Lab Assignment 1: Shell and System Calls

Notes

You need to work on this project individually.
This project must be implemented in C.

Objectives

- To get familiar with Linux and its programming environment.
- To understand the relationship between OS command interpreters (shells), system calls, and the kernel.
- To learn how processes are handled (i.e., starting and waiting for their termination).
- To learn robust programming and modular programming.

Overview

You are required to design and implement a C program that serves as a shell interface. The program accepts user commands and then executes each command in a separate process. The commands are those that can be executed on a Unix-like system such as Linux and Mac OS X. The shell displays a user prompt at which a user can enter the command to be executed. The example below illustrates the prompt `shell>` 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
```

A Unix shell is a command-line interpreter that provides a traditional user interface for the Unix operating system and for Unix-like systems. The command can be executed in one of these forms:

- **foreground:** One implementation of this shell interface mode is to have the parent process (shell) 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. The parent process waits for the child to exit before continuing to read input from the user. This is similar in functionality to the new process creation illustrated in Figure 1.
- **background:** In this case, the Unix shell allows the child process (which is executing the command) to run in the background, or concurrently. To differentiate between it and the foreground mode, 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.

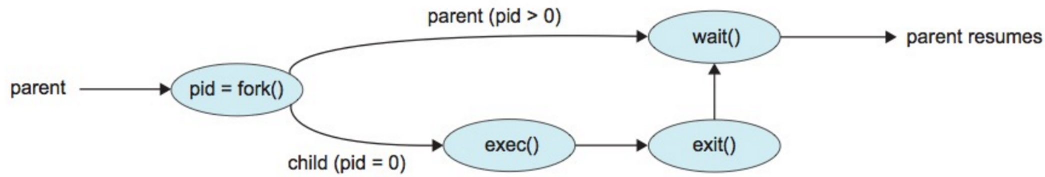

Figure 1: Process creation using the fork() system call (Credit: Abraham Silberschatz et al. Operating system concepts. 2012.).

You will need to create the child process using the fork() system call. To execute the user's command, you need to use one of the system calls in the exec() family (e.g. `execv()`). Below is a C program that provides the general operations of a command-line shell. The main() function presents the prompt `shell>` 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("shell>");
          fflush(stdout);

          /**
           * After reading user input, the steps are:
           * (1) fork a child process using fork()
           * (2) the child process will invoke execv()
           * (3) if command included &, parent will invoke wait()
          */
     }

     return 0;
}
```

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.

PART I: Basic Shell Function

For each command that the user enters at the prompt, you are required to parse what the user has entered into separate tokens and store the tokens in an array of character strings (`args` in the above sample code). For example, if the user enters the command `ps -ael` at the `shell>` prompt, the values stored in the `args` array are:

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

The above explained details describe the interactive mode, in which the user enters commands at the displayed prompt and terminates when the user enters the `exit` command at the prompt. You will also need to support a batch mode of execution. In the shell batch mode, you start the shell by calling your shell program and specifying a batch file to execute the commands included in it. For example:

```
./myShell batchFile.txt
```

This batch file (`batchFile.txt` in the example) contains the list of commands (on separate lines) that the user wants to execute. In batch mode, you should not display a prompt, however, you will need to echo each line you read from the batch file (print it) before executing it. This feature in your program is to help debugging and testing your code. Your shell terminates when the end of the batch file is reached, an `exit` command is among the listed commands in the batch file, or the user types `Ctrl-D`.
Your first task is to modify the main() function listed above so that a child process is forked and executes the command specified by the user. After parsing the command entered by the user and adding it to the `args` array, you will need to pass `args` to the `execv()` function, which has the following prototype:

```
execv(char *pathToCommand, char *params[]);
```

Here, `pathToCommand` represents the path to the command to be performed and `params` stores the parameters to this command. For this project, the `execv()` function should be invoked as `execv(path+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 this 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

```
history
```

at the `shell>` 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:

1. When the user enters `!!`, the most recent command in the history is executed.
2. 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.

Handling Errors

Your shell should handle errors in a decent way. Your C program should not core dump, hang indefinitely, or prematurely terminate. Your program should check for errors and handle them by printing an understandable error message and either continue processing or exit, depending upon the situation.
The following cases are considered errors and you need to handle them in your program:

- An incorrect number of command line arguments to your shell program.
- The batch file does not exist or cannot be opened.

In the following cases, you should print a message to the user (stderr) and continue reading the following commands:

- A command does not exist or cannot be executed.
- A very long command line (over 80 characters).
- 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."

These cases are not errors, however, you still need to handle them in your shell program:

- An empty command line.
- Multiple white spaces on a command line.
- White space before or after the `&` character.
- The input batch file ends without and `exit` command or the user types `Ctrl-D` as command in the interactive mode.
- If the `&` character appears in the middle of a line, then the job should not be placed in the background; instead, the `&` character is treated as one of the job arguments.

Hints

- For reading input lines, check `fgets()`.
- For reading input files, check `fopen()`.
- Do not forget to check the return codes of routines for any errors. `perror()` can be useful to display the error.
- Use `fork()` to create a new child process.
- Use `execv()` to execute a command. Note that `execv()` only returns if an error has occurred. The return value is -1, and `errno` is set to indicate the error.
- For testing your program, you can use `execvp()`. However, the delivered version should use `execv()`.
- use `wait()` or `waitpid()` to force the parent process (i.e., your shell program) to wait for its child process to finish.
- You do not need to handle (setting or checking) environment variables in your shell.

Testing

- Make sure that your code will run on the lab machines. Do not make assumptions that will make the code execute only on your personal machines. For example do not hard code the `$PATH` values.
- Use our sample test cases to test your program. You will need to come up with more test cases to test the different command types, and the robustness of your program. First test that your program can start up simple programs (e.g. a calculator, a text editor). Then test it by running standard UNIX/Linux utilities, such as `ls`, `cat`, `cp`, and `rm`. Next, test passing parameters to these commands. We will have many other test cases to test your program.

Deliverables

- Complete source code in **C**, commented thoroughly and clearly. You also need to submit a makefile so we can use it to compile your code.
- A report that describes the following: (1) how your code is organized, (2) its main functions, and (3) how to compile and run your code.
- Sample runs.
- All deliverables are to be put in one directory named lab1_XX, where XX is your ID and then zipped.
- You need to send your code to CS333F16@gmail.com on October 15 before 11:59 PM. The subject line should be: "Assignment 1 - SID:XX".