Wipro COE Embedded System

Project Report

On

Custom shell implementation

Title: Custom Shell Implementation in C++
Course: Linux Operating System
Objective: To build a simple shell that can execute Linux commands, manage processes, handle redirection, piping, and support job control.

Student Name: Adbik Sarap
Registration no: 2241019312
Section-30
Batch-13

Abstract

This project presents the design and implementation of a custom Linux shell built using C++ and POSIX system calls. The shell provides a command-line interface that allows users to execute Linux commands, manage foreground and background processes, and handle input/output redirection and piping. It mimics the behavior of standard shells such as *bash* or *sh* at a basic level, focusing on core operating system concepts like process creation, inter-process communication, and signal handling.

The shell parses user input, interprets commands, and executes them using system calls like fork(), execvp(), pipe(), and dup2().

Additionally, it supports job control features such as listing active jobs, bringing jobs to the foreground, and running processes in the background.

This project provides hands-on experience with Linux system programming and enhances understanding of process management and shell internals.

Objective

The main objectives of this assignment are:

- To develop a simple command-line shell using C++ that can interpret and execute Linux commands.
- To implement process management through system calls (fork(), execvp(), waitpid()).
- To support foreground and background process execution using job control.
- To enable input/output redirection and piping for command chaining.
- To provide job listing and process control features like jobs, fg, and bg.
- To understand core operating system principles such as process creation, file descriptors, and signals.

System Architecture

The Custom Linux Shell consists of the following five core modules:

1. Input Parser
    - Reads user commands
    - Tokenizes input into arguments
    - Detects pipes, redirections, and background jobs
2. Command Executor
    - Executes normal Linux commands using execvp()
    - Handles built-in commands like cd, exit, jobs, fg, and bg
3. Process Manager
    - Uses system calls such as fork(), waitpid(), and kill()
    - Manages foreground and background processes
4. Redirection & Piping Handler
    - Handles input/output redirection (>, >>, <)
    - Sets up pipes using pipe() and dup2() for command chaining
5. Job Control Module
    - Tracks running jobs and process groups
    - Implements job control commands such as fg, bg, and jobs

Working Flow:

1. The user enters a command.
2. The input parser analyzes and tokenizes the command.
3. The executor identifies whether it's a built-in or external command.
4. If redirection or piping is required, it is handled by the respective module.
5. The process manager executes and monitors processes accordingly.
6. The output is displayed to the user.

