Nhóm tối đa 3 sv
Deadline: 23:00, 10-10-2019
Nộp bài: MSSV1_MSSV2_MSSV3.zip(rar)
Bài nộp gồm:

- **Mã nguồn**
- **Readme: hướng dẫn sử dụng + ghi chú**
- **Docs: mô tả tổ chức/thiết kế của đồ án**

Project 1—Simple Shell

This project consists of designing a C program to serve as a shell interface that accepts user commands and then executes each command in a separate process. Your implementation will support input and output redirection, as well as pipes as a form of IPC between a pair of commands. Completing this project will involve using the UNIX fork(), exec(), wait(), dup2(), and pipe() system calls and can be completed on Linux system.

I. Overview

A shell interface gives the user a prompt, after which the next command is entered. The example below illustrates the prompt *osh>* and the user's next command: cat prog.c. (This command displays the file prog.c on the terminal using the UNIX cat command.)

```
osh>cat prog.c
```

One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line (in this case, cat prog.c) and then create a separate child process that performs the command. Unless otherwise specified, the parent process waits for the child to exit before continuing. However, UNIX shells typically also allow the child process to run in the background, or concurrently. To accomplish this, we add an ampersand (&) at the end of the command. Thus, if we rewrite the above command as

```
osh>cat prog.c &
```

the parent and child processes will run concurrently

The separate child process is created using the fork() system call, and the user's command is executed using one of the system calls in the exec() family.

A C program that provides the general operations of a command-line shell is supplied as below listing code:

```
#include <stdio.h>
#include <unistd.h>
```

```
#define MAX LINE 80 /* The maximum length command */
int main(void)
{
	char *args[MAX LINE/2 + 1]; /* command line arguments */
	int should run = 1; /* flag to determine when to exit program */
	while (should run) {
		printf("osh>");
		fflush(stdout);
		/**
		* After reading user input, the steps are:
		* (1) fork a child process using fork()
		* (2) the child process will invoke execvp()
		* (3) parent will invoke wait() unless command included &
		*/
		}
	return 0;
}
```

The main() function presents the prompt *osh->* and outlines the steps to be taken after input from the user has been read. The main() function continually loops as long as should run equals 1; when the user enters exit at the prompt, your program will set should run to 0 and terminate.

This project is organized into several parts:
1. Creating the child process and executing the command in the child
2. Providing a history feature
3. Adding support of input and output redirection
4. Allowing the parent and child processes to communicate via a pipe

II. Executing Command in a Child Process

The first task is to modify the main() function so that a child process is forked and executes the command specified by the user. This will require parsing what the user has entered into separate tokens and storing the tokens in an array of character strings. For example, if the user enters the command ps -ael at the *osh>* prompt, the values stored in the args array are:

```
args[0] = "ps"
args[1] = "-ael"
args[2] = NULL
```

This args array will be passed to the execvp() function, which has the following prototype:

```
execvp(char *command, char *params[])
```

Here, command represents the command to be performed and params stores the parameters to this command. For this project, the execvp() function should be invoked as execvp(args[0], args). Be sure to check whether the user included & to determine whether or not the parent process is to wait for the child to exit.

III. Creating a History Feature

The next task is to modify the shell interface program so that it provides a history feature to allow a user to execute the most recent command by entering !!. For example, if a user enters the command ls -l, she can then execute that command again by entering !! at the prompt. Any command executed in this fashion should be echoed on the user's screen, and the command should also be placed in the history buffer as the next command. Your program should also manage basic error handling. If there is no recent command in the history, entering !! should result in a message "No commands in history."

IV. Redirecting Input and Output

Your shell should then be modified to support the '>' and '<' redirection operators, where '>' redirects the output of a command to a file and '<' redirects the input to a command from a file. For example, if a user enters

```
osh>ls > out.txt
```

the output from the ls command will be redirected to the file out.txt. Similarly, input can be redirected as well. For example, if the user enters

```
osh>sort < in.txt
```

the file in.txt will serve as input to the sort command.

Managing the redirection of both input and output will involve using the dup2() function, which duplicates an existing file descriptor to another file descriptor. For example, if fd is a file descriptor to the file out.txt, the call

```
dup2(fd, STDOUT FILENO);
```

duplicates fd to standard output (the terminal). This means that any writes to standard output will in fact be sent to the out.txt file.

You can assume that commands will contain either one input or one output redirection and will not contain both. In other words, you do not have to be concerned with command sequences such as sort < in.txt > out.txt.

V. Communication via a Pipe

The final modification to your shell is to allow the output of one command to serve as input to another using a pipe. For example, the following command sequence

```
osh>ls -l | less
```

has the output of the command ls -l serve as the input to the less command. Both the ls and less commands will run as separate processes and will communicate using the UNIX pipe() function. Perhaps the easiest

way to create these separate processes is to have the parent process create the child process (which will execute ls -l). This child will also create another child process (which will execute less) and will establish a pipe between itself and the child process it creates. Implementing pipe functionality will also require using the dup2() function as described in the previous section. Finally, although several commands can be chained together using multiple pipes, you can assume that commands will contain only one pipe character and will not be combined with any redirection operators.