CSE422: Operating Systems Organization Fall 2016

Project 1

Assigned: Feb. 11th Due: Mar. 8th

Teams

This project is a complex project. As such, you are allowed (expected?) to work in teams of four to complete this project.

Overview

This assignment will help you to understand the functions of a command language interpreter and to learn how to create processes, and share files and file descriptors in a UNIX environment. You are to write and test a C/C++ program that implements a simple shell whose executable is called xsh. The features of the xsh shell are described below. It is a toy shell since we will assume that the program only operates correctly when given correct input; the behavior of xsh when given incorrect input is undefined (although it will be a bonus if you can get the shell to work correctly for incorrect input). It is an incomplete shell since it is missing <u>many</u> features included in other shells. Some of the missing features are briefly described at the end of this document. It is a pedagogic shell since it produces output that exposes the workings of the shell and contains many of the fundamental features in common shells. The command line looks like:

```
xsh [-x] [-d <level>] [-f file [arg] ... ]
```

where the square brackets ([ ]) indicate an optional word(s), and ”...” indicates 0 or more words. xsh normally reads commands from stdin. The options are:

- -x: The command line after variable substitution is displayed before the command is evaluated.
- -d <DebugLevel>: Output debug messages. DebugLevel=0 means don’t output any debug messages. DebugLevel=1 outputs enough information so that correct operation can be verified. Larger integers can be used as the user sees fit to indicate more detailed debug messages.
- -f file [Arg] ...: Input is from a file instead of stdin; i.e., File is a shell script. If there are arguments, the strings are assigned to the shell variables $1, $2, etc. The location of File follows the rules described in Search Path below.

All output from your shell should be sent to the terminal display. Input to your shell will come either from the terminal keyboard or from the shell script file. If xsh is executed interactively (i.e., there is no shell script file provided as input), then:

> Your shell should prompt for input from the user using the standard input, and display any output generated to standard out on the terminal display. Every command line prompt displayed should be of the form “xsh >>” to avoid confusion with the real shell. Every line of input will represent a command that the shell is to execute. This command will consist of a sequence of arguments. Each argument will be separated by one or more whitespace characters. The first argument will be the name of an internal command, or the name of a program to be executed. All other

arguments on the line (unless a I/O redirection character is present) will be passed to the program using the UNIX conventions (i.e., you should use command line argument processing in your shell when spawning a new process to execute the command). An error message should be returned to the display if the program/internal command specified for execution does not exist or is inaccessible.

Batch Mode

If your shell is executed in batch mode (i.e., using the `-f file` option), then your shell should open the file `file`, read all the commands in that file and execute them one after the other. When the end-of-file is reached, your shell should exit. You may assume that the batchfile `file` contains commands that are in the same format as for commands entered by the keyboard in interactive mode. All commands will appear on a single line in the file and are terminated by a <crlf> (i.e., a new line character).

External Versus Internal Commands

An internal or built-in command is one that the shell executes itself instead of running another program (via fork/exec). All commands that are not listed as internal commands below are treated as if they were external or non-built-in commands.

Search Path

The shell searches for external commands and shell scripts using these rules:

- If the command name is an absolute or relative pathname P, the file P should be an executable binary (e.g., /usr/bin/zip) or executable script (/usr/bin/spell).
- Otherwise, search the directories listed in the PATH environment variable.

Internal commands

In the description below, W stands for a single word. A whitespace character (SPACE or TAB) or a newline character terminates a word. ”...” indicates a continuation of 0 or more repetitions of the preceding symbol. For example, ”W ...” means one or more W’s. I stands for a particular type of word that is an integer. In cases where the ordinal position of a word is important, we will use W1, W2, ... to mean Word1, Word2. Also, a word can be a variable and should be substituted as noted in the “Variable Substitution” section below.

