CS 744 - Autumn 2020

Programming Assignment 1

Writing a custom UNIX Shell using system calls

Release date: 19th August 2020
Due date: 30th August 2020, 5 pm (via moodle)

Prerequisites

- You should write the program in C.
- Prepare yourself to know about the following system calls: fork, exec (and it's variants), wait and exit, calls to manipulate files (opening, reading and writing files using **open**, **read**, **write** and **close** system calls), **pipe**, **dup2**.
  Here is a link to the list of UNIX System Calls.
- Use Ubuntu 18.04 and above only. You must stick to Linux system calls only (from glibc). If it does not compile on our systems, you don't get any points.

Description

In this assignment you will code a program that will use a process control API such as `fork`, `exec` and `wait` to build a command line interpreter (a lightweight shell). It should repeatedly take commands as input from the user via the terminal and execute each command.

- The shell program when invoked, should display a command prompt as follows:
  "**mySheII>** " and then wait for user input.
- When input is entered, the program should first check whether input is a legal (one of the commands given below) command. If not, print a message "**Illegal command or arguments**" and go back to display the prompt and wait for user input. *Do not exit your program*.
- If the command is legal and has the right number of arguments, execute it using one of the two strategies shown below, wait until the command is finished (using the **`wait`** system call in the parent), and then print the command prompt and wait for the next input.

Each command can be coded to run in one of two ways:

I. To run the commands that are already present as executable files (for example: ls), you must **fork** a new child process and use **`execve`** (or variants of **exec**) to execute the corresponding executable, while the main process waits for the child to finish. When the child exits, the wait call returns to the parent process which will then print the command prompt and wait for the user to enter commands again.
NOTE: Do not use the **system()** function in your shell anywhere to execute commands.

II. For commands which do not have an executable already and mentioned below, you must code these commands in the child process using only POSIX compliant UNIX system calls. We will indicate for each command whether there is an executable that you can use or you must implement it from scratch.

A description of the custom functions to be written and executed using the custom shell is as follows,

1. Implement the following command **`checkcpupercentage <pid>`**
   This command accepts a process identifier as an argument and returns the percentage CPU used by the process with identifier **`<pid>`** in user mode and system mode. Details regarding the CPU usage of every process is stored in a file **`/proc/[pid]/stat`** where [pid] is the process identifier of the process. Look up **`/proc/[pid]/stat`** in the man page for **`proc`** for more information (**`man proc`**).

   Example usage:
   myShell> **`checkcpupercentage 12810`**
   `user mode cpu percentage: 38%`
   `system mode cpu percentage: 11%`

   The output MUST match the syntax EXACTLY as shown above.
   To implement this:
   a. First, fork a new child process to execute this command.
   b. The child process should read the stat file for the specified pid at **`/proc/[pid]/stat`**. The 14th and the 15th word in this file should denote the time (in number of clocks ticks) spent in user mode and the time (in number of clocks ticks) spent in system mode respectively.
   c. The **/proc/stat/** file also reports total CPU time (in ticks) used so far. This is the sum of all numbers that follows "CPU" (note: CPU, not CPU1 etc).
      Check the following for more details regarding **/proc** proc(5) - Linux manual page.
   d. Printout percentage of execution in **`user mode`** and **`system mode`** to the terminal with labels "**`user mode cpu percentage: `**" and "**`system mode cpu percentage:`**"

      Hint: Get the total number of CPU ticks for a second to calculate percentage.

   e. Exit the child, so that the parent who is waiting will be released.

2. Many bash commands are executable programs stored in /bin or /usr/bin. For example, the popular UNIX command **`ps`** that is used to list processes running in the system is one such executable. Implement the following command **`checkresidentmemory <pid>`**
   This command should accept a **pid** of a process and output the resident set size which is the physical memory that the task has used *(output a single number and no other text)*. Look up the manpage of **`ps`** and find out how to obtain the resident set size, without showing the header.

   Example usage:
   myShell> **`check resident memory 13410`**
   `538290`

   To do this:
   a. The main process should fork a child and do a **`wait`** on it. Once this call returns, print the command prompt and wait for user input.
   b. The child process should use **`execve`** on the **`ps`** command with the appropriate flags to obtain the necessary output for the given **`pid`**. Note that **`execve`** will print the results to the terminal by itself.

c. In **`execve`**, the first argument is a string which points to the path to the executable (which here is just **`ps`** since ps is in your $PATH[1])
d. The second argument is a char** which is an array of strings that are to be given as arguments to the executable.
e. However, note that for most shell commands, the first argument is treated as the name with which the executable was called. The code to do **`ps -u user`** would look something like:

```
char *eargs[] = {"ps", "-u", "user"};
execve("ps", eargs, NULL);
```

3. Recall the **`ls`** is a bash command that lists all the files and folders in the given path. Using the **`ls`** executable, implement a command **`listFiles`** on your command line interface which will save all the files and folders, in the current working directory to a file named **files.txt**. If the file exists already, overwrite its contents.

   Examples:
   myShell> **`listFiles`**

   This should make a files.txt in the same directory, and should contain something similar to:
   `Desktop Downloads Documents Music Photos Videos`

   Hint: dup2 system call can be used for this task.

