Assignment #1 - Set up your Linux environment and Basic Operating system Access

This is an individual assignment. However, you are encouraged to help (and seek help from) your peers (except sharing code, of course). This assignment contains the following two parts: 1) observing the OS through the /proc file system; 2) building a shell. Everything you do in this warm-up assignment is at the user-level (outside of the OS kernel). Your code must be compiled for Linux, and our reference machines will be the ones in the CSIL lab(Make sure your code compiles and runs correctly on these machines).

Part II: Building a shell

UNIX shells:
The OS command interpreter is the program that people interact with in order to launch and control programs. On UNIX systems, the command interpreter is often called shell: a user-level program that gives people a command-line interface to launching, suspending, and killing other programs. sh, ksh, csh, tcsh, bash, ... are all examples of UNIX shells. You use a shell like this every time you log into a Linux machine at a CS computer lab and bring up a terminal. It might be useful to look at the manual pages of these shells, for example, type "man csh".

The most rudimentary shell is structured as the following loop:

Print out a prompt;
Read a line from the user;
Parse the line into the program name and an array of parameters;
Use the fork() system call to spawn a new child process;
The child process then uses the exec() system call (or one of its variants) to launch the specified program;
The parent process (the shell) uses the wait() system call (or one of its variants) to wait for the child to terminate;
Once the child (the launched program) finishes, the shell repeats the loop by jumping to 1.
Although most commands people type on the shell prompt are the names of other UNIX programs (such as ps or cat), shells also recognize some special commands (called internal commands) that are not program names. For example, the exit command terminates the shell, and the cd command changes the current working directory. Shells directly make system calls to execute these commands, instead of forking a child process to handle them.

Requirements in detail:

Your job is to implement a very primitive shell that knows how to launch new programs in the foreground and the background. It should also recognize a few internal commands. More specifically, it should support the following features.

It should recognize the internal commands: exit, jobs, and cd. exit should use the exit() system call to terminate the shell. cd uses the chdir() system call to change to a new directory.
If the command line does not indicate any internal commands, it should be in the following form:
<program name> <arg1> <arg2> .... <argN> [&]

Your shell should invoke the program, passing it the list of arguments in the command line. The shell must wait until the started program completes unless the user runs it in the background (with &).

To allow users to pass arguments you need to parse the input line into words separated by whitespace (spaces and '\t' tab characters). You might try to use strtok_r() for parsing (check the manual page of strtok_r() and Google it for examples of using it). In case you wonder, strtok_r() is a user-level utility, not a system call. This means this function is fulfilled without the help of the operating system kernel. To make the parsing easy for you, you can assume the '&' token (when used) is separated from the last argument with one or more spaces or '\t' tab characters.

The shell runs programs using two core system calls: fork() and execvp(). Read the manual pages to see how to use them. In short, fork() creates an exact copy of the currently running process, and is used by the shell to spawn a new process. The execvp() call is used to overload the currently running program with a new program, which is how the shell turns a forked process into the program it wants to run. In addition, the shell must wait until the previously started program completes unless the user runs it in the background (with &). This is done with the wait() system call or one of its variants (such as waitpid()). All these system calls can fail due to unforeseen reasons (see their manual pages for details). You should check their return status and report errors if they occur.

No input the user gives should cause the shell to exit (except when the user types exit or Ctrl+D). This means your shell should handle errors gracefully, no matter where they occur. Even if an error occurs in the middle of a long pipeline, it should be reported accurately and your shell should recover gracefully. In addition, your shell should not generate leaking open file descriptors. Hint: you can monitor the current open file descriptors of the shell process through the /proc file system.

Your shell needs to support pipes. Pipes allow the stdins and stdouts of a list of programs to be concatenated in a chain. More specifically, the first program's stdout is directed to the stdin of the second program; the second program's stdout is directed to the stdin of the third program; and so on so forth. Multiple piped programs in a command line are separated with the token "|'. A command line will therefore have the following form:

<program1> <arglist1> | <program2> <arglist2> | ... | <programN> <arglistN> [&]

Try an example like this: pick a text file with more than 10 lines (assume it is called textfile) and then type

cat textfile | gzip -c | gunzip -c | tail -n 10

in a regular shell or in the working shell we provide. Pause a bit to think what it really does. Note that multiple processes need to be launched for piped commands and all of them should be waited on in a foreground execution. The pipe() and dup2() system calls will be useful.

Your compiled executable **must be called my_shell.**

Administrative policies

A note on the programming language:
C/C++ is the only choice for this assignment and all later programming assignments. We are not alone in this. Most existing operating system kernels (Linux and other UNIX variants) themselves are written in C; the remaining parts are written in assembly language. Higher-level languages (Java, Perl, ...), while possible, are less desirable because C allows more flexible and direct control of system resources.