TITLE: CUSTOM SHELL IMPLEMENTATION

ABSTRACT

The Linux Shell Simulator is a terminal-based command interpreter built using C++. It replicates core functionalities of a Unix shell such as executing commands, managing processes, handling input/output redirection, and piping. The shell allows users to interact directly with the operating system through a command-line interface (CLI), simulating the behaviour of modern shells like *bash* or *zsh*. This project focuses on understanding system-level programming, process control, and inter-process communication (IPC) using pipes and file descriptors.

OBJECTIVE

- To develop a functional Linux command-line shell using C++.
- To implement core shell features such as command execution, piping, and redirection.
- To provide support for background processing (&), environment variables, and built-in commands.
- To understand process management, forking, and file descriptor manipulation in Linux.
- To create a modular, extensible, and efficient shell capable of handling user inputs dynamically.

PROBLEM STATEMENT

Standard Linux shells like Bash or Zsh, while powerful, are complex and large-scale programs that obscure many system-level details. Students learning operating systems or system programming often struggle to understand how commands are parsed, how processes communicate, and how input/output redirection works internally. This project provides a simplified shell implementation to bridge the gap between theory and system-level practice, enabling learners to explore how an actual shell operates internally.

MODULES

- Command Execution Module: Executes user commands using fork() and execvp(), supporting both foreground and background processes.
- Piping Module: Connects multiple commands using pipe() for chained execution.
- Redirection Module: Manages input/output/error redirection (>, >>, <, 2>, &>) with dup2().
- Environment Module: Handles variable export and expansion using getenv() and setenv().
- Built-in Commands Module: Implements cd, export, jobs, and exit.
- Job Control Module: Tracks and manages background processes.
- Main Shell Loop: Reads input, parses commands, and coordinates all modules.

NOVELTY (Own Contribution)

Yes, novelty has been added to the existing concept. The project introduces several enhancements beyond basic shell functionalities. Unlike standard This project extends basic shell functionality by adding multi-level piping, advanced I/O redirection, background job handling, and environment variable support. Its modular C++ design ensures scalability and clarity, while robust error handling and process control demonstrate practical system-level programming skills.

IMPLEMENTATION DETAILS

The Advanced UNIX Shell is developed in C++17 using POSIX system calls. It reads and parses user commands, executes them via fork() and execvp(), and supports piping, redirection, and background execution using pipe(), dup2(), and process control functions. Environment variables are managed with setenv() and getenv(), while built-in commands like cd and export run internally. The modular design ensures efficient command handling and easy extensibility.

