CS 370: Operating Systems

Assignment 2

Fall 2021

Instructions

- The skeleton code has been provided to you in the file simple-shell.c.
- While submitting, please zip the file and name the zip as *yourRollNo_PA2.zip.* For example, 22100027_PA2.zip
- **Do Not** plagiarize. Any cases of plagiarism will be forwarded to the disciplinary committee
- The Deadline for this assignment is 1st November, 2021. Make sure to start early to avoid any last minute inconveniences

The Unix Shell

This assignment 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 means of IPC between a pair of commands. Completing this project will involve using the UNIX fork ( ), exec (), wait ( ), dupe (), and pipe ( ) system calls and can be completed on any Linux, UNIX, or macOS system.

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 in 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 thqt, *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. For the purpose of this assignment, we will only be using execvp () system

A C program that provides the general operations of a command-line shell is supplied in the figure below. 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.

```
#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;
}
```

Figure 3.36

This assignment is divided into the following parts:

1) Creating a child process and executing commands in the child **(15 marks)**
2) Providing a history feature **(15 marks)**
3) Adding support of input and output redirection **(20 marks)**

4) Creating environment variables, assigning them values and then reading those values. **(25 marks)**
5) Making the parent and child process communicate via pipes **(25 marks)**

Part 1 – Executing Command in a child process

The first task is to modify the main() function in Figure 3.36 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 (args in Figure 3.36).

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.

Part 2 – 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 -1, 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.

Part 3 – Redirecting Output

Your shell should then be modified to support the '>' redirection operators, where '>' redirects the output of a command to 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.

Managing the redirection of 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 only one output redirection

Part 4 – Creating Environment Variables

An environment variable is a variable which is a part of the environment in which a process runs. All processes running in the environment are

able to access the values of the environment variables within the environment and then use them for different purposes such as performing calculations using their values, storing their values in a file or simply displaying their values on the shell.

In this task, we modify our simple shell so that it allows the creation of environment variables and then lets us access them. For simplicity, we will only be displaying the values of our created environment variables on our shell and storing their values in a file. The following command is used to create environment variables within the shell:

VAR=hello_world

where VAR is the name of the environment variable and *hello_world* is its value. For simplicity, we will assume that our environment variables can only store values of type string and our program can have only 10 environment variables at max. Since our environment variables can store only string values, we do not need to add quotes around values. Hence, you are only expected to cater to inputs of the above format, without any quotation marks.

Once the environment variable has been created, you should be able to access it and print its value. This can be done via the following command:

echo $VAR

This should display the string literal, *hello_world* on your terminal. If an environment variable does not exist, the terminal should display *NotFound*. For example:

echo $ABC,

should display *NotFound.*

Now that our environment variable is being displayed on the shell, we will use output redirection to store its value in the file. For example

echo $VAR > newFile

should create a file by the name of *newFile* and store the literal *hello_world* in this file. You should be able to verify this by typing the following command:

cat newFile

This command should display *hello_world* on the terminal screen.

Part 5 – 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 described in Section 3.7.4. Perhaps the easiest way to create these separate processes is to have the parent process create the child process (which will execute ls -1). 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.