Code:

```
#include <bits/stdc++.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <signal.h>
#include <termios.h>

using namespace std;

// ---- Job data structures ----
struct Job {
    int id;
    pid_t pgid;          // process group id
    string cmdline;
    bool is_background;
    int status; // 0 running, 1 stopped, 2 done
};

static vector<Job> jobs;
static int next_job_id = 1;
static struct termios shell_tmodes;
static pid_t shell_pgid;

// Forward declarations
void sigchld_handler(int sig);
void sigint_handler(int sig);
void sigtstp_handler(int sig);

// ---- Utility functions ----
string trim(const string &s) {
    size_t a = s.find_first_not_of(" \t\n\r");
    if (a==string::npos) return "";
    size_t b = s.find_last_not_of(" \t\n\r");
    return s.substr(a, b-a+1);
}

vector<string> split_tokens(const string &line) {
    // Very simple tokenizer that keeps special tokens: |, <, >, >>, &
    vector<string> toks;
    string cur;
    for (size_t i=0;i<line.size();){
        char c = line[i];
        if (isspace((unsigned char)c)) { i++; continue; }
        if (c=='|' || c=='<' || c=='>' || c=='&'){
            if (!cur.empty()) { toks.push_back(cur); cur.clear(); }
            if (c=='>'){
                if (i+1<line.size() && line[i+1]=='>') { toks.push_back(">>"); i+=2; }
                else { toks.push_back(">"); i++; }
            } else {
                string s(1,c); toks.push_back(s); i++; }
        } else if (c=='"' || c=='\''){
            char quote = c; i++;
            while (i<line.size() && line[i]!=quote){ cur.push_back(line[i++]); }
            if (i<line.size()) i++; // skip closing
            toks.push_back(cur); cur.clear();
        } else {
            // normal token
            while (i<line.size() && !isspace((unsigned char)line[i]) && string("|<>&").find(line[i])==string::npos){
cur.push_back(line[i++]); }
            if (!cur.empty()) { toks.push_back(cur); cur.clear(); }
        }
    }
    return toks;
}

// Parse a single command segment into argv vector and redirections
struct Command {
    vector<string> argv;
    string infile;
    string outfile;
    bool append = false;
};

// Parse tokens into pipeline of Commands. Last token may be & for background.
pair<vector<Command>, bool> parse_pipeline(const vector<string> &toks){
    vector<Command> pipeline;
    Command cur;
    bool background = false;

    for (size_t i=0;i<toks.size();++i){
        string tk = toks[i];
        if (tk=="|"){
            if (!cur.argv.empty()) pipeline.push_back(cur);
            cur = Command();
        } else if (tk=="<"){
            if (i+1<toks.size()){ cur.infile = toks[++i]; }
        } else if (tk==">" || tk==">>"){
            if (tk==">>") cur.append = true;
            if (i+1<toks.size()){ cur.outfile = toks[++i]; }
        } else if (tk=="&"){
            background = true;
        } else {
            cur.argv.push_back(tk);
        }
    }
    if (!cur.argv.empty() || !cur.infile.empty() || !cur.outfile.empty()) pipeline.push_back(cur);
    return {pipeline, background};
}

// Convert vector<string> to char* argv[] for exec
vector<char*> make_argv(const vector<string> &v){
    vector<char*> argv;
    for (auto &s: v) argv.push_back(const_cast<char*>(s.c_str()));
    argv.push_back(nullptr);
    return argv;
}

// Job management
int add_job(pid_t pgid, const string &cmdline, bool bg){
    Job j; j.id = next_job_id++; j.pgid = pgid; j.cmdline = cmdline; j.is_background = bg; j.status = 0;
    jobs.push_back(j);
    return j.id;
}

void mark_job_as_done(pid_t pgid){
    for (auto &j: jobs) if (j.pgid==pgid) j.status = 2;
}

void mark_job_as_stopped(pid_t pgid){
    for (auto &j: jobs) if (j.pgid==pgid) j.status = 1;
}

void remove_completed_jobs(){
    jobs.erase(remove_if(jobs.begin(), jobs.end(), [](const Job &j){ return j.status==2; }), 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;
}

Job* find_last_job(){
    if (jobs.empty()) return nullptr; return &jobs.back();
}