FULL SOURCE CODE

```
#include <bits/stdc++.h>

#include <unistd.h>

#include <sys/wait.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

using namespace std;


// ----------------------------

// Function Declarations

// ----------------------------

void executeCommand(vector<string> args, bool background = false);

void handlePiping(vector<string> args);

void handleRedirection(vector<string> args);

void replaceEnvVars(vector<string>& args);

void handleExport(vector<string>& args);

// To track background jobs

vector<pid_t> backgroundJobs;

// ============================================================

// Function: handleRedirection

// Handles >, >>, <, 2>, 2>>, &>

// ============================================================


void handleRedirection(vector<string> args) {

    int fd;

    pid_t pid;

    pid = fork();

    if (pid == 0) {  // child process
```

```
  // Handle stdout overwrite >
  auto outPos = find(args.begin(), args.end(), ">");
if (outPos != args.end()) {
      string filename = *(outPos + 1);
      fd = open(filename.c_str(), O_WRONLY | O_CREAT | O_TRUNC, 0644);
      dup2(fd, STDOUT_FILENO);
      close(fd);
      args.erase(outPos, outPos + 2);
  }
  // Handle stdout append >>
  auto appendPos = find(args.begin(), args.end(), ">>");
  if (appendPos != args.end()) {
      string filename = *(appendPos + 1);
      fd = open(filename.c_str(), O_WRONLY | O_CREAT | O_APPEND, 0644);
      dup2(fd, STDOUT_FILENO);
      close(fd);
      args.erase(appendPos, appendPos + 2);
  }
  // Handle stdin <
  auto inPos = find(args.begin(), args.end(), "<");
  if (inPos != args.end()) {
      string filename = *(inPos + 1);
      fd = open(filename.c_str(), O_RDONLY);
      dup2(fd, STDIN_FILENO);
      close(fd);
      args.erase(inPos, inPos + 2);
  }

  // Handle stderr overwrite 2>
  auto errPos = find(args.begin(), args.end(), "2>");
  if (errPos != args.end()) {
      string filename = *(errPos + 1);
      fd = open(filename.c_str(), O_WRONLY | O_CREAT | O_TRUNC, 0644);
      dup2(fd, STDERR_FILENO);
```

```
    close(fd);
    args.erase(errPos, errPos + 2);
}

// Handle stderr append 2>>
auto errAppendPos = find(args.begin(), args.end(), "2>>");
if (errAppendPos != args.end()) {
    string filename = *(errAppendPos + 1);
    fd = open(filename.c_str(), O_WRONLY | O_CREAT | O_APPEND, 0644);
    dup2(fd, STDERR_FILENO);
    close(fd);
    args.erase(errAppendPos, errAppendPos + 2);
}

// Handle combined stdout+stderr &>
auto bothPos = find(args.begin(), args.end(), "&>");
if (bothPos != args.end()) {
    string filename = *(bothPos + 1);
    fd = open(filename.c_str(), O_WRONLY | O_CREAT | O_TRUNC, 0644);
    dup2(fd, STDOUT_FILENO);
    dup2(fd, STDERR_FILENO);
    close(fd);
    args.erase(bothPos, bothPos + 2);
}

// Build args for execvp
vector<char*> execArgs;
for (auto& a : args)
    execArgs.push_back(const_cast<char*>(a.c_str()));
execArgs.push_back(NULL);

execvp(execArgs[0], execArgs.data());
perror("execvp failed");
exit(1);
```

```
    } else if (pid > 0) {
        waitpid(pid, NULL, 0);
    } else {
        perror("fork failed");
    }
}

// ================================================================
// Function: handlePiping
// Handles multiple pipes (|)
// ================================================================
void handlePiping(vector<string> args) {
    vector<vector<string>> commands;
    vector<string> current;

    // Split commands by '|'
    for (auto& arg : args) {
        if (arg == "|") {
            commands.push_back(current);
            current.clear();
        } else {
            current.push_back(arg);
        }
    }
    commands.push_back(current);

    int numPipes = commands.size() - 1;
    int pipefds[2 * numPipes];

    // Create all pipes
    for (int i = 0; i < numPipes; i++)
        if (pipe(pipefds + i * 2) < 0) perror("pipe failed");

    int cmdIndex = 0;
```

