Práctica 3

UNIX Shell and History Feature

Sistemas Operativos

Dra. Mireya Paredes López

Fecha:

Fecha de entrega: 10/03/2020

Durante la clase anterior, vimos como se crean nuevos procesos en Linux usando la llamada al sistema ***fork().*** El objetivo de esta práctica es entender esta llamada a sistema mediante la creación de un shell creado por ustedes.

Ustedes podrán más información acerca de **fork()** en el capítulo 3 del libro de Operating System Concepts (Silberschatz) para resolver los siguientes problemas de programación.

Entregables:

En el syllabus del curso vienen las especificaciones de lo entregables.

Lab 3: UNIX Shell and History Feature

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. This project can be completed on any Linux, UNIX, or Mac OS X system.

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. This is similar in functionality to the new process creation illustrated in Figure 3.10 (attached at the end of this document). 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 the parent and child processes will run concurrently.

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

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 (as described in Section 3.3.1 of the book).

A C program that provides the general operations of a command-line shell is supplied in *Figure 3.36.* The **main()** function presents the prompt **osh**-> and outlines the steps to be taken after the 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 two parts: (1) creating the child process and executing the command in the child, and (2) modifying the shell to allow a history feature.

```
#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) if command included &, parent will invoke 
            wait()
                  */
      }
      return 0;
 }
```

Figure 3.36 Outline of a simple shell.

Part I— Creating 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 an **&** to determine whether or not the parent process is to **wait for the child to exit**.

Part II—Creating a History Feature

The next task is to modify the shell interface program so that it provides a history feature that allows the user to access the most recently entered commands. The user will be able to access up to 10 commands by using the feature. The commands will be consecutively numbered starting at 1, and the numbering will continue past 10. For example, if the user has entered 35 commands, the 10 most recent commands will be numbered 26 to 35.
The user will be able to list the command history by entering the command:

osh> history

at the **osh>** prompt. As an example, assume that the history consists of the commands (from most to least recent):

```
ps, ls -l, top, cal, who, date
```

The command history will output:

```
6 ps
5 ls -l
4 top
3 cal
2 who
1 date
```

Your program should support two techniques for retrieving commands from the command history:

- When the user enters !!, the **most recent** command in the history is executed.
- When the user enters a **single !** followed by an integer **N**, the **Nth** command in the history is executed.

Continuing our example from above, if the user enters **!!**, the `ps` command will be performed; if the user enters `!3`, the command `cal` will be executed. Any command executed in this fashion should be echoed on the user's screen. The command should also be placed in the history buffer as the next command.

The program should also manage basic error handling. If there are no commands in the history, entering !! should result in a message ***"No commands in history".*** If there is no command corresponding to the number entered with the single !, the program should output "**No such command in history**".

Extra:

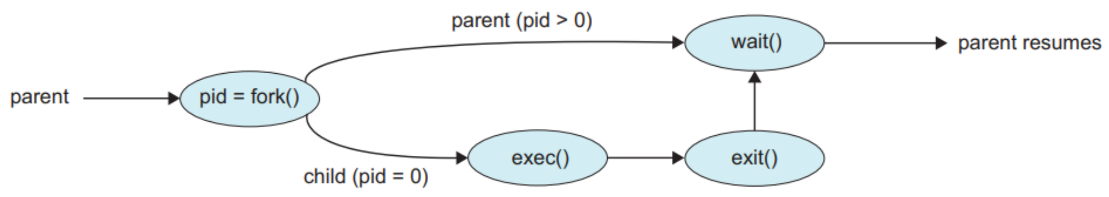

Figure 3.10 Process creation using the `fork()` system call.