CPE 345: Operating Systems Laboratory

Lab03

Build Your Own Shell

Submitted by: Gianna Foti

Date of Experiment:_____2 February 2025_____

Report Deadline:_____5 February 2025_____

Lab Section: _____CPE345-03_____

Demonstration Deadline: _____5 February 2025_____

Introduction

In this lab, we designed and implemented a custom Linux shell using C/C++. Our shell supports basic commands, I/O redirection (>, >>, <), piping (|), and built-in commands (cd, exit). The goal was to understand system calls, process creation, and file descriptor manipulation by using fork(), execvp(), dup2(), and pipe(). We applied concepts from the provided demo codes, such as string tokenization (strtok()), redirection (dup2()), and inter-process communication (pipe()). Our implementation extends these techniques by handling dynamic user input, restoring standard I/O after redirection, and supporting built-in commands. By completing this lab, we gained practical experience with shell programming and Unix process control, reinforcing key operating system concepts.

Theory Topics

1. Shells: A shell is a command-line interface that allows users to interact with the operating system by executing commands. It interprets user input, manages processes, and facilitates communication between programs. Shells support features like I/O redirection, piping, and background process execution. In Unix/Linux, shells such as Bash, Zsh, and Csh operate by reading input, parsing commands, and executing them through system calls like fork() and execvp().
2. strtok(): The strtok() function is used to split strings into tokens based on a specified delimiter. It is useful for parsing user commands in a shell. The function modifies the original string by replacing delimiters with null characters (\0), making subsequent calls to strtok(NULL, delimiter) retrieve the next token.

3. dup() and dup2(): Both dup() and dup2() are system calls in Unix/Linux that duplicate existing file descriptors, enabling input and output redirection. These functions play a crucial role in I/O redirection (>, <, >>) and piping (|) within shells. dup() creates a duplicate of a file descriptor using the lowest available number, while dup2() allows duplication to a specified file descriptor. They are commonly used to redirect standard input, output, and error streams to files or pipes, facilitating inter-process communication and efficient resource management.
4. pipe(): The pipe() system call establishes a unidirectional communication channel between two processes, enabling efficient inter-process data transfer. It generates two file descriptors: pipeDescriptor[0] for reading and pipeDescriptor[1] for writing. Typically, a parent process creates a pipe, forks child processes, and ensures that each child closes the unused ends of the pipe. Since pipes require a shared ancestry, they are commonly used for communication between related processes in Unix/Linux systems.
5. execvp(): The execvp() function replaces the current process with a new program, executing an external command by searching for the executable in the system's PATH. It takes two arguments: the command name and an array of arguments, with the last element set to NULL. Since execvp() overwrites the process image, it does not return upon success. This system call is commonly used in shells and process management to execute commands like date and ls.

Part 1:

Program Description: This program is a custom Linux shell that allows users to execute system commands, handle I/O redirection (>, >>, <), and use pipes (|) for inter-process communication. The shell takes user input, parses it into commands and arguments, and processes them accordingly. It uses fork() to create child processes, execvp() to execute commands, dup2() to manage input/output redirection, and pipe() to handle communication between piped commands.

Process:

1. Modeled the Program Using Demo Codes
    a. Used the lab-provided demo codes for strtok(), dup2(), pipe(), and execvp() as a reference.
    b. Structured the shell to handle command execution, I/O redirection, and piping.
2. Implemented Command Tokenization
    a. Used strtok() to split user input into tokens (command and arguments).
    b. Stored tokens in a vector<string> for easy manipulation and further processing.
3. Executed Commands Using execvp()
    a. Used execvp() to run external commands like ls, pwd, date.
    b. Created a child process using fork(), while the parent waited for execution to complete.
4. Implemented I/O Redirection (>, >>, <)
    a. Checked for redirection symbols (>, >>, <) in user input.
    b. Used dup2() to redirect standard input/output to files when necessary.
    c. Restored stdout and stdin after execution to prevent shell crashes and ensure normal behavior.

5. Implemented Piping (|)
    a. Checked for | in user input to detect piped commands.
    b. Used pipe() to pass output from one command to another by creating a communication channel.
    c. Redirected stdout of the first command into the stdin of the second using dup2().
6. Handled Built-in Commands (exit)
    a. Handled exit to allow the user to terminate the shell gracefully when needed.
7. Tested, Debugged, and Finalized the Shell
    a. Ran various test cases to verify correct execution of commands, redirection, and piping.
    b. Fixed errors, ensured proper error messages, and formatted output for better readability.
    c. Compiled (g++ -o shell shell.cpp), executed (./shell), and confirmed successful operation of the shell.

