ASSIGNMENT_02(LINUXOS)

CustomShell — A Unix-style Shell Implementation in C++

Abstract

This project presents the design and implementation of CustomShell, a minimal Unix-style command-line shell written in C++. The shell supports command execution, background processing, job control, single-stage piping, and input/output redirection. It is built using essential Linux system calls such as fork, execvp, pipe, dup2, and waitpid.

Introduction

CustomShell is a simplified Linux shell created as part of a capstone project. Its purpose is to demonstrate understanding of process management, inter-process communication, and system-level programming.

Objectives

- Build a simple shell in C++ that can execute commands, manage processes and handle redirection and piping

System Architecture

The shell uses a modular C++ design:
- Parser: parses input into a Command structure
- Executor: executes commands, manages redirection and jobs
- Pipeline: handles A | B execution
- JobManager: tracks background processes

Implementation Overview

The shell reads user input using getline(), parses it to detect pipes and redirection, and executes commands via fork() + execvp(). Background jobs are stored in a job table.

Features

- Command execution
- Argument support
- Background execution (&)
- Job control (jobs, fg %n, bg %n)
- Single pipe (ls | grep .cpp)
- Input redirection (<)
- Output redirection (>)

Testing & Results

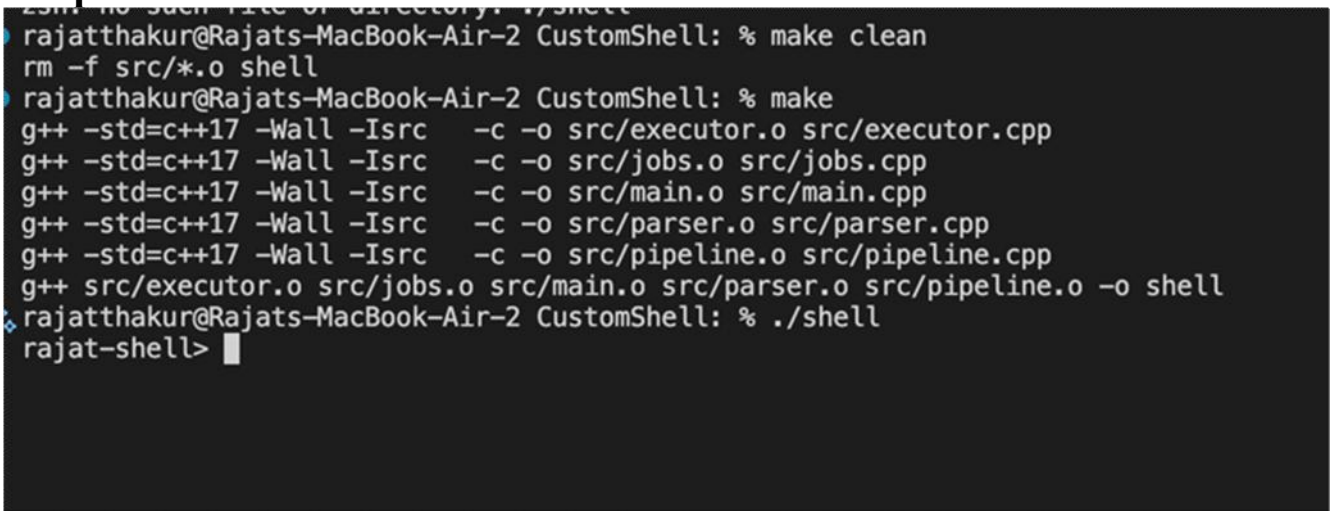
DAY 1:Create Basic Shell REPL

file name - main.cpp
Create a simple interactive shell (`rajat-shell>`) that reads input using `getline()` and executes commands continuously using a REPL loop.

