11-2025 00:34	File	425 KB
README	09-11-2025 00:28	Markdown Source File	1 KB
test	09-11-2025 00:39	Text Document	1 KB

Screenshot 2: Compilation



```
kiran@KIRAN:~/custom_shell_project$ make
g++ -std=c++17 -Wall -Wextra -g -c src/main.cpp -o src/main.o
src/main.cpp: In function ‘int main(int, char**)’:
src/main.cpp:3:14: warning: unused parameter ‘argc’ [-Wunused-parameter]
    3 | int main(int argc, char **argv) {
      |          ~~~~^~~~
src/main.cpp:3:27: warning: unused parameter ‘argv’ [-Wunused-parameter]
    3 | int main(int argc, char **argv) {
      |                    ~~~~~~~^~~~
g++ -std=c++17 -Wall -Wextra -g -c src/shell.cpp -o src/shell.o
src/shell.cpp: In function ‘void sigint_handler(int)’:
src/shell.cpp:21:25: warning: unused parameter ‘signo’ [-Wunused-parameter]
   21 | void sigint_handler(int signo) {
      |                     ~~~~^~~~~
src/shell.cpp: In function ‘void sigtstp_handler(int)’:
src/shell.cpp:27:26: warning: unused parameter ‘signo’ [-Wunused-parameter]
   27 | void sigtstp_handler(int signo) {
      |                      ~~~~^~~~~
g++ -std=c++17 -Wall -Wextra -g -c src/jobs.cpp -o src/jobs.o
g++ -std=c++17 -Wall -Wextra -g -o myshell src/main.o src/shell.o src/jobs.o
kiran@KIRAN:~/custom_shell_project$ ls
Makefile  README.md  myshell  src
```

Screenshot 3: Running the Shell



```
kiran@KIRAN:~/custom_shell_project$ ./myshell
/home/kiran/custom_shell_project $ pwd
/home/kiran/custom_shell_project
/home/kiran/custom_shell_project $ ls
Makefile  README.md  myshell  src

[1]+  Stopped                 ./myshell
```

Screenshot 4: Execution of Commands, Redirection, Piping, and Job Control:

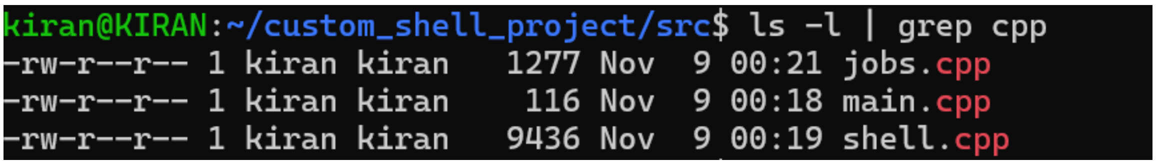
Commands Demonstrated:

1. Basic Command: ls – Lists all files and folders in the current directory.
2. Navigation Command: cd src – Moves into the src directory.
3. Print Working Directory: pwd – Displays the current directory path.
4. Redirection (Output): echo "Hello" > test.txt – Creates a file named test.txt and writes “Hello” into it.
5. Redirection (Input): cat < test.txt – Reads and displays the contents of test.txt.
6. Piping: ls -l | grep cpp – Sends the output of ls -l to grep cpp, showing only .cpp files.