4. The shell utility **`sort`** sorts the input alphabetically, line by line, given to it from STDIN and outputs it to STDOUT. Add a **`sortFile`** command in your shell using **`sort`**, but this should accept input from a file. So instead of giving input for sort from STDIN, you should be able to write several lines into a file and then type **`sortFile [file]`** to get the file's contents sorted line by line printed into STDOUT (**`sort`** prints to STDOUT by default so you don't have to change anything regarding the output). For this do what you did for the previous task, but instead of closing and replacing STDOUT with a file, do it for STDIN.

   Examples:
   myShell> **`sortFile aFileThatHasBCA.txt`**
   `A`
   `B`
   `C`

5. Your shell should be able to execute two commands simultaneously by spawning two child processes while the parent shell process waits for both of them to complete. Implement this by an operator **`;`** which when called as **`command1 ; command2`** should execute both commands in parallel (**not sequential**). The shell should only display the prompt for the next command once both of the commands are done executing. You can assume that there won't be more than two commands executed in parallel. To do this:

[1]PATH is an environmental variable in Linux and other Unix-like operating systems that tells the shell which directories to search for executable files in response to commands issued by a user. Since the directory /usr/bin is in the $PATH for most Linux systems, you just need to mention the commands and not the entire path to it.

a. Fork two processes and execute the required commands in both of them. Save the pid of both the processes.
b. The parent should call a wait on both the pids. Lookup the syntax of 'waitpid' to implement this.

Example:

```
myshell> checkresidentmemory 13410 ; listFiles
Desktop Downloads Documents Music Photos Videos
325235
myshell>
```

6. Implement a custom command **executeCommands** which takes a filename as an argument and executes all the commands listed in the file sequentially in the same order as given in the file.

Example:

```
myShell> executeCommands commandsFile.txt
.
.
.
myShell>
```

7. Your shell should be able to handle Unix signals. Signals mentioned below should be handled by your shell:
   a. **SIGINT:** whenever the user presses **Ctrl-C** then your shell program should show the message that "**the program is interrupted, do you want to exit [Y/N]**". If Y is pressed then terminate otherwise keep the shell running.
   b. **SIGTERM**: When a user sends a SIGTERM (kill -15 <pid>) the program shows the message: "Got SIGTERM-Leaving" and exits.

8. **I/O Redirection**

Your shell must support i/o-redirection on both STDIN and STDOUT.
i.e., the following command line input,

a. myShell> programName arg1 arg2 < inputfile > outputfile

should execute the program **programName** with arguments **arg1** and **arg2**, the STDIN FILE stream replaced by **inputfile** and the STDOUT FILE stream replaced by **outputfile**.

b. myShell> command1 < inputfile | command2 > outputfile

should execute the command **command1** with the STDIN FILE stream replaced by **inputfile** and then pipe its output to the input for the **command2** with the STDOUT FILE stream replaced by **outputfile**.
For more clarity about **pipe()** check this out pipe(2) - Linux manual page.

With output redirection, if the redirection character is **>** then the **outputfile** is created if it does not exist and truncated if it does. If the redirection token is **>>** then **outputfile** is created if it does not exist and appended to if it does.

Note: Assume that the redirection symbols, **<** , **>**, **>>** will be delimited from other command line arguments by white space - one or more spaces.

I/O redirection can be accomplished in the child process immediately after the **fork** and before the **exec** command. At this point, the child has inherited all the file handles of its parent and still has access to a copy of the parent memory. Thus, it will know if redirection is to be performed or not. If it does, then the child process will change its stdin and/or stdout file streams. Note that this will only affect the child file descriptors and not the parent.

You can use **open** to create file descriptors for **inputfile** and/or **outputfile** and then use **dup** or **dup2** to replace either the stdin descriptor (**STDIN_FILENO** from **unistd.h**) or the stdout descriptor (**STDOUT_FILENO** from **unistd.h**).

UPDATE [Aug 25]

To test pipe utility and I/O redirection you have to implement these type of commands:

cat <file_name> **|** **grep** <pattern_string>
cat <file_name> **>** output_file.txt
grep <pattern_string> **<** input_file.txt **>** output_file.txt

Use **exec()** to implement **cat** and **grep** individually.

9. You should be able to exit the shell by typing in a command named **`exit`**.

   In all of the commands, make sure to reap the dead child (using **`wait`**) processes and avoid zombies. Make sure that if execution of any commands results in an error, the error messages thrown by the program are printed. Explicitly check that you do not leave any zombie processes on exit from the shell.

References

- UNIX System Calls
- Comprehensive List of Linux Syscalls
- https://www.cs.princeton.edu/courses/archive/fall04/cos217/lectures/21pipes.pdf
- https://linuxhint.com/fork-system-call-linux/
- Exec System Call in C – Linux Hint

Submission details

a. The assignment must be done using C.
b. Submit a single tarball named <your rollnumber>-pa1.tar.gz.
c. The tarball should contain:
   i. A README that shows how to compile and execute the shell.
   ii. A Makefile that will be used to compile the program.

iii. All the source files (includes and C sources) needed to compile. Please do NOT have any dependency on include files other than standard C includes. Especially Microsoft files and so on.

d. The output of each command MUST BE EXACTLY as specified. Any deviations could result in you not getting any marks since it will be auto evaluated.

Grading Rubric

1. **Part 1** (10 marks)
2. **Part 2** (10 marks)
3. **Part 3** (10 marks)
4. **Part 4** (10 marks)
5. **Part 5** (10 marks)
6. **Part 6** (05 marks)
7. **Part 7** (05 marks)
8. **Part 8** (30 marks [15 + 15])