



Assignment 1

[20 marks]

Submission Deadline: **THURSDAY, JUNE 14, at 11:59PM**

ASSIGNMENT TASK

Design a C program to serve as a shell interface that accepts user commands, then execute each command in a separate process. In addition, provide a history feature that allows the user to access the most recently entered commands.

ASSIGNMENT DETAILS:

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 `CSCI3120>` and the user's next command: `cat prog.c`. (This command displays the file `prog.c` on the terminal using the UNIX `cat` command.)

```
CSCI3120> 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. UNIX shells typically also allow the child process to run in the background, or concurrently. Unless otherwise specified, the parent process waits for the child to exit before continuing. To enable the parent process to run concurrently with the child process, we add an ampersand (`&`) at the end of the command. Thus, if we rewrite the above command as

```
CSCI3120> 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 shown in the code snippet below. The `main()` function presents the prompt `CSCI3120>` 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 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("CSCI3120>");
        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;
}

```

Outline of simple shell

Part I— Creating a Child Process

The first task is to modify the `main()` function in the code shown above 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 above code). For example, if the user enters the command `ps -ael` at the `CSCI3120>` 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

```
history
```

at the `CSCI3120>` 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 N^{th} 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."

SUBMISSION DETAILS:

Upload a single zip file containing:

1. your code (consisting of your c file),
2. an image file (.jpg or .png) showing a screen capture of your program in action, and
3. a readme.txt file containing names of the group members and instructions to compile your program

to the Assignment 1 dropbox on Brightspace. Do NOT submit the executable.

SUBMISSION DEADLINE:

Thursday, 14-June-2018, 11:59PM

GRADING CRITERIA:

- Code compiles using gcc. Assignment instructions are properly followed.
 - Note that you will get a zero in this assignment if your code does not compile
- Users commands are properly parsed and executed in a child process [8 marks]
- History feature works properly [8 marks]
- Basic error handling is done as specified in Part II of the assignment [2 marks]
- Code is well written, properly formatted and commented [2 marks]