mySHell.c

# Notes and Style

* must be written in C
* Include a Makefile. Include a “make clean” command like Lab1, that will remove all files resulting from compilation (.o, executables, etc).
* Note: make will catch your SIGSTP signals, so test your exe just with the  
  executable. (e.g., ./myshell).
* Your program **must** compile with –Wall, with no warnings showing.
* You **must** include a “listing.txt” file: a single .txt file with all your source files  
  concatenated into it. (e.g., cat \*.h \*.c > listing.txt)
* Use common-sense commenting. Provide function headers, comment regions, etc. No particular format or style is required. Overall good style is required

# myShell – create a Linux shell to interact with the Kernel

You will create a new shell (like bash) that uses system calls to enable the user to  
interact with the kernel to start programs, pause / continue them, catch signals, and to pipe data between programs.

**Synopsis:** Your shell will:

* Display a prompt that shows the current working folder (getcwd) followed by a % sign. It will take text commands from the user. Empty input does nothing. Use strntok\_r to tokenize your strings. strtok is not acceptable (why? Read the man page).
* Implement the following internal commands

➢ cd <dir> – change directory to dir (chdir)

➢ exit – quit the program gracefully and return error code 0

* Enable the user to execute system commands

➢ Use fork, and in the child use execvp to execute the command (this version checks the system path variable, so you don’t have to hunt down executables)

➢ Make sure all parameters (up to some hard-coded maximum, e.g., 100) are passed to the command

➢ Make the parent wait (block) until the child is done.

* Enable the user to pipe commands together. E.g., the following should work:

➢ ls /usr/bin | grep a | more – lists all the files in the /usr/bin directory, pipes that output into grep, which filters by only showing lines that have a, and then pipe that output into more.

➢ Support an arbitrary number of commands (up to some hard-coded maximum, e.g., 100).

➢ Use anonymous pipes, and dup2, to re-write a process’ stdin and stdout entry to instead send data to and take from your pipe. Hint: the file table entries for standard in and out have fixed, standardized numbers, available in system constant somewhere. See if you can find it.

➢ Close unused pipe ends in the child / parent (e.g., the child may use the write end of a pipe, so the parent should close that after forking).

➢ Make sure that the parent blocks until all children are done!

* Implement simple job control.

➢ Catch the SIGTSTP signal (generated by ctrl+z in most terminals). If there currently is no program running, tell the user that there is no job to suspend. Remember to re-subscribe to the signal after you catch it!

➢ If you had a program running, it would have caught the signal and paused. Tell the user that the job is suspended. Your shell should be back alive now.

 Internal commands should still work. If the user tries an external command, tell them they can only have one job at a time (it’s not hard to enable multiple jobs, but it’s busy work, and you learn little..)

 Implement internal commands fg (foreground) brings the job back to life and pauses your shell, by sending SIGCONT to all children processes (can be multiple, e.g., with pipes), and waiting again on the children. bg (background) brings the job back to life, by sending SIGCONT, but your shell does not pause now. You will notice interleaved output on screen

 For testing, run the sample count program, which slowly counts to

30. Try pausing and resuming (fg and bg) to see it work properly.

you do not have to worry about arguments have spaces, quotes, or escape characters. For example, you do NOT have to account for:

cd “./~/’My Desktop’/\t exe”|less

Spaces may be assumed to separate each command or argument or special character

When waiting on the children, use the flag WUNTRACED, which tells the wait to also stop blocking when the child pauses (SIGTSTP), not only when it finishes. Also, check the status given to you from the wait to see why the wait finished. Write the wait code once, in a function, and use it when starting a new job, and when bringing a suspended job into the foreground.

errno is a common error mechanism used by system call functions. Use perror to turn this into text.

Piping between processes is tricky. Remember that one pipe will go between two children. In the example above (ls /usr/bin | grep a | more), ls has its stdin untouched, but stdout is re-directed to pipe1’s “in” end. pipe1’s “out” end is remapped to grep’s stdin, such that output from ls goes into grep’s stdin. Simiarly, pipe2 takes grep’s stdout (to it’s write end), and gives it to more (read end linked Piping between processes is tricky. Remember that one pipe will go between two children. In the example above (ls /usr/bin | grep a | more), ls has its stdin untouched, but stdout is re-directed to pipe1’s “in” end. pipe1’s “out” end is remapped to grep’s stdin, such that output from ls goes into grep’s stdin. Similarly, pipe2 takes grep’s stdout (to it’s write end), and gives it to more (read end linked

In addition to using the system calls to change a process’ working directory, you should also normally change the environment variables. This is tedious so it’s not necessary for the assignment, but some programs may fail to work properly if they rely on this. E.g., you may have some bugs with specific programs in pipe chains. But if it works in general with most UNIX basic tools, it should be fine for marks, e.g., ls, grep, more, less, etc.