- **show *W1 W2 …*** : display the word(s) followed by a newline
- **set *W1 W2***: set the value of the local variable W1 to the value W2
- **unset *W1***: un set a previously set local variable W1
- **export *W1 W2***: export the global variable W1 with the value W2 to the environment
- **unexport *W***: unexport the global variable W from the environment
- **environ**: this command should display to the terminal a listing of all environment strings that are currently defined
- **chdir *W***: change the current directory

- **exit I**: exit the shell and return status I. If no value for I is given, the shell should exit with a status code of 0. You should also ensure that any currently running processes are killed off before the shell exits.
- **wait I**: the shell waits for process I (a pid) to complete. If I is -1, the shell waits for any children to complete. Note that you can't wait on grandchildren.
- **clr**: clear the screen and display a new command line prompt at the top
- **dir**: list the contents of the current directory
- **echo <comment>**: display <comment> on the stdout followed by a new line
- **help**: display the user manual using the *more* filter
- **pause**: pause the operation of the shell until the 'Enter' key is pressed
- **history n**: this command should list to the display the last n executed commands (i.e., those that you entered at the command line prompt, even if the command fails or can't be found). If your shell executed less than n commands, then your shell should just display all of these commands. if n is not specified, then you should display all of the previous commands (you may assume that you only have to keep track of the last 100 commands issued). Each line of the output of history should contain a line number followed by two spaces followed by the input to the command line prompt. In other words, it should look something like the following:

  ```
  1  ls -alg
  2  date
  3  who
  4  cd /usr/local
     ⋮
  ```

  In this list, line 1 is the oldest command in the history list, and the last line n is the most recent command in the list. You should do a man history to get more information about how the UNIX tcsh and bash versions of history work.
- **repeat n**: prints to the screen, and then executes the command in the history list which corresponds to the nth line. If n is not specified, then repeat should execute the previous command (i.e., the one which currently has the highest history list number).
- **kill [-n] pid**: this command should send a signal to the specified process. For example, if you do the following sequence of commands:

  ```
  xsh >> ps –aef

  UID      PID  PPID  C    STIME TTY       TIME CMD
  root       0     0  0   Jan 11 ?         0:01 sched
  root       1     0  0   Jan 11 ?         4:49 /etc/init -
  root       2     0  0   Jan 11 ?         0:03 pageout
  root       3     0  0   Jan 11 ?       142:17 fsflush
  root     418   415  0   Jan 11 ?         5:07 usr/lib/saf/listen tcp
  root     244     1  0   Jan 11 ?         0:28 /usr/sbin/cron
  root     415     1  0   Jan 11 ?         0:00 /usr/lib/saf/sac -t 300
  root     137     1  0   Jan 11 ?         0:00 /usr/sbin/keyserv
  ```

```
root     161     1  0   Jan 11 ?        0:26 usr/lib/netsvc/yp/ypbind
root     169     1  0   Jan 11 ?        0:00 usr/lib/netsvc/yp/ypxfrd
root     176     1  0   Jan 11 ?        0:00 /usr/sbin/kerbd
lmiller  329     1  0   Jan 11 ?        0:01 /usr/home/lmiller/shell/xsh
lmiller  372   329  0   Jan 11 ?        0:04 /usr/home/lmiller/myprog/myprog
```

xsh>> kill –9 372

then the SIGKILL signal should be sent to process number 372. The format of this command is "kill –<signal #> <PID>", where signal # is the numeric id associated with the signal to be sent, and PID is the process ID number. If the signal # is not specified, then the kill command should send the SIGTERM signal by default. See the man pages for signal in order to learn about the various signal numbers (i.e., issue man –s 5 signal).

Shell Environment

Environment Shell Variable: The shell environment should contain:

shell=<pathname>/xsh

where shell=<pathname>/xsh is the full path for the executable (not a hardwired path back to your directory, but the one from which it was executed).

Environment Path Variable: Your shell should use the environment PATH variable, if defined, to direct the search for any external programs to be executed. You should search the whitespace separated list of paths in the order they appear in the PATH variable, and execute the first matching command that is found. The use of a path consisting of a single period, (that is a ".") means to look in the current directory. The use of a path consisting of a double period (that is a "..") means to look in the parent directory. You may assume that all paths are absolute paths, except for the period and double period just described.

Environment Parent Variable: For any programs/processes that are forked to run external programs, the environment variable "parent" should be defined as: parent = <pathname>/xsh similar to that described above.