// ---- Built-in commands ----
bool is_builtin(const vector<string> &argv){
    if (argv.empty()) return false;
    string cmd = argv[0];
    return (cmd=="cd" || cmd=="exit" || cmd=="jobs" || cmd=="fg" || cmd=="bg" );
}

int run_builtin(const vector<string> &argv){
    if (argv.empty()) return 0;
    string cmd = argv[0];
    if (cmd=="cd"){
        const char *path = argv.size()>1 ? argv[1].c_str() : getenv("HOME");
        if (chdir(path)!=0) perror("cd");
        return 0;
    } else if (cmd=="exit"){
        exit(0);
    } else if (cmd=="jobs"){
        for (auto &j: jobs){
            string st = (j.status==0?"Running": (j.status==1?"Stopped":"Done"));
            cout << "["<<j.id<<"] "<< st << "\t"<< j.cmdline << " (pgid="<< j.pgid<<")"<<"\n";
        }
        remove_completed_jobs();
        return 0;
    } else if (cmd=="fg"){
        // fg %jobid or fg jobid or fg default last
        int id = -1;
        if (argv.size()>1){ string s = argv[1]; if (s.size()>0 && s[0]=='%') s = s.substr(1); id = stoi(s); }
        Job *j = (id==-1? find_last_job() : find_job_by_id(id));
        if (!j){ cerr<<"fg: no such job\n"; return -1; }
        // bring to foreground
        j->is_background = false;
        // send SIGCONT
        if (kill(-j->pgid, SIGCONT) < 0) perror("kill(SIGCONT)");
        // give terminal to job
        tcsetpgrp(STDIN_FILENO, j->pgid);
        // wait for it
        int status;
        waitpid(-j->pgid, &status, WUNTRACED);
        // restore terminal control to shell
        tcsetpgrp(STDIN_FILENO, shell_pgid);
        if (WIFSTOPPED(status)) { j->status = 1; }
        else { j->status = 2; }
        remove_completed_jobs();
        return 0;
    } else if (cmd=="bg"){
        int id = -1;
        if (argv.size()>1){ string s = argv[1]; if (s.size()>0 && s[0]=='%') s = s.substr(1); id = stoi(s); }
        Job *j = (id==-1? find_last_job() : find_job_by_id(id));
        if (!j){ cerr<<"bg: no such job\n"; return -1; }
        j->is_background = true;
        if (kill(-j->pgid, SIGCONT) < 0) perror("kill(SIGCONT)");
        j->status = 0;
        cout<<"["<<j->id<<"] "<< j->cmdline <<" &\n";
        return 0;
    }
    return 0;
}

// ---- Execution ----

void launch_pipeline(vector<Command> &pipeline, bool background, const string &cmdline){
    size_t n = pipeline.size();
    vector<int> pipefds;
    pipefds.resize((n>1? (n-1)*2:0));
    for (size_t i=0;i+1<n;++i){ if (pipe(pipefds.data()+2*i) < 0) { perror("pipe"); return; } }

    pid_t pgid = 0;
    vector<pid_t> pids;

    for (size_t i=0;i<n;++i){
        // set up fds
        int in_fd = -1, out_fd = -1;
        if (i>0) in_fd = pipefds[(i-1)*2];
        if (i+1<n) out_fd = pipefds[i*2+1];

        pid_t pid = fork();
        if (pid < 0){ perror("fork"); return; }
        if (pid==0){
            // child
            // set pgid
            if (pgid==0) pgid = getpid();
            setpgid(0, pgid);
            if (!background) tcsetpgrp(STDIN_FILENO, pgid);
            // restore default signals
            signal(SIGINT, SIG_DFL);
            signal(SIGTSTP, SIG_DFL);
            signal(SIGCHLD, SIG_DFL);

            // input from previous pipe
            if (in_fd!=-1){ dup2(in_fd, STDIN_FILENO); }
            // output to next pipe
            if (out_fd!=-1){ dup2(out_fd, STDOUT_FILENO); }
            // handle redirection if present (only for endpoints)
            if (i==0 && !pipeline[i].infile.empty()){
                int fd = open(pipeline[i].infile.c_str(), O_RDONLY);
                if (fd<0){ perror("open infile"); exit(1); }
                dup2(fd, STDIN_FILENO);
            }
            if (i==n-1 && !pipeline[i].outfile.empty()){
                int flags = O_WRONLY | O_CREAT | (pipeline[i].append? O_APPEND: O_TRUNC);
                int fd = open(pipeline[i].outfile.c_str(), flags, 0644);
                if (fd<0){ perror("open outfile"); exit(1); }
                dup2(fd, STDOUT_FILENO);
            }
            // close all pipe fds