Code:

```cpp
#include <iostream>
#include <string>
#include "parser.h"
#include "executor.h"

int main() {
    std::string input;

    while (true) {
        std::cout << "rajat-shell> ";
        std::getline(std::cin, input);  // IMPORTANT: read the whole line

        if (input == "exit") break;
        if (input.empty()) continue;

        Command cmd = Parser::parseInput(input);  //  send full line to parser
        Executor::executeCommand(cmd);            // pipe/redirect logic now works
    }

    return 0;
}
```

Output:



```
rajatthakur@Rajats-MacBook-Air-2 CustomShell: % make clean
rm -f src/*.o shell
rajatthakur@Rajats-MacBook-Air-2 CustomShell: % make
g++ -std=c++17 -Wall -Isrc   -c -o src/executor.o src/executor.cpp
g++ -std=c++17 -Wall -Isrc   -c -o src/jobs.o src/jobs.cpp
g++ -std=c++17 -Wall -Isrc   -c -o src/main.o src/main.cpp
g++ -std=c++17 -Wall -Isrc   -c -o src/parser.o src/parser.cpp
g++ -std=c++17 -Wall -Isrc   -c -o src/pipeline.o src/pipeline.cpp
g++ src/executor.o src/jobs.o src/main.o src/parser.o src/pipeline.o -o shell
rajatthakur@Rajats-MacBook-Air-2 CustomShell: % ./shell
rajat-shell>
```

DAY 2 : Implement execution of basic command using shell

file name – parser.h

Implement a parser that identifies:

- command arguments
- background symbol &
- pipe |
- input redirection <
- output redirection >

Code:

```cpp
#include "parser.h"
#include <sstream>

Command Parser::parseInput(const std::string &input) {
    Command cmd;
    std::stringstream ss(input);
    std::string token;

    bool rightSide = false;  // used for pipe

    while (ss >> token) {

        // 1. Handle PIPE |
        if (token == "|") {
            cmd.isPipe = true;
            rightSide = true;
        }

        // 2. Handle OUTPUT REDIRECTION >
        else if (token == ">") {
            ss >> cmd.outFile;          // get filename
            cmd.outputRedirect = true;
        }

        // 3. Handle INPUT REDIRECTION <
        else if (token == "<") {
            ss >> cmd.inFile;
            cmd.inputRedirect = true;
        }

        // 4. Handle BACKGROUND EXECUTION &
        else if (token == "&") {
            cmd.isBackground = true;
        }

        // 5. Normal command tokens
        else {
            if (!rightSide)
                cmd.args.push_back(token);        // left side
            else
                cmd.pipeRightArgs.push_back(token); // right side
        }
    }

    return cmd;
}
```

Output:

```
rajatthakur@Rajats-MacBook-Air-2 CustomShell: % ./shell
rajat-shell> ls
a.txt           build           Makefile        shell
b.txt           data            README.md       src
rajat-shell> echo hello world
hello world
rajat-shell> sleep 5 &
[1] 39454 started in background
rajat-shell>
```

DAY 3 : Add support for process management (foreground,background,processes).

File name –jobs.cpp

Add background job tracking and control using:

`jobs`

`fg %n`

`bg %n`

job completion detection

Code:

```cpp
#include "jobs.h"
#include <iostream>
#include <sys/wait.h>
#include <signal.h>
#include <unistd.h>

std::vector<Job> JobManager::jobs;
int JobManager::nextJobId = 1;

void JobManager::addJob(pid_t pid, const std::string &cmd) {
    Job job;
    job.jobId = nextJobId++;
    job.pid = pid;
    job.command = cmd;
    job.running = true;

    jobs.push_back(job);

    std::cout << "[" << job.jobId << "] " << pid << " started in background\n";
}

void JobManager::listJobs() {
    for (auto &job : jobs) {
        std::cout << "[" << job.jobId << "] "
                  << (job.running ? "Running " : "Stopped ")
                  << "PID: " << job.pid
                  << "  Command: " << job.command << "\n";
    }
}

void JobManager::bringToForeground(int jobId) {
    for (auto &job : jobs) {
        if (job.jobId == jobId) {
            std::cout << "Bringing job to foreground: " << job.command << "\n";
            kill(job.pid, SIGCONT);
            waitpid(job.pid, nullptr, 0);
            job.running = false;
            return;
        }
    }
    std::cout << "fg: job not found\n";
}

void JobManager::moveToBackground(int jobId) {
    for (auto &job : jobs) {
        if (job.jobId == jobId) {
            std::cout << "Resuming job in background: " << job.command << "\n";
            kill(job.pid, SIGCONT);
            job.running = true;
            return;
        }
    }
    std::cout << "bg: job not found\n";
}

void JobManager::checkCompletedJobs() {
    int status;
    pid_t pid;

    while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
        for (auto &job : jobs) {
            if (job.pid == pid) {
                std::cout << "\n[Job Completed] PID " << pid
                          << " (" << job.command << ")\n";
                job.running = false;
            }
        }
    }
}
```