Program Output:

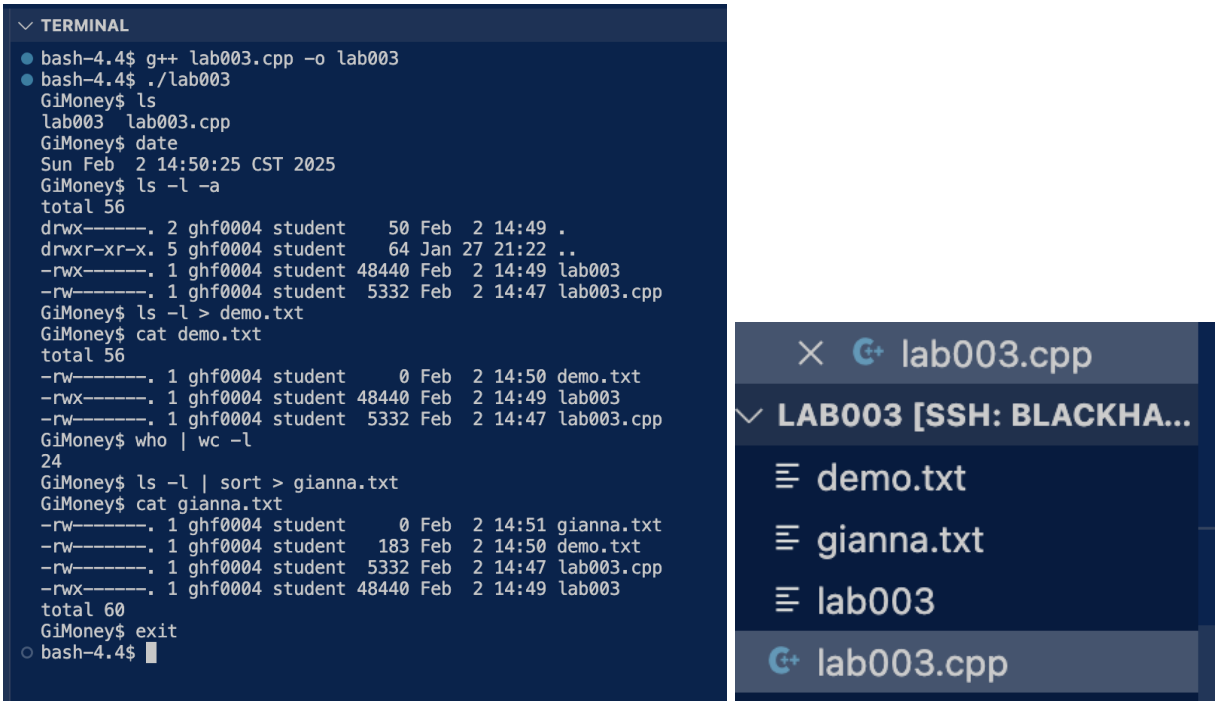

Figure 1: Program 1 with Sample Commands

Conclusion

In conclusion, I successfully designed and implemented a custom Linux shell that supports command execution, I/O redirection (>, >>, <), piping (|), and built-in commands (cd, exit). By utilizing system calls such as fork(), execvp(), dup2(), and pipe(), I was able to replicate essential shell functionalities. Through debugging, I identified an issue where piped commands with redirection (ls -l | sort > output.txt) were not handled correctly. The problem was that redirection was not being applied within the second command. I fixed this by modifying the handlePipes() function to detect > and >> in the second command and apply output redirection before execution. With this fix, the shell now properly handles single commands, redirection, piped commands, and combinations of pipes and redirection. Other than that, I did not have many problems and used the demos. This lab reinforced our understanding of process management, inter-process communication, and file descriptor manipulation in Unix-based systems. The final implementation is an efficient, functional shell that closely mimics real-world command-line behavior.