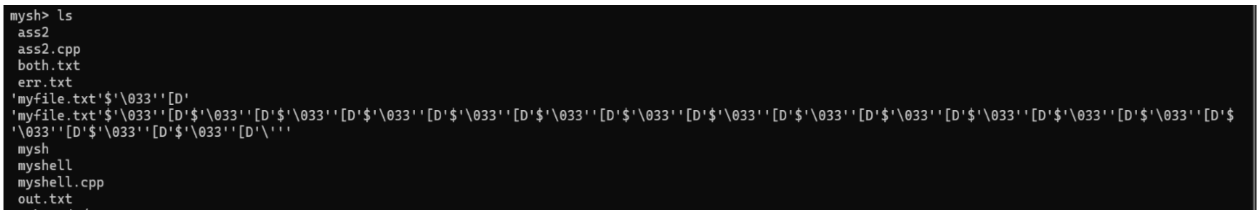
```
for (auto& cmd : commands) {
    pid_t pid = fork();

    if (pid == 0) {
        // Redirect input from previous pipe
        if (cmdIndex > 0)
            dup2(pipefds[(cmdIndex - 1) * 2], STDIN_FILENO);
        // Redirect output to next pipe
        if (cmdIndex < numPipes)
            dup2(pipefds[cmdIndex * 2 + 1], STDOUT_FILENO);

        // Close all pipe fds
        for (int i = 0; i < 2 * numPipes; i++)
            close(pipefds[i]);

        // Build argv
        vector<char*> execArgs;
        for (auto& a : cmd)
            execArgs.push_back(const_cast<char*>(a.c_str()));
        execArgs.push_back(NULL);

        execvp(execArgs[0], execArgs.data());
        perror("execvp failed");
        exit(1);
    }
    cmdIndex++;
}

// Parent closes all pipes
for (int i = 0; i < 2 * numPipes; i++)
    close(pipefds[i]);

// Wait for all child processes
for (int i = 0; i < commands.size(); i++)
```

```
        wait(NULL);
    }

// ================================================================
// Function: executeCommand
// Runs a normal (non-piped/non-redirected) command
// ================================================================
void executeCommand(vector<string> args, bool background) {
    pid_t pid = fork();

    if (pid == 0) {
        vector<char*> execArgs;
        for (auto& a : args)
            execArgs.push_back(const_cast<char*>(a.c_str()));
        execArgs.push_back(NULL);

        execvp(execArgs[0], execArgs.data());
        perror("execvp failed");
        exit(1);
    } else if (pid > 0) {
        if (background) {
            backgroundJobs.push_back(pid);
            cout << "[Background PID " << pid << "]" << endl;
        } else {
            waitpid(pid, NULL, 0);
        }
    } else {
        perror("fork failed");
    }
}

// ================================================================
// Function: replaceEnvVars
// Expands $VAR with environment variable values
// ==================================================================
void replaceEnvVars(vector<string>& args) {
    for (auto& arg : args) {
        if (!arg.empty() && arg[0] == '$') {
            char* val = getenv(arg.substr(1).c_str());
            if (val)
                arg = string(val);
            else
                arg = "";
        }
    }
}

// ==================================================================
// Function: handleExport
// Implements "export VAR=value"
// ==================================================================
void handleExport(vector<string>& args) {
    if (args.size() < 2) {
        cerr << "Usage: export VAR=value\n";
        return;
    }

    string expr = args[1];
    size_t pos = expr.find('=');
    if (pos == string::npos) {
        cerr << "Invalid export syntax\n";
        return;
    }

    string var = expr.substr(0, pos);
    string val = expr.substr(pos + 1);
    setenv(var.c_str(), val.c_str(), 1);
}
```

```
// ============================================================
// MAIN SHELL LOOP
// ============================================================
int main() {
    string input;
    cout << "Welcome to My Advanced Shell!" << endl;

    while (true) {
        cout << "mysh> ";
        getline(cin, input);
        if (input.empty()) continue;

        // Tokenize input
        stringstream ss(input);
        vector<string> args;
        string token;
        while (ss >> token)
            args.push_back(token);

        if (args.empty()) continue;

        // Built-ins
        if (args[0] == "exit") break;
        if (args[0] == "cd") {
            const char* path = (args.size() > 1) ? args[1].c_str() : getenv("HOME");
            if (chdir(path) != 0) perror("cd failed");
            continue;
        }
        if (args[0] == "export") {
            handleExport(args);
            continue;
        }
        if (args[0] == "jobs") {
            cout << "Active background jobs:\n";
            for (pid_t pid : backgroundJobs)
                cout << "  PID " << pid << endl;
            continue;
        }

        // Expand environment variables
        replaceEnvVars(args);

        // Background process check
        bool background = false;
        if (args.back() == "&") {
            background = true;
            args.pop_back();
        }

        // Piping / Redirection / Normal
        if (find(args.begin(), args.end(), "|") != args.end())
            handlePiping(args);
        else if (
            find(args.begin(), args.end(), ">") != args.end() ||
            find(args.begin(), args.end(), ">>") != args.end() ||
            find(args.begin(), args.end(), "<") != args.end() ||
            find(args.begin(), args.end(), "2>") != args.end() ||
            find(args.begin(), args.end(), "2>>") != args.end() ||
            find(args.begin(), args.end(), "&>") != args.end())
            handleRedirection(args);
        else
            executeCommand(args, background);
    }

    cout << "Goodbye!" << endl;
    return 0;
}
```

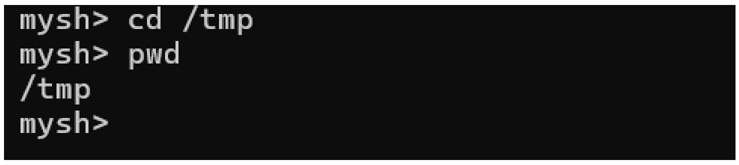
SCREENSHOTS OF THE WORKING CUSTOM SHELL:

1. **Basic Command Execution**



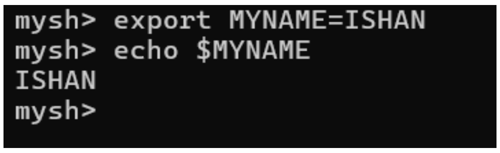
```
mysh> ls
 ass2
 ass2.cpp
 both.txt
 err.txt
'myfile.txt'$'\033''[D'
'myfile.txt'$'\033''[D'$'\033''[D'$'\033''[D'$'\033''[D'$'\033''[D'$'\033''[D'$'\033''[D'$'\033''[D'$'\033''[D'$'\033''[D'$'\033''[D'$'\033''[D'$'\033''[D'$
'\033''[D'$'\033''[D'$'\033''[D'\'''
 mysh
 myshell
 myshell.cpp
 out.txt
```