External Commands

An *internal* (or *built-in*) command is one that the shell executes itself instead of running another program (using fork/exec). All commands not listed in the Internal Commands section should be consider *external* (or *non-built-in*) commands. External commands require that your shell find the file(s) corresponding to the command(s) specified, and then forking and execing the program(s) as its own child process(es). The program(s) should be executed with an environment that contains the entry: parent=<pathname>/xsh .

Background Commands

A command that should be run in the background is ended with an ampersand "&" as in "myprog &" which means execute "myprog" in the background. Background commands return immediately and execute in the background. The xsh should return control immediately to the user so they may execute additional commands. This means that your shell should continue execution without waiting for the specified programs to complete. When you execute a command in the background, you should display to the terminal the process ID numbers for any processes associated with that command.

Local vs. Global Variables

Local variables are available only to the current shell and not any sub shells or processes. These are variables that are configured using the “set/unset” internal commands. Global variables are available to the current shell and any child processes including sub shells. Global variables are configured using “export/unexport” internal commands.

Variable Substitution

The dollar sign character "$" as the first character of a word "$XY" signifies that "XY" is a variable. Single-level variable substitution is always done before command evaluation. No recursive substitution is allowed (e.g. the "$Y" in "$X$Y" is not replaced since XY is a single unit of text, aka “word”).

There are three special shell variables:

$$: PID of this shell

$?: Decimal value returned by the last foreground command

$!: PID of the last background command

Stdin/Stdout Redirection

Stdin and/or stdout can be redirected to/from a file using the following syntax: "myprog arg1 arg2 < F1 > F2" which means execute program myprog with command line arguments arg1 and arg2, and redirect stdin from F1 and redirect stdout to F2 for program "myprog"

Pipes

If pipes are present in the input line (i.e., the symbol “|” appears on the input line), then the input line will consist of multiple commands with their associated arguments. The “|” symbol is the command separator in this case, and all characters between two “|” symbols (but not including the “|” symbols) should be passed as arguments to the programs specified in each command (which is either the first argument on the input line, or the first argument following the “|” symbol). For this project, you will only have to worry about up to three pipe symbols in a command line (you might see something like: `ls –alg | wc | cat | more` but you won’t see `ls –alg | grep xyz | wc | cat | more`).

Terminal-Generated Signals

"CTRL-C" should terminate the foreground process but not xsh. “CTRL-S” should stop the foreground process but not xsh. “CTRL-Q” should continue (i.e., resume) the foreground process. “CTRL-Z” should stop the set of processes running in the foreground (if any). Your shell will receive signals related to these CTRL characters whenever you type one of them on the keyboard. This means you will need to handle the following signals:

1. SIGINT: you should handle this signal, first to make sure that the interrupt (i.e., <ctrl-c>) can't cause your shell to be terminated, and second to cause the foreground process(es) to be sent a SIGINT signal. Thus, if your shell receives a SIGINT, it should "forward" this signal to all the processes associated with the current foreground command (if any).
2. SIGQUIT: you should handle this signal, first to make sure that it can't terminate your shell, and second to cause the foreground process(es) to be sent a SIGQUIT signal. Thus, if your shell receives a SIGQUIT, it should "forward" this signal to all the processes associated with the current foreground command (if any).
3. SIGCONT: you should handle this signal to cause the foreground process(es) to be sent a SIGCONT signal. Thus, if your shell receives a SIGCONT, it should "forward" this signal to all the processes associated with the current foreground command (if any).
4. SIGTSTP: you should handle this signal, first to make sure that the stop (i.e., <ctrl-\>) can't cause your shell to be stopped, and second to cause the foreground process(es) to be sent a SIGTSTP signal. Thus, if your shell receives a SIGTSTP, it should "forward" this signal to all the processes associated with the current foreground command (if any).

Other Signals

Your shell should ignore the following signals:

1. SIGABRT: you should simply ignore this signal.
2. SIGALRM: you should simply ignore this signal. Just to make sure no one can terminate your shell with an alarm signal.
3. SIGHUP: you should simply ignore this signal. Just to make sure that no one can terminate your shell with a hang up signal.
4. SIGTERM: you should simply ignore this signal. Just to make sure that no one can terminate your shell with a terminate signal.
5. SIGUSR1: this signal is generated by other processes. You should simply ignore this signal.
6. SIGUSR2: this signal is generated by other processes. You should simply ignore this signal.