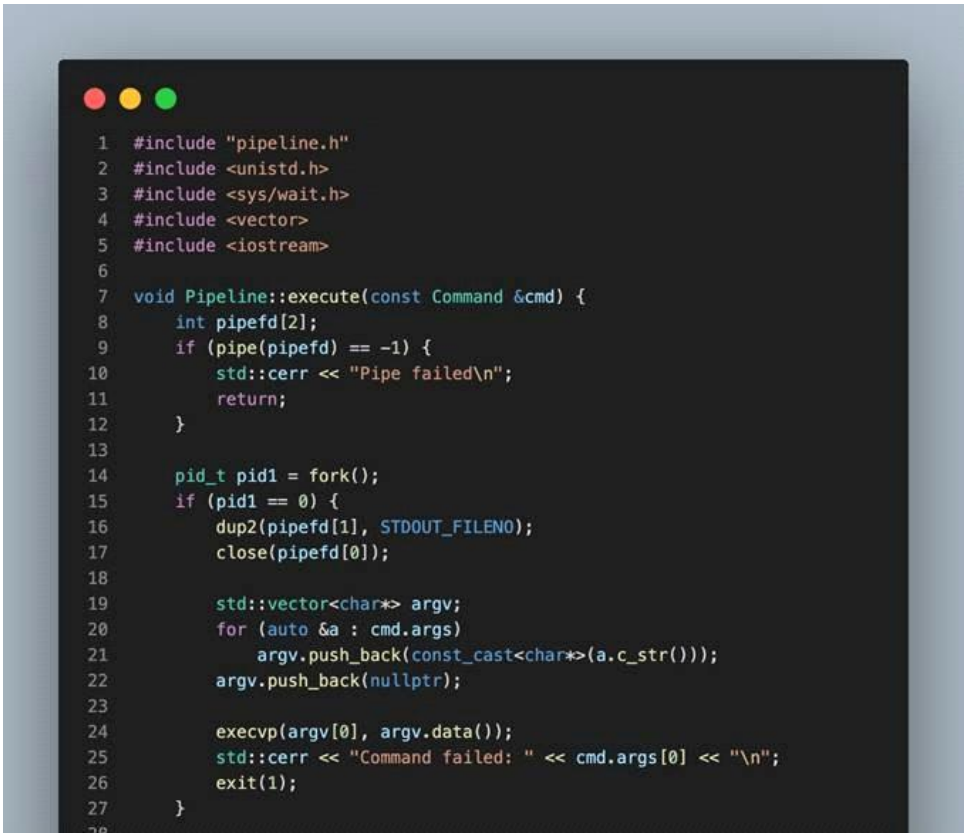
Output:

```
rajat-shell> sleep 5 &
[1] 39454 started in background
rajat-shell> sleep 5 &
sleep 3 &
jobs
fg %1
bg %2
[2] 39909 started in background

[Job Completed] PID 39454 (sleep)
rajat-shell> [3] 39910 started in background
rajat-shell> [1] Stopped PID: 39454  Command: sleep
[2] Running PID: 39909  Command: sleep
[3] Running PID: 39910  Command: sleep
rajat-shell> Bringing job to foreground: sleep
rajat-shell> Resuming job in background: sleep
rajat-shell>
```

DAY 4 : Implement piping and redirection features.