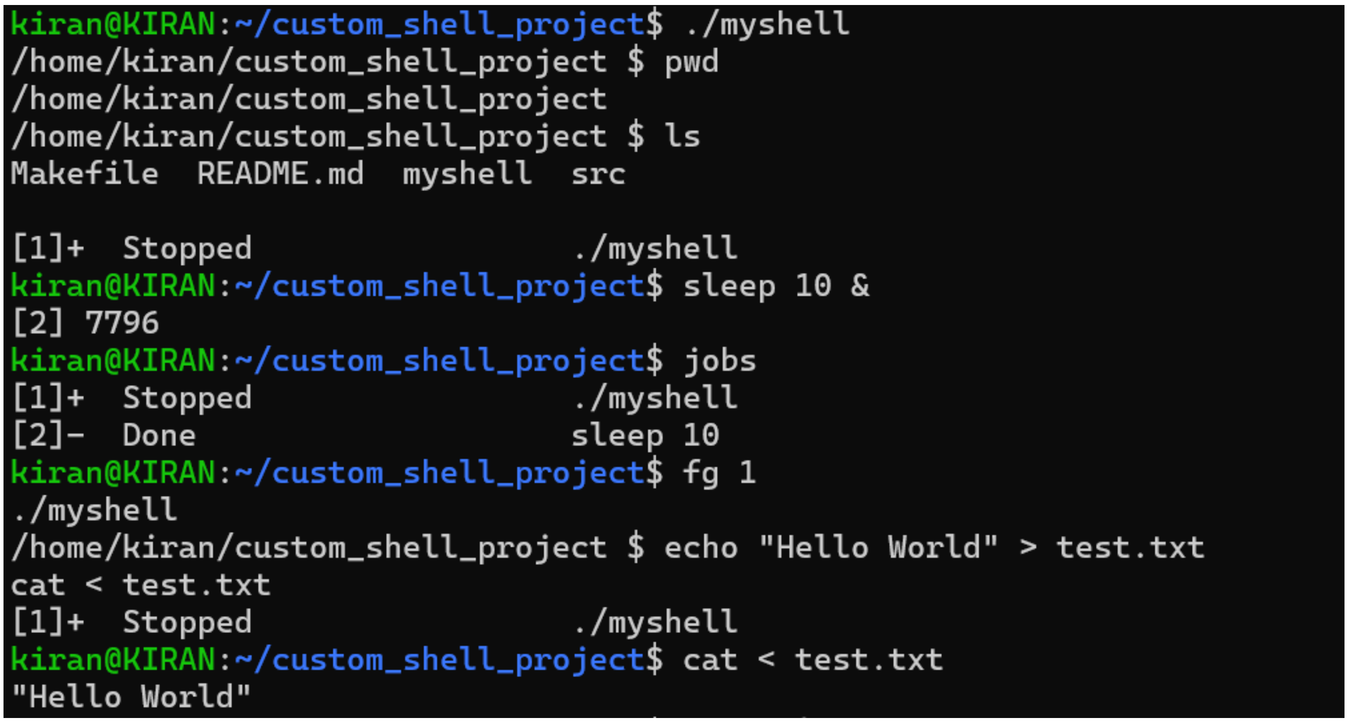
7. Background Process: sleep 10 & – Runs the sleep command in the background.

8. Job Listing: jobs – Displays the currently running background processes.

9. Foreground Process: fg 1 – Brings job number 1 to the foreground for execution.



```
kiran@KIRAN:~/custom_shell_project/src$ ls -l | grep cpp
-rw-r--r-- 1 kiran kiran   1277 Nov  9 00:21 jobs.cpp
-rw-r--r-- 1 kiran kiran    116 Nov  9 00:18 main.cpp
-rw-r--r-- 1 kiran kiran   9436 Nov  9 00:19 shell.cpp
```



```
kiran@KIRAN:~/custom_shell_project$ ./myshell
/home/kiran/custom_shell_project $ pwd
/home/kiran/custom_shell_project
/home/kiran/custom_shell_project $ ls
Makefile  README.md  myshell  src

[1]+  Stopped                 ./myshell
kiran@KIRAN:~/custom_shell_project$ sleep 10 &
[2] 7796
kiran@KIRAN:~/custom_shell_project$ jobs
[1]+  Stopped                 ./myshell
[2]-  Done                    sleep 10
kiran@KIRAN:~/custom_shell_project$ fg 1
./myshell
/home/kiran/custom_shell_project $ echo "Hello World" > test.txt
cat < test.txt
[1]+  Stopped                 ./myshell
kiran@KIRAN:~/custom_shell_project$ cat < test.txt
"Hello World"
```

9.**Learning Outcomes:**

Through this project, I learned how a shell communicates with the operating system and how system calls like fork(), execvp(), and waitpid() are used for process management.

I also understood how input/output redirection, pipes, and background job control are implemented in Linux.

This project gave me hands-on experience with low-level system programming and helped me understand how real shells like Bash work internally

10. Conclusion

The *Custom Shell Project* successfully demonstrates command execution, redirection, piping, and background job management using C++.

It fulfills all the requirements of the Capstone Assignment 2 and provided valuable learning on how terminal commands are processed at the OS level.

11.Github Respository Link:

https://github.com/nayakkirankrishna/Custom-Shell-Implementation.git

---

12.Author Information:

Name : Kiran Krishna Nayak

Regd No : 2241014087

Batch : 7

Email id :nayakkirankrishna@gmail.com

University : Siksha ‘O’ Anusandhan (SOA) University, Bhubaneswar

Year:4th 2026