Remember, you should be able to issue the internal command "exit" in order to cause your shell to terminate. In addition, you can always send your shell a SIGKILL signal in order to cause it to terminate.

Other Features

1. Each word (token) is separated by white space.
2. Multiple spaces/tabs are reduced to a single space during the substitution and line scanning phase.
3. The command line prompt is the character sequence 'xsh >> ' (i.e., x, s, h, space, >, >, space).
4. The # character signifies the beginning of a comment. All other characters following and including the # is ignored during interpretation.
5. Blank lines are ignored.

Assumptions

You may assume all of the following for this simple shell:

- invalid input is undefined – as a general rule though, after you have parsed the first argument on the command line, you should check to see if it matches an internal command, and if it doesn't you should try to find a file with that name which can be executed, and if you fail to find it, or it is not executable, then you should report some kind of error.
- no built-in control structures (e.g. if, while, etc) will be used on the command line, or in any shell scripts executed
- no interactive features like auto-complete
- no filename expansion like "~" or "*"
- no command substitution
- no job control commands like "fg" or "bg" and no job control management (see man pages for csh, tcsh, bash, or ksh for more information on job control functions of a shell)

Man pages

In addition to fork(2), waitpid(2), execvp(2), execve(2), execle(2), sh(1), csh(1), exit(3), and fgets(3), you may want to look at getenv(3), putenv(3), chdir(2), dup2(2), open(2), read(2), write(2), close(2), getpid(2), getppid(2), setpgid(2), sigaction(2), sigprocmask(2), sigsetjmp(3) and siglongjmp(3), and strtok(3). You should also look at the signals.h file.

Implementation Notes

You are NOT allowed to use the ***system*** (e.g. "man system") system call / library function to implement commands. The ***system*** system call / library function refers to the function defined as "int system(const char *command);". But as a strategy, there is nothing wrong with using the ***system*** system call / library function as a temporary implementation means while debugging and developing the code.

Warning

When using the `fork()` system call, be extra careful not to repeatedly spawn off processes without either doing an `exec()` system call, or doing some simple work and terminating. This will prevent you from spawning off processes until you have used up your limit of processes on the machine. This will make your life, as well as the lives of the system administrators much easier.

If you do get into the situation where you have forked of the maximum number of processes allowed, you will have to log into the machine remotely, send each process a SIGSTOP signal, and then after **all** processes are stopped, send each of the processes a SIGKILL signal to terminate them.

Project Requirements

1. Design a simple command line shell that satisfies the above criteria and implement it on the specified LINUX platform. Your project must be able to execute under Linux. All projects will be tested under the Linux environment.

2. Write a simple manual describing how to use the shell. The manual should contain enough detail for a beginner to UNIX to use it. For example, you should explain the concepts of I/O redirection, the program environment, and background program execution. The manual MUST be named readme and must be a simple text document capable of being read by a standard Text Editor.

For an example of the sort of depth and type of description required, you should have a look at the online manuals for csh and tcsh (man csh, man tcsh). These shells obviously have much more functionality than yours and thus, your manuals don't have to be quite so large.

You should NOT include building instructions, included file lists or source code - we can find that out from the other files you submit. This should be an Operator's manual not a Developer's manual.

The user manual can be presented in a format of your choice (within the limitations of being displayable by a simple Text Editor). Again, the manual should contain enough detail for a beginner to UNIX to use the shell. For example, you should explain the concepts of I/O redirection, the program environment and background execution. The manual MUST be named readme (all lowercase, please, NO .txt extension).

3. The source code MUST be extensively commented and appropriately structured to allow your peers to understand and easily maintain the code. Properly commented and laid out code is much easier to interpret, and it is in your interests to ensure that the person marking your project is able to understand your coding without having to perform mental gymnastics!

4. The submission should contain only source code file(s), include file(s), a makefile (all lower case please), and the readme file (all lowercase, please). No executable program (binary files) or object code files should be included. The person marking your project will be automatically rebuilding your shell program from the source code provided. If the submitted code does not compile it cannot be marked!

5. The makefile (all lowercase, please) MUST generate the binary file `xsh` (all lower case please). A sample makefile would be:

```
