COMP 3413 Fall 2021Assignment 2

Due Date: Oct 15, 11:59 PM

Notes and Style

- must be written in C
- The prof has provided a working demo on d2l (samplePrograms zip, myshell). try running on FCS linux machines/vm image
- your assignment code must be handed in electronically using the D2L drop box. Try early and re-submit later.
- Include a Makefile. If "make" doesn't compile and produce an executable, your assignment **will not be marked.** Include a "make clean" command like Lab1, that wil 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 run. Programs that do not run, or immediately segfault before working at all, **will lose 50%**. Save early and often, and use version control. Submit often, only submit working versions.
- Your program **must** compile with –Wall, with no warnings showing. We will check your makefile. You will lose marks for warnings, and for not including -Wall.
- Include a "listing" file: a single .txt file with all your source files concatenated into it. Good header text in EACH file is imperative here. (e.g., cat *.h *.c > listing.txt)
- Confusion over D2L at the last minute is your own fault.
- Use common-sense commenting. Provide function headers, comment regions, etc. No particular format or style is required. Overall good style is required (detailed below – no magic numbers, good use of functions, etc).
- Error checking is extremely important in real world OS work, and you must do it rigorously. Error handing, however, is hard in C. Try to handle gracefully when you can, but for hard errors (out of memory, etc.), hard fail (exit) is OK.
- Your assignment will be tested on FCS linux machines/images. TAs will not fix your compilation bugs.

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.

For this assignment, 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 (e.g., pipe).

Strategy and Hints: For many of you, you will struggle with getting back into C.

1) Get back into your work environment. Spending time to get used to working in a terminal will help throughout the class and a CS career (e.g., ssh into FCS linux machines), using VI, Emacs, etc., and not in notepad++, etc. This will save you time vs uploading from some IDE each time you make a change.

2) Focus on the basic C stuff first (Makefile, multi file compilation, etc.), including your basic string parsing, before doing the OS stuff (fork, signals, etc.). **The above requirements list is ordered from easiest to hardest.**

3) Use the C resources given to you. The GNU manual (https://www.gnu.org/software/libc/manual/html_node/index.html#Top) is a good place. http://c-faq.com/ is great for those tricky, dusty corners.

4) Refresh your GDB, and **program very defensively**. Program **SLOWLY**. Test **FREQUENTLY**. C is nasty. Check all errors, check pointers, give error messages. Name all constants at the top of the file or in a .h file. This will save you major debug time. This assignment does not require much malloc / freeing, and **can use arrays with pre-set reasonable sizes**. This will save errors.

5) Use the sample executable to test operation, to ensure you understand expected functionality.

6) Use the man pages, on the target machine (FCS machines), extensively. The web is good for general understanding, but implementations vary wildly. Only trust the man pages on your target machine.

7) It's easy to get lost in the nested operations (tokenize input for pipes, check each command, tokenize for parameters, etc.). Write helper functions to "unroll" these nested loops. E.g., I have a function that tokenizes a string into a pointer array for me, handling all the strtok mess, and then I just use the result:

```
int tokenizeIntoArray(char *buf, char** bufArr, const int bufArrSize, const char* delim).
```

8) 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

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

10) 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

to more's stdin). Think about this in the general case, and you can setup a for loop going through as many pipes as you need. **I highly recommend reading the full man page on pipe.** (including the examples in it)

11) 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.

12) This is hard. Start early.

Marking:

Assignments that do not have a Makefile, do not compile, or do not run (e.g., no action, or segfault before running) will lose 50%. Makefiles will not be marked (except to check you compile with -Wall), just get it working.

Marks in brackets, with resolution to half mark. To get full marks in a category your solution must be "amazing". Good solutions that have small quirks or could be improved will lose 1-2 marks, and on the opposite ends, solutions that have something kind of working may gain up to half marks.

Be strategic: ironing out every single bug takes a lot of time, and may save you 1 mark on that section, compared to moving to the next section and implementing most of that functionality.

[5] C quality – defensive programming, error checking, no magic numbers. Your listing file is used for this. No listing file, you get 0.

[5] Basic functionality – prompt generation, input loop, proper error messages.

[2] Internal commands CD and EXIT

[3] Enable the user to execute external commands with parameters

[5] Enable arbitrary number of pipes

[5] Simple job control

More general solutions get better marks (unless static values are explicitly allowed, like the argument number in this assignment being allowed to be 100). If we allow static values, they must still be using good style (constants, etc).