Experiment Report

— UNIX Shell and History Feature

Name:Wang Xingyi StudentID:5140309531

November 8, 2016

1 Introduction

This project consists of modifying a C program which serves as a shell interface that accepts user commands and then executes each command in a separate process. One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line, 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 – as well by specifying the ampersand(&) at the end of the command.

2 Running environment

⟶ Ubuntu 16.04

3 Experimental procedure

3.1 Simple Shell

The outline of simple shell is given in the textbook, so we first need to complete the *setup* function, which is used to get the input command. As the textbook says, *setup()* reads in the next command line, separating it into distinct tokens using whitespace as delimiters. *setup()* modifies the args parameter so that it holds pointers to the null-terminated strings that are the tokens int the most recent user command line as well as a NULL pointer, indicating the end of the argument list, which comes after the string pointers that have been assigned to args. Full source code of *setup* function is available online. The we need to solve the multi-process problem. We get the command in parent process and execute it in child process. The child process will invoke *execvp()*, and the parent process can either invoke the *setup()* or wait – if background == 1.

3.2 Creating a Child Process

As noted above, the *setup()* function loads the contents of the args array with the command specified by the user. This args array will be passed to the execvp() function, which can be used as:

execvp(args[0], args);

3.3 Creating a History Feature

The next task is to modify the program as that it provides a history feature that allows the user access up to the 10 most recently entered commands. The user will be able to list these commands when he/she presses {Control}{C}, which is the SIGINT signal. Once a signal has been generated by the occurrence of a certain event, we can handle it by ignoring the signal, using the default signal handler, or providing a separate signal-handling function. So the function *handle_SIGINT()* is to provide the history feature. As the textbook writes, we can use sigaction to catch a signal, and it is surely a stronger way to achieve this function. But I found *signal()* more convenient to provide a separate signal-handling function, and this function can easily solve the sigaction-remaining problem.

The command history is stored in an array named *history*. It's a 10-by-10 array which can save 10 recent commands by turn. *nextPosition* is used to mark the next position to save a new command. We can move forward by 1 command using *nextPosition = (nextPosition + 1) % 10*, and backward using *nextPosition = (nextPosition + 9) % 10*. The length of each command should also be preserved.

When I reached this step of the task, here came a serious problem – communication among processes. We can know whether one command is true only after it has been executed. If *execvp* returns -1, then we can mark this command wrong. All the commands are executed in child process, but we need to know the correctness of the command which is called by *r* command in parent process so we can decide whether it will be stored in *history*. To solve this problem, we have two different ways. One is to save the correctness of each command in local storage as *success.txt*. This file is read in parent process and written in child process, so they can communicate via this. But this method seems a little bit stupid as we have to create a *.txt* file. So there's another solution, which is to realize communication among processes by using shared memory. We can get a piece of shared memory by using *shmget()* function, and *shmat()* is used to attach a variable to this memory, then we can use it whether in parent process or child process.

3.4 Executing Commands

Most of the commands can be implemented by *execvp()* function, so the work of a shell is more like string-processing. We read the input and put it

in a buffer and then separate the string into command and parameter part, and save it in the args array. But some commands are implemented by the shell itself, such as *cd* command. However this command is rather easy to realize. We have *chdir* function for us to change the current directory. Either relative path or absolute path is available. But we must attention that this *chdir* function must be used in the parent process, which means being used before *pid = fork();*, or it won't be able to change the path permanently.

The last function is about & command. If a command is followed by & in the end of the line, it means we can run child process in the background. It is easy to create a concurrent child process, and we don't tell the parent process to wait, then we can continue the parent process, which is to read the next command and execute it. But the parent process didn't wait the concurrent process, which makes it a zombie process when it is done. So we need to call *wait(NULL)* to collect the "corpse" of this process. The function *wait(NULL)* will return the ID of the process it has collected. When *background == 0*, if the ID we collect is not the same as the process ID we just create, keep waiting. Because this means we have collected a background-running child process. This will successfully solve the concurrent process problem.

4 Conclusion and Discussion

By doing this project, I learn more about what Shell is used to do and how those functions are implemented in the code level. And it let me have a better understanding of processes and concurrent process executing. Signal handling in Linux is also very interesting and powerful. But there are still many shortcomings in my shell, such as not being able to realize pipe communication among processes. I'll continue to learn about UNIX Shell and improve mine.