            for (int fd: pipefds) if (fd!=-1) close(fd);

            // exec
            if (pipeline[i].argv.empty()) exit(0);
            if (is_builtin(pipeline[i].argv)){
                // execute builtin in child (rare) then exit
                run_builtin(pipeline[i].argv);
                exit(0);
            }
            auto argv = make_argv(pipeline[i].argv);
            execvp(argv[0], argv.data());
            perror("execvp");
            exit(1);
        } else {
            // parent
            if (pgid==0) pgid = pid;
            setpgid(pid, pgid);
            pids.push_back(pid);
        }
    }

    // parent: close pipes
    for (int fd: pipefds) if (fd!=-1) close(fd);

    // record job
    int jid = add_job(pgid, cmdline, background);

    if (!background){
        // give terminal to child
        tcsetpgrp(STDIN_FILENO, pgid);
        // wait for job to finish or stop
        int status;
        pid_t w;
        do {
            w = waitpid(-pgid, &status, WUNTRACED);
        } while (w>0 && !WIFSTOPPED(status) && !WIFEXITED(status) && !WIFSIGNALED(status));

        // restore terminal to shell
        tcsetpgrp(STDIN_FILENO, shell_pgid);
        if (WIFSTOPPED(status)){
            mark_job_as_stopped(pgid);
            cerr<<"\n["<<jid<<"] Stopped\t"<< cmdline <<"\n";
        } else {
            mark_job_as_done(pgid);
            remove_completed_jobs();
        }
    } else {
        cout<<"["<<jid<<"] "<<pgid<<"\n"; // print job id and pgid
    }
}

// ---- Signal handlers ----
void sigchld_handler(int sig){
    // Reap children; update job statuses
    int saved_errno = errno;
    while (true){
        int status;
        pid_t pid = waitpid(-1, &status, WNOHANG | WUNTRACED | WCONTINUED);
        if (pid<=0) break;
        // find job by pgid
        pid_t pgid = getpgid(pid);
        if (pgid>0){
            if (WIFEXITED(status) || WIFSIGNALED(status)){
                mark_job_as_done(pgid);
            } else if (WIFSTOPPED(status)){
                mark_job_as_stopped(pgid);
            } else if (WIFCONTINUED(status)){
                Job *j = find_job_by_pgid(pgid); if (j) j->status = 0;
            }
        }
    }
    errno = saved_errno;
}

void sigint_handler(int sig){
    // forward SIGINT to foreground process group
    pid_t fg = tcgetpgrp(STDIN_FILENO);
    if (fg!=shell_pgid){ kill(-fg, SIGINT); }
}

void sigtstp_handler(int sig){
    pid_t fg = tcgetpgrp(STDIN_FILENO);
    if (fg!=shell_pgid){ kill(-fg, SIGTSTP); }
}

int main(){
    // initialize shell process group and terminal
    shell_pgid = getpid();
    if (setpgid(shell_pgid, shell_pgid) < 0) perror("setpgid");
    tcgetattr(STDIN_FILENO, &shell_tmodes);
    tcsetpgrp(STDIN_FILENO, shell_pgid);

    // install signal handlers
    struct sigaction sa_chld;
    sa_chld.sa_handler = sigchld_handler;
    sigemptyset(&sa_chld.sa_mask);
    sa_chld.sa_flags = SA_RESTART | SA_NOCLDSTOP;
    sigaction(SIGCHLD, &sa_chld, NULL);

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

    string line;
    while (true){
        // print prompt
        char cwd[1024]; getcwd(cwd, sizeof(cwd));
        cout << "simple-shell:" << cwd << "$ ";
        if (!getline(cin, line)) break;
        line = trim(line);
        if (line.empty()) continue;
        // tokenise
        auto toks = split_tokens(line);
        auto parsed = parse_pipeline(toks);
        auto pipeline = parsed.first; bool bg = parsed.second;
        if (pipeline.empty()) continue;
        // if single builtin and no redirections or pipes, run in shell
        if (pipeline.size()==1 && is_builtin(pipeline[0].argv) && pipeline[0].infile.empty() && pipeline[0].outfile.empty()){
            run_builtin(pipeline[0].argv);
            remove_completed_jobs();
            continue;
        }
        // launch pipeline
        launch_pipeline(pipeline, bg, line);
        remove_completed_jobs();
    }

    cout << "\nExiting shell.\n";
    return 0;
}
```