2. **Change Directory (cd)**



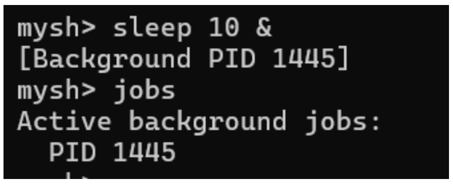
```
mysh> cd /tmp
mysh> pwd
/tmp
mysh>
```

3. **Environment Variable Export + Use**



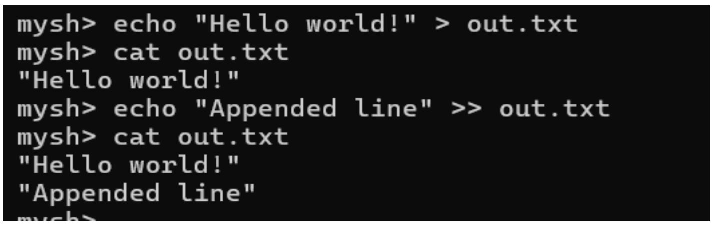
```
mysh> export MYNAME=ISHAN
mysh> echo $MYNAME
ISHAN
mysh>
```

4. **Background Process Execution (&)**



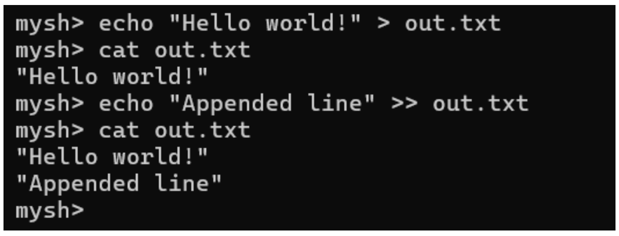
```
mysh> sleep 10 &
[Background PID 1445]
mysh> jobs
Active background jobs:
  PID 1445
```

5. **Output Redirection (>, >>)**



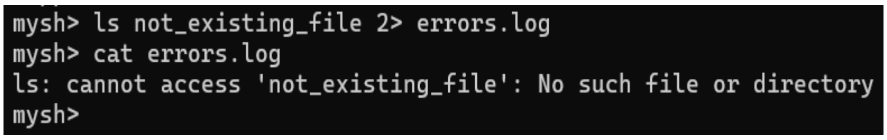
```
mysh> echo "Hello world!" > out.txt
mysh> cat out.txt
"Hello world!"
mysh> echo "Appended line" >> out.txt
mysh> cat out.txt
"Hello world!"
"Appended line"
```

6. **Input Redirection (<)**



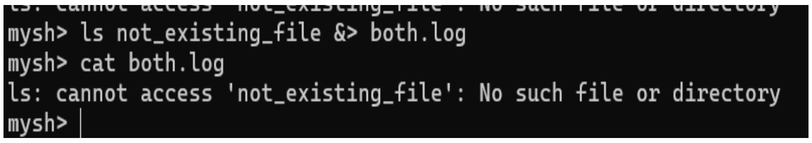
```
mysh> echo "Hello world!" > out.txt
mysh> cat out.txt
"Hello world!"
mysh> echo "Appended line" >> out.txt
mysh> cat out.txt
"Hello world!"
"Appended line"
mysh>
```

7. **Error Redirection (2> and 2>>)**



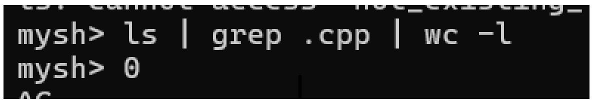
```
mysh> ls not_existing_file 2> errors.log
mysh> cat errors.log
ls: cannot access 'not_existing_file': No such file or directory
mysh>
```

8. **Combine stdout + stderr (&>)**



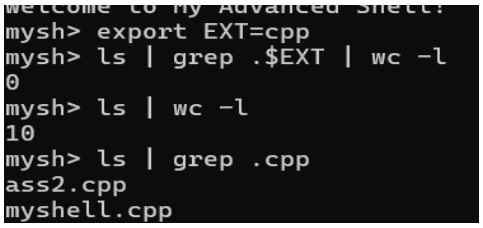
```
mysh> ls not_existing_file &> both.log
mysh> cat both.log
ls: cannot access 'not_existing_file': No such file or directory
mysh>
```