File name -pipleline.cpp
Integrate:
`A | B` (single pipe)
Output redirection: >
Input redirection: <
Combined `< file > file2`

Code:



```cpp
#include "pipeline.h"
#include <unistd.h>
#include <sys/wait.h>
#include <vector>
#include <iostream>

void Pipeline::execute(const Command &cmd) {
    int pipefd[2];
    if (pipe(pipefd) == -1) {
        std::cerr << "Pipe failed\n";
        return;
    }

    pid_t pid1 = fork();
    if (pid1 == 0) {
        dup2(pipefd[1], STDOUT_FILENO);
        close(pipefd[0]);

        std::vector<char*> argv;
        for (auto &a : cmd.args)
            argv.push_back(const_cast<char*>(a.c_str()));
        argv.push_back(nullptr);

        execvp(argv[0], argv.data());
        std::cerr << "Command failed: " << cmd.args[0] << "\n";
        exit(1);
    }
```

```
    pid_t pid2 = fork();
    if (pid2 == 0) {
        dup2(pipefd[0], STDIN_FILENO);
        close(pipefd[1]);

        std::vector<char*> argv;
        for (auto &a : cmd.pipeRightArgs)
            argv.push_back(const_cast<char*>(a.c_str()));
        argv.push_back(nullptr);

        execvp(argv[0], argv.data());
        std::cerr << "Command failed: " << cmd.pipeRightArgs[0] << "\n";
        exit(1);
    }

    close(pipefd[0]);
    close(pipefd[1]);
    waitpid(pid1, nullptr, 0);
    waitpid(pid2, nullptr, 0);
}
```

Output:

```
rajat-shell> ls | grep cpp
echo hello > a.txt
cat a.txt
cat < a.txt
grep h < a.txt > b.txt
cat b.txt
rajat-shell>
[Job Completed] PID 39909 (sleep)

[Job Completed] PID 39910 (sleep)
rajat-shell> hello
rajat-shell> hello
rajat-shell> rajat-shell> hello
rajat-shell>
```

DAY 5 : Incorporate job control (listing jobs, bringing jobs to foreground/background).

File name - `main.cpp`, `executor.cpp`, `parser.cpp`, `jobs.cpp`, `pipeline.cpp`

Combine all features:

- REPL
- parser
- job control
- pipeline
- redirection
- background execution
- final testing

Code: main.cpp

```cpp
#include <iostream>
#include <string>
#include "parser.h"
#include "executor.h"

int main() {
    std::string input;

    while (true) {
        std::cout << "rajat-shell> ";
        std::getline(std::cin, input);  // IMPORTANT: read the whole line

        if (input == "exit") break;
        if (input.empty()) continue;

        Command cmd = Parser::parseInput(input);  //  send full line to parser
        Executor::executeCommand(cmd);            // pipe/redirect logic now works
    }

    return 0;
}