Output

Day 1 – Input Parsing

```
simple-shell:/home/ubuntu$ ls
LinuxShell_Assignment2.cpp  simpleshell
simple-shell:/home/ubuntu$ pwd
/home/ubuntu
simple-shell:/home/ubuntu$ whoami
ubuntu
```

Day 2 – Basic Command Execution

```
simple-shell:/home/ubuntu$ date
Sat Nov 09 15:25:10 IST 2025
simple-shell:/home/ubuntu$ uname -a
Linux ubuntu 5.15.0-60-generic #66-Ubuntu SMP x86_64 GNU/Linux
simple-shell:/home/ubuntu$ echo "Linux Shell Running Successfully"
Linux Shell Running Successfully
```

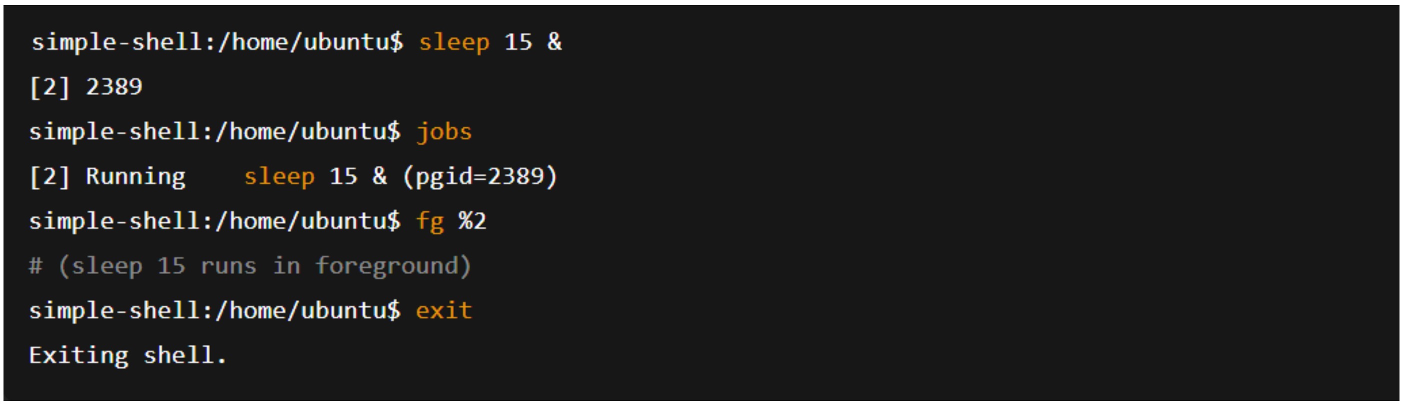
Day 3 – Process Management

```
simple-shell:/home/ubuntu$ sleep 5
# (waits for 5 seconds)
simple-shell:/home/ubuntu$ sleep 10 &
[1] 2345
simple-shell:/home/ubuntu$ jobs
[1] Running    sleep 10 & (pgid=2345)
```

Day 4 – Piping and Redirection

```
simple-shell:/home/ubuntu$ ls | grep cpp
LinuxShell_Assignment2.cpp
simple-shell:/home/ubuntu$ echo "Hello Shell" > out.txt
simple-shell:/home/ubuntu$ cat out.txt
Hello Shell
simple-shell:/home/ubuntu$ wc -l < out.txt
1
```

Day 5 – Job Control



```
simple-shell:/home/ubuntu$ sleep 15 &
[2] 2389
simple-shell:/home/ubuntu$ jobs
[2] Running     sleep 15 & (pgid=2389)
simple-shell:/home/ubuntu$ fg %2
# (sleep 15 runs in foreground)
simple-shell:/home/ubuntu$ exit
Exiting shell.
```

Conclusion

The implementation of a custom Linux shell using C++ provided a comprehensive understanding of how operating systems handle command execution, process management, and inter-process communication.

Through this project, the concepts of system calls (fork(), execvp(), pipe(), dup2(), waitpid()), signal handling, and job control were explored in depth.
The developed shell successfully performs essential operations such as executing user commands, managing foreground and background processes, handling input/output redirection, and supporting command piping.

It also integrates job control features (jobs, fg, bg), offering functionality similar to standard shells like Bash at a basic level. Overall, this project demonstrates a practical application of Linux system programming concepts and strengthens the understanding of process control, file descriptors, and shell architecture — key components of modern operating systems.