COMP 4736 **Lab 03: Building a Shell (Part A)**

In this lab, you will build a simple shell to execute user commands, much like the `bash` shell in Linux. This lab will deepen your understanding of various concepts of process management in Linux.

Before You Begin

- 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. You can access the man pages from the Linux terminal by typing `man fork`, `man 2 fork` and so on. You can also find several helpful links online.

- It is important to understand the different variants of these system calls. In particular, there are many different variants of the `exec` and `wait` system calls; you need to understand these to use the correct variant in your code. For example, you may need to invoke wait differently depending on whether you need to block for a child to terminate or not.

- Familiarize yourself with simple shell commands in Linux like `echo`, `cat`, `sleep`, `ls`, `ps`, `top`, `grep` and so on. To run these commands from your shell, you must simply "`exec`" these existing executables, and not implement the functionality yourself.

- Understand the `chdir` system call in Linux (see `man chdir`). This will be useful to implement the `cd` command in your shell.

- Understand the concepts of foreground and background execution in Linux. Execute various commands on the Linux shell both in foreground and background, to understand the behavior of these modes of execution.

- Understand the concept of signals and signal handling in Linux. Understand how processes can send signals to one another using the `kill` system call. Read up on the common signals (SIGINT, SIGTERM, SIGKILL, …), and how to write custom signal handlers to "catch" signals and override the default signal handling mechanism, using interfaces such as `signal()` or `sigaction()`.

- Read the problem statement fully, and build your shell incrementally, part by part. Test each part thoroughly before adding more code to your shell for the next part.

Warm-up Exercises

Below are some "warm-up" exercises you can do before you start implementing the shell.

1. Write a program that forks a child process using the fork system call. The child should print "I am child" and exit. The parent, after forking, must print "I am parent". The parent must also reap the dead child before exiting. Run this program a few times. What is the order in which the statements are printed? How can you ensure that the parent always prints after the child?

2. Write a program that forks a child process using the fork system call. The child should print its PID and exit. The parent should wait for the child to terminate, reap it, print the PID of the child that it has reaped, and then exit. Explore different variants of the wait system call while you write this program.

3. Write a program that uses the exec system call to run the "`ls`" command in the program. First, run the command with no arguments. Then, change your program to provide some arguments to "`ls`", e.g., "`ls -l`". In both cases, running your program should produce the same output as that produced by the "`ls`" command. There are many variants of the `exec` system call. Read through the man pages to find something that suits your needs.

4. Write a program that uses the `exec` system call to run some command. Place a print statement just before and just after the `exec` statements, and observe which of these statements is printed. Understand the behavior of print statement placed after the `exec` system call statement in your program by giving different arguments to `exec`. When is this statement printed and when is it not?

5. Write a program where the fork system call is invoked *N* times, for different values of *N*. Let each newly spawned child print its PID before it finishes. Predict the number of child processes that will be spawned for each value of *N*, and verify your prediction by actually running the code. You must also ensure that all the newly created processes are reaped by using the correct number of wait system calls.

6. Write a program where a process forks a child. The child runs for a long time, say, by calling the `sleep` function. The parent must use the `kill` system call to terminate the child process, reap it, print a message, and exit. Understand how signals work before you write the program.

7. Write a program that runs an infinite while loop. The program should not terminate when it receives SIGINT (`Ctrl+C`) signal, but instead, it must print "I will run forever" and continue its execution. You must write a signal handler that handles SIGINT in order to achieve this. So, how do you end this program? When you are done playing with this program, you may need to terminate it using the SIGKILL signal.

Part A: A Simple Shell

We will first build a simple shell to run simple Linux commands. A shell takes in user input, forks a child process using the `fork` system call, calls `exec` from this child to execute the user command, reaps the dead child using the `wait` system call, and goes back to fetch the next user input. Your shell must execute ***all*** simple Linux command given as input, e.g., `ls`, `cat`, `echo` and `sleep`. These commands are readily available as executables on Linux, and your shell must simply invoke the corresponding executable, using the user input string as argument to the `exec` system call. 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!

Your simple shell must use the string "`$ `" as the command prompt. Your shell should interactively

accept inputs from the user and execute them. In this part, the shell should continue execution indefinitely until the user hits `Ctrl+C` to terminate the shell. You can assume that the command to run and its arguments are separated by one or more spaces in the input, so that you can "tokenize" the input stream using spaces as the delimiters. You are given starter code `my_shell.c` which reads in user input and "tokenizes" the string for you. For this part, you can assume that the Linux commands are invoked with simple command-line arguments, and without any special modes of execution like background execution, I/O redirection, or pipes. You need not parse any other special characters in the input stream. Please do not worry about corner cases or overly complicated command inputs for now; just focus on getting the basics right.

Note that it is not easy to identify if the user has provided incorrect options to the Linux command (unless you can check all possible options of all commands), so you need not worry about checking the arguments to the command, or whether the command exists or not. 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. In either case, errors must be suitably notified to the user, and you must move on to the next command.

A skeleton code `my_shell.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.

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 <directoryname>` must cause the shell process to change its working directory, and `cd ..` should take you to the parent directory. You need not 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.

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 commands such as `ps` or `top` 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 did not succeed 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.

Sample Testing Criteria

- Shows correct prompt "`$ `".
- Runs continuously (until `Ctrl-C`).
- Handles empty commands properly.
- Handles valid Linux commands properly.
  - With no argument
  - With valid argument(s)
  - With invalid argument(s)
- Handles unknown commands properly.
- Terminates child process properly.
  - With valid Linux commands and valid argument(s)
  - With valid Linux commands and invalid argument(s)
  - With unknown commands
- Handles "`cd`" command properly.
  - With valid argument
  - With invalid argument

Submission Instructions

You must submit a single shell code file as `my_shell.c`.

— End of Lab 03 —