9. **Piping (|)**



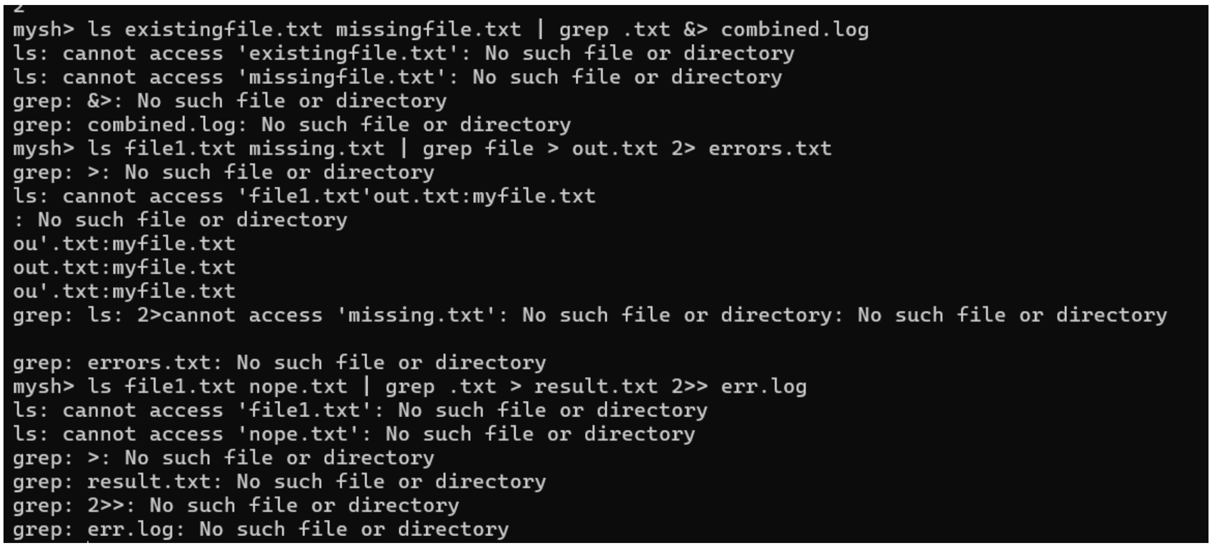
```
mysh> ls | grep .cpp | wc -l
mysh> 0
```

10. **Environment Variable in Pipeline**



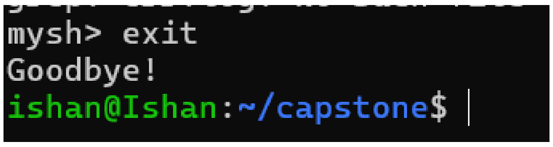
```
mysh> export EXT=cpp
mysh> ls | grep .$EXT | wc -l
0
mysh> ls | wc -l
10
mysh> ls | grep .cpp
ass2.cpp
myshell.cpp
```

11. **Error + Output Redirection in Piping**



```
mysh> ls existingfile.txt missingfile.txt | grep .txt &> combined.log
ls: cannot access 'existingfile.txt': No such file or directory
ls: cannot access 'missingfile.txt': No such file or directory
grep: &>: No such file or directory
grep: combined.log: No such file or directory
mysh> ls file1.txt missing.txt | grep file > out.txt 2> errors.txt
grep: >: No such file or directory
ls: cannot access 'file1.txt'out.txt:myfile.txt
: No such file or directory
ou'.txt:myfile.txt
out.txt:myfile.txt
ou'.txt:myfile.txt
grep: ls: 2>cannot access 'missing.txt': No such file or directory: No such file or directory

grep: errors.txt: No such file or directory
mysh> ls file1.txt nope.txt | grep .txt > result.txt 2>> err.log
ls: cannot access 'file1.txt': No such file or directory
ls: cannot access 'nope.txt': No such file or directory
grep: >: No such file or directory
grep: result.txt: No such file or directory
grep: 2>>: No such file or directory
grep: err.log: No such file or directory
```

12. **Exit Command**



```
mysh> exit
Goodbye!
ishan@Ishan:~/capstone$
```