# CS 333: Operating Systems Lab
# Autumn 2017

# Lab 3: Building your own shell[1]

## Goal

In this lab you will learn how to build a simple interactive shell of your own.

## Before you begin

*You must use C for this lab. You may use C++ for the string-handling parts of the assignment.*

- Read about and understand the structure of a typical shell: read an input string from the user, `fork` a child, `exec` a command executable, `wait` for the child, and repeat the process. Read about all variants of the `wait` and `exec` system calls - there are several of them, and you need to pick different variants under different circumstances.
- Read the complete lab before starting, especially the tips and guidelines towards the end of the lab carefully, to avoid spending too much time on some tricky parts of the lab.
- As this is a slightly longer lab, make sure you aim to finish much before the deadline.

## mosh: My Own SHell

## Part 1

You will write `mosh.c` that implements a simple command shell for Linux. You are provided sample code `make_tokens.c` that takes a string input from the user and tokenizes it; you may reuse this as part of your shell. Use any creative prompt message that you want. Below are the the commands you need to implement in the shell, and the expected behavior of the shell for that command.

**Note**: You must use the `fork` and `exec` system calls to implement the shell. The idea is for the main program to act as a parent process that accepts and parses commands and then instantiates child processes to execute the desired commands.

You must **not** use the `system` function provided by the C-library, which simply calls an existing Linux shell. Also, you must execute Linux system programs wherever possible, instead of re-implementing functionality. Example: Given command `echo "You know nothing Jon Snow."`, you should use the echo binary executable available rather than implementing `echo`.

The following functionalities should be supported:

- `cd directory-name` must cause the shell process to change its working directory. This command should take one and only one argument; an incorrect number of arguments (e.g., commands such as `cd`, `cd dir1 dir2` etc.) should print an error in the shell. `cd` should also return an error if the change directory command cannot be executed (e.g., because the directory does not exist). For this, and all commands below, incorrect number of arguments or incorrect command format should print an error in the shell. After such errors are printed by the shell, the shell should not crash. It must simply move on and prompt the user for the next command.
- All simple standalone built-in commands of Linux e.g., (`ls`, `cat`, `echo`, `sleep`) should be executed, as they would be in a regular shell. All such commands should execute in the foreground, and the shell should wait for the command to finish before prompting the user for the next command. Any errors returned during the execution of these commands must be displayed in the shell.
- Any simple Linux command followed by the output redirector `>` (e.g., `echo hi > hi.txt`) should cause the output of the command to be redirected to the specified output file. The command should execute in the foreground. An incorrect command format must print an error and prompt for the next command.

[1] Nope! not the tortoise-style protective cover.

- Any list of simple Linux commands separated by "`;;`" (e.g., `sleep 100 ;; echo hi ;; ls -l`) should all be executed in the **foreground** and **sequentially** one after the other. The shell should start the execution of the first command, and proceed to the next one only after the previous command completes (successfully or unsuccessfully). The shell should prompt the user for the next input only after all the commands have finished execution. An error in one of the commands should simply cause the shell to move on to the next command in the sequence. An incorrect command format must print an error and prompt for the next command.

## Part 2

- Any list of simple Linux commands separated by "`&&`" (e.g., `sleep 100 && echo hi && ls -l`) should all be executed in the **background** in **parallel**. The shell should start the execution of all the commands in parallel by creating child processes for each, and come back to prompt the user for the next command without waiting for all the commands to finish. For each command that completes in the background at a future time, the shell should print a message about the background process being completed. An incorrect command format must print an error and prompt for the next command.
- `exit` command should cause the shell to terminate, along with all child processes it has spawned.
- For any other command that doesn't match what is specified above, the shell should print an error and move on to prompt the user for the next command.
- The handling of the SIGINT (`Ctrl+C`) signal by the shell should be as follows. While the shell is running a command in the foreground, typing `Ctrl+C` should cause the foreground child process to terminate, but not the shell itself. When the user is executing a set of commands in sequence using `;;`, `Ctrl+C` should terminate the execution of the currently executing command, and all future commands in that sequence, and must cause the shell to return to the user prompt. Hitting `Ctrl+C` should **not terminate any background processes** started by using `&&`.
- Typing the `exit` command should cause all background child processes to receive the SIGINT signal and terminate; the shell itself must terminate after cleaning up all state and dynamic memory.

## Important Guidelines:

- When a process completes its execution, all of the memory and resources associated with it are deallocated so they can be used by other processes. This cleanup has to be done by the parent of the process and is called `reaping the child process`.The shell must also carefully reap all its children that have terminated. For commands that must run in the foreground, the shell must wait for and reap its terminated foreground child processes before it prompts the user for the next input.
- For commands that create background child processes, the shell must reap any terminated background processes, while running other commands. This can be handled by overriding the `SIGCHLD` signal.When the shell reaps a terminated background process at a future time, it must print a message to let the user know that a background process has finished.
- By carefully reaping all children (foreground and background), the shell must ensure that it does not leave behind any zombies or orphans when it exits.
- You must implement all the commands above in your shell. Test your shell using several test cases, and record them all in your report.

## Tips

- You are given a sample code `make-tokens.c` that takes a string of input, and "tokenizes" it (i.e., separates it into space-separated commands). You may find it useful to split the user's input string into individual commands.
- 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.

- You may want to build a simple shell that runs simple Linux built-in commands first, before you go on to tackle the more complicated cases.
- You will find the `dup` system call and its variants useful in implementing I/O redirection and pipes. When you `dup` a file descriptor, be careful to close all unused, extra copies of the file descriptor. Also recall that child processes will inherit copies of the parent file descriptors, so be careful and close extra copies of inherited file descriptors also. Otherwise, your I/O redirection and pipe implementations will not work correctly.
- You must catch the SIGINT signal in your shell and handle it correctly, so that your shell does not termiate on a `Ctrl+C`, but only on receiving the exit command.
- When you send a SIGINT signal to a process, all child processes will automatically receive the signal. However, this will not work for us, because we want to send SIGINT selectively only to the foreground process (recall that the background processes must terminate when the shell finally exits, not on a `Ctrl+C`). One way to overcome this problem is to place your child processes in a separate process group (check out the `setpgid` system call), so that they do not receive the SIGINT sent to the parent automatically. The parent shell process must then manually send the SIGINT (using the kill system call) to those child processes that it wants to terminate.
- You will find the `chdir` system call useful to implement the `cd` command.
- You may assume that no more than 64 background processes or 64 foreground processes will exist at any time.
- Carefully handle all error cases listed above for each command. For example, an incorrect command string should always print an error.

## Submission Guidelines

- All submissions via moodle. Name your submission as: **<rollno_lab3>.tar.gz**
- The tar should contain the following files in the following directory structure:

```
<roll_number_lab3>/
|__code/
|____mosh.c
|____...
|__report/
|____report.pdf
|__testcases/
|____testcases.pdf
|____outputs of sample runs (samplecases + testcases)
|____readme with ordered list of commands and output file names
```

- Your code (`mosh.c`) should be well commented, indented and easily readable.
- The `report.pdf` should explain briefly how you built the shell
- The README contains some testcases (some with their expected outputs and some not). You should try all the testcases using your own shell. The `testcases.pdf` should show results for sample runs of your code using appropriate screenshots on these testcases as well as your own testcases.
- We will evaluate your submission by reading through your code, executing it over several test cases, and by reading your report.
- **Deadline:**
  **Part 1 : Friday 4th August 2017 05:00 PM (In-lab)**
  **Part 2 : Sunday, 6th August 2017, 08:00 AM**