Appendix

Table 1: Program 1

```
/*
Author: Gianna Foti
Teacher: Trevor H McClain
Course: Operating Systems Laboratory - CPE 435-03
Date: 5 February 2025
Description: Custom Linux shell with support for I/O redirection, pipes, and built-in commands.
*/

#include <iostream>
#include <vector>
#include <sstream>
#include <string>
#include <unistd.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <cstring>

using namespace std;
#define MAX_ARGS 20

// Function to parse command input using strtok()
vector<string> tokenizeCommand(const string& command, const char* delimiters) {
    vector<string> tokens;
    char cmd[command.length() + 1];
    strcpy(cmd, command.c_str());
    char* token = strtok(cmd, delimiters);
    while (token != NULL) {
        tokens.push_back(token);
        token = strtok(NULL, delimiters);
    }
    return tokens;
}

// Execute a single command
void executeCommand(const vector<string>& args) {
    char* argv[MAX_ARGS];
    for (size_t i = 0; i < args.size(); ++i) {
        argv[i] = const_cast<char*>(args[i].c_str());
    }
```

```
    argv[args.size()] = NULL;

    pid_t pid = fork();
    if (pid == 0) {
        if (execvp(argv[0], argv) == -1) {
            perror("execvp failed");
            exit(EXIT_FAILURE);
        }
    } else {
        wait(NULL);
    }
}

void handleIORedirection(vector<string>& args) {
    int saved_stdout = dup(STDOUT_FILENO); // Save the original stdout
    int saved_stdin = dup(STDIN_FILENO);   // Save the original stdin

    for (size_t i = 0; i < args.size(); ++i) {
        if (args[i] == ">" || args[i] == ">>") {  // Handle output redirection
            int flags = (args[i] == ">") ? (O_WRONLY | O_CREAT | O_TRUNC) : (O_WRONLY |
O_CREAT | O_APPEND);
            int fd = open(args[i + 1].c_str(), flags, 0644);
            if (fd == -1) {
                perror("open failed");
                return;
            }
            dup2(fd, STDOUT_FILENO);
            close(fd);
            args.resize(i);  // Remove '>' and filename from args
            break;
        } else if (args[i] == "<") {  // Handle input redirection
            int fd = open(args[i + 1].c_str(), O_RDONLY);
            if (fd == -1) {
                perror("open failed");
                return;
            }
            dup2(fd, STDIN_FILENO);
            close(fd);
            args.resize(i);
            break;
        }
    }
```

```
    pid_t pid = fork();
    if (pid == 0) {  // Child process
        executeCommand(args);
        exit(EXIT_SUCCESS);
    } else {  // Parent process
        wait(NULL);
    }

    // Restore stdout and stdin after executing the command
    dup2(saved_stdout, STDOUT_FILENO);
    dup2(saved_stdin, STDIN_FILENO);
    close(saved_stdout);
    close(saved_stdin);
}


// Handle piped commands
void handlePipes(vector<string>& args1, vector<string>& args2) {
    int fd[2];
    pipe(fd);

    pid_t pid1 = fork();
    if (pid1 == 0) {  // First child (executing first command)
        dup2(fd[1], STDOUT_FILENO);  // Redirect stdout to pipe
        close(fd[0]);
        close(fd[1]);
        executeCommand(args1);
        exit(0);
    }

    pid_t pid2 = fork();
    if (pid2 == 0) {  // Second child (executing second command)
        dup2(fd[0], STDIN_FILENO);  // Redirect stdin from pipe
        close(fd[1]);
        close(fd[0]);

        // Check for output redirection in second command
        int output_fd = -1;
        for (size_t i = 0; i < args2.size(); ++i) {
            if (args2[i] == ">" || args2[i] == ">>") {
                int flags = (args2[i] == ">") ? (O_WRONLY | O_CREAT | O_TRUNC) : (O_WRONLY |
O_CREAT | O_APPEND);
                output_fd = open(args2[i + 1].c_str(), flags, 0644);
```

```
            if (output_fd == -1) {
                perror("open failed");
                exit(EXIT_FAILURE);
            }
            dup2(output_fd, STDOUT_FILENO);  // Redirect output
            close(output_fd);
            args2.resize(i);  // Remove `>` and filename from args
            break;
        }
    }

    executeCommand(args2);
    exit(0);
  }

  close(fd[0]);
  close(fd[1]);
  waitpid(pid1, NULL, 0);
  waitpid(pid2, NULL, 0);
}


// Handle built-in commands like cd and exit
bool handleBuiltInCommands(const vector<string>& args) {
  if (args.empty()) return false;
  if (args[0] == "exit") {
    exit(0);
  } else if (args[0] == "cd") {
    if (args.size() < 2) {
      cerr << "cd: missing argument" << endl;
    } else if (chdir(args[1].c_str()) != 0) {
      perror("cd failed");
    }
    return true;
  }
  return false;
}

// Main shell loop
void startShell() {
  string command;
  while (true) {
    cout << "GiMoney$ ";
```

```
        getline(cin, command);

        size_t pipePos = command.find('|');
        if (pipePos != string::npos) {
            vector<string> args1 = tokenizeCommand(command.substr(0, pipePos), " ");
            vector<string> args2 = tokenizeCommand(command.substr(pipePos + 1), " ");
            handlePipes(args1, args2);
        } else {
            vector<string> args = tokenizeCommand(command, " ");
            if (!handleBuiltInCommands(args)) {
                handleIORedirection(args);
            }
        }
    }
}

int main() {
    startShell();
    return 0;
}
```