CS4375: Theory of Operating Systems **Instructor: Dr. Deepak Tosh**

Fall 2021 **Due Date:** 11/17/2021, Midnight

Assignment # 2A

In this assignment, you will implement a *command line interpreter (CLI)* or, as it is more commonly known, a *shell*. The shell should operate in this basic way: when you type in a command (in response to its prompt), the shell creates a child process that executes the command you entered and then prompts for more user input when it has finished.

The shells you implement will be similar to, but simpler than, the one you run every day in Unix. If you don't know what shell you are running, it's probably **bash**. One thing you should do on your own time is learn more about your shell, by reading the man pages or other online materials.

Note: This is a multi-phased assignment, where you will be incrementally developing advanced functionalities to make it work similar to the shell program. It is better to execute this program in a UNIX environment as most of the shell commands will work better there. We encourage to use a Ubuntu 20.04 LTS VM to develop and test this assignment, if you are presently working on a Windows.

Program Spec

Your shell "**minershell**" is going to be an interactive loop, i.e. repeatedly prints the prompt "**minersh**$ ". Your program parses the input, executes the command specified on that line of input, and waits for the command to finish. This is repeated until the user types **exit**.

```
prompt>./minershell
minersh$
```

At this point, **minersh** is running, and ready to accept commands. Type away!

Prerequisites:

1. Familiarize yourself with the various process related system calls in Linux: **fork**, **exec**, **exit** and **wait**. The "man pages" in Linux are a good source of learning.
2. Familiarize yourself with simple shell commands in Linux like **pwd**, **wc**, **echo**, **cat**, **sleep**, **ls**, **ps**, **top**, **grep** and so on. To implement these commands in your shell, you must simply "**exec**" these existing executables, and not implement the functionality yourself.
3. Understand the **chdir** system call in Linux (see **man chdir**). This will be useful to implement the **cd** command in your shell.

Building a simple shell

Task-1: Basic shell implementation

You will be building a simple shell to run simple Linux commands. A shell takes in user input, forks one or more child processes using the fork system call, calls **exec** from these children to execute user commands, and reaps the dead children using the **wait** system call. Your shell must execute all simple Linux commands like **ls**, **wc, cat**, **echo, pwd** and **sleep**. These commands are readily available as executables on Linux, and your shell must simply invoke the existing executable. It is important to note that you must implement the shell functionality yourself, using the **fork**, **exec**, and **wait** system calls. You must not use library functions like **system** which implement shell commands by invoking the Linux shell—doing so defeats the purpose of this assignment!

To implement this task, your job is to simply invoke exec on any command that the user gives as input. If the Linux command execution fails due to incorrect arguments, an error message will be printed on screen by the executable, and your shell must move on to the next command. If the command itself does not exist, then the exec system call will fail, and you will need to print an error message on screen and move on to the next command.

Your shell must gracefully handle errors. An empty command (typing return) should simply cause the shell to display a prompt again without any error messages. For all incorrect commands or any other erroneous input, the shell itself should not crash. It must simply notify the error and move on to prompt the user for the next command.

For all commands, you must take care to terminate and carefully reap any child process the shell may have spawned. Please verify this property using the **ps** command during testing. When the forked child calls exec to execute a command, the child automatically terminates after the executable completes. However, if the **exec** system call failed for some reason, the shell must ensure that the child is terminated suitably. **When not running any command, there should only be the one main shell process running in your system, and no other children**.

To **test this lab**, run a few common Linux commands in your shell, and check that the output matches what you would get on a regular Linux shell. Further, check that your shell is correctly reaping dead children, so that there are no extra zombie processes left behind in the system.

Assumptions –

A skeleton code **minershell.c** is provided to get you started. This program reads input and tokenizes it for you. You must add code to this file to execute the commands found in the "**tokens**". You may assume that the input command has no more than **1024** characters, and no more than 64 tokens. Further, you may assume that each token is no longer than **64** characters.

Task-2: Enabling change of directory

Once you complete the execution of simple commands, proceed to implement support for the **cd** command in your shell using the **chdir** system call. The command **cd** must cause the shell process to change its working directory, and **cd ..** should take you to the parent directory. You are not required to support other variants of **cd** that are available in the various Linux shells. For example, just typing cd will take you to your home directory in some shells; you need not support such complex features.

Note that you **must NOT spawn a separate child process to execute the chdir** system call, but must call **chdir** from your shell itself, because calling **chdir** from the child will change the current working directory of the child whereas we wish to change the working directory of the main parent shell itself. Any incorrect format of the cd command should result in your shell printing Shell: Incorrect command to the display and prompting for the next command.

Grading:

The assignment will be graded on following items:

1. Completeness and correctness on your program.
2. Inclusion of appropriate evidence with enough test scenarios (in form of screenshots) in a report
3. Programs' readability and correctness
4. Hurdles faced while implementing
5. References (including links where you found some sample code.