```

Code:executor.cpp

```cpp
#include "executor.h"
#include <unistd.h>
#include <sys/wait.h>
#include <iostream>
#include <fcntl.h>          // for redirection
#include "jobs.h"           // Job Manager
#include "pipeline.h"       // Pipe handling

void Executor::executeCommand(const Command &cmd) {

    if (cmd.args.empty()) return;

    // ✅ Built-in job commands
    if (cmd.args[0] == "jobs") {
        JobManager::listJobs();
        return;
    }
    if (cmd.args[0] == "fg" && cmd.args.size() == 2) {
        int jobId = std::stoi(cmd.args[1].substr(1)); // remove %
        JobManager::bringToForeground(jobId);
        return;
    }
    if (cmd.args[0] == "bg" && cmd.args.size() == 2) {
        int jobId = std::stoi(cmd.args[1].substr(1)); // remove %
        JobManager::moveToBackground(jobId);
        return;
    }

    // ✅ PIPE handling (Day 4)
    if (cmd.isPipe) {
        Pipeline::execute(cmd);
        return;
    }

    // ✅ Convert args to execvp format
    std::vector<char*> argv;
    for (auto &arg : cmd.args)
        argv.push_back(const_cast<char*>(arg.c_str()));
    argv.push_back(nullptr);

    pid_t pid = fork();
```

```
    if (pid == 0) {
        // ✅ Output redirection >
        if (cmd.outputRedirect) {
            int fd = open(cmd.outFile.c_str(), O_WRONLY | O_CREAT | O_TRUNC, 0644);
            if (fd < 0) {
                std::cerr << "Failed to open file for output\n";
                exit(1);
            }
            dup2(fd, STDOUT_FILENO);
            close(fd);
        }

        // ✅ Input redirection <
        if (cmd.inputRedirect) {
            int fd = open(cmd.inFile.c_str(), O_RDONLY);
            if (fd < 0) {
                std::cerr << "Failed to open file for input\n";
                exit(1);
            }
            dup2(fd, STDIN_FILENO);
            close(fd);
        }

        // ✅ Execute normal command
        execvp(argv[0], argv.data());
        std::cerr << "Command not found: " << cmd.args[0] << "\n";
        exit(1);
    }

    // ✅ Foreground or Background execution
    if (!cmd.isBackground) {
        waitpid(pid, nullptr, 0);
    } else {
        JobManager::addJob(pid, cmd.args[0]);
    }

    // ✅ Check completed background jobs
    JobManager::checkCompletedJobs();
}
```

Code: parser.cpp

```
#include "parser.h"
#include <sstream>

Command Parser::parseInput(const std::string &input) {
    Command cmd;
    std::stringstream ss(input);
    std::string token;

    bool rightSide = false;  // used for pipe

    while (ss >> token) {

        // 1. Handle PIPE |
        if (token == "|") {
            cmd.isPipe = true;
            rightSide = true;
        }

        // 2. Handle OUTPUT REDIRECTION >
        else if (token == ">") {
            ss >> cmd.outFile;          // get filename
            cmd.outputRedirect = true;
        }

        // 3. Handle INPUT REDIRECTION <
        else if (token == "<") {
            ss >> cmd.inFile;
            cmd.inputRedirect = true;
        }

        // 4. Handle BACKGROUND EXECUTION &
        else if (token == "&") {
            cmd.isBackground = true;
        }

        // 5. Normal command tokens
        else {
            if (!rightSide)
                cmd.args.push_back(token);        // left side
            else
                cmd.pipeRightArgs.push_back(token); // right side
        }
    }

    return cmd;
}
```

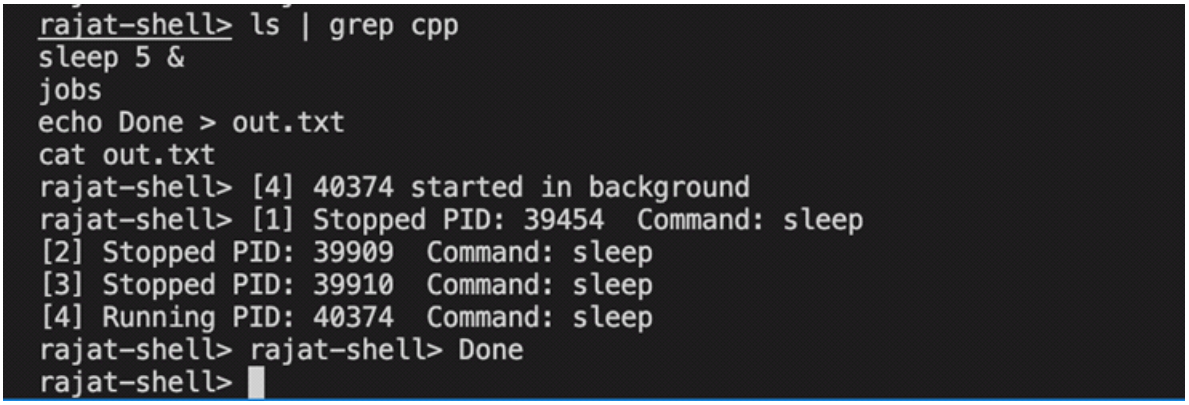
Code:jobs.cpp

```cpp
#include "jobs.h"
#include <iostream>
#include <sys/wait.h>
#include <signal.h>
#include <unistd.h>

std::vector<Job> JobManager::jobs;
int JobManager::nextJobId = 1;

void JobManager::addJob(pid_t pid, const std::string &cmd) {
    Job job;
    job.jobId = nextJobId++;
    job.pid = pid;
    job.command = cmd;
    job.running = true;

    jobs.push_back(job);

    std::cout << "[" << job.jobId << "] " << pid << " started in background\n";
}

void JobManager::listJobs() {
    for (auto &job : jobs) {
        std::cout << "[" << job.jobId << "] "
                  << (job.running ? "Running " : "Stopped ")
                  << "PID: " << job.pid
                  << "  Command: " << job.command << "\n";
    }
}

void JobManager::bringToForeground(int jobId) {
    for (auto &job : jobs) {
        if (job.jobId == jobId) {
            std::cout << "Bringing job to foreground: " << job.command << "\n";
            kill(job.pid, SIGCONT);
            waitpid(job.pid, nullptr, 0);
            job.running = false;
            return;
        }
    }
    std::cout << "fg: job not found\n";
}

void JobManager::moveToBackground(int jobId) {
    for (auto &job : jobs) {
        if (job.jobId == jobId) {
            std::cout << "Resuming job in background: " << job.command << "\n";
            kill(job.pid, SIGCONT);
            job.running = true;
            return;
        }
    }
    std::cout << "bg: job not found\n";
}

void JobManager::checkCompletedJobs() {
    int status;
    pid_t pid;

    while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
        for (auto &job : jobs) {
            if (job.pid == pid) {
                std::cout << "\n[Job Completed] PID " << pid
                          << " (" << job.command << ")\n";
                job.running = false;
            }
        }
    }
}

```

Code:pipeline.cpp

```cpp
#include "pipeline.h"
#include <unistd.h>
#include <sys/wait.h>
#include <vector>
#include <iostream>

void Pipeline::execute(const Command &cmd) {
    int pipefd[2];
    if (pipe(pipefd) == -1) {
        std::cerr << "Pipe failed\n";
        return;
    }

    pid_t pid1 = fork();
    if (pid1 == 0) {
        dup2(pipefd[1], STDOUT_FILENO);
        close(pipefd[0]);

        std::vector<char*> argv;
        for (auto &a : cmd.args)
            argv.push_back(const_cast<char*>(a.c_str()));
        argv.push_back(nullptr);

        execvp(argv[0], argv.data());
        std::cerr << "Command failed: " << cmd.args[0] << "\n";
        exit(1);
    }

    pid_t pid2 = fork();
    if (pid2 == 0) {
        dup2(pipefd[0], STDIN_FILENO);
        close(pipefd[1]);

        std::vector<char*> argv;
        for (auto &a : cmd.pipeRightArgs)
            argv.push_back(const_cast<char*>(a.c_str()));
        argv.push_back(nullptr);

        execvp(argv[0], argv.data());
        std::cerr << "Command failed: " << cmd.pipeRightArgs[0] << "\n";
        exit(1);
    }

    close(pipefd[0]);
    close(pipefd[1]);
    waitpid(pid1, nullptr, 0);
    waitpid(pid2, nullptr, 0);
}
```

Output:



CONCLUSION

This project demonstrates the effective use of C++ and system-level programming to build a functional Unix-style shell. It integrates command execution, job control, piping, and redirection into a cohesive and efficient shell environment. The implementation reflects strong understanding of core OS concepts and provides a solid foundation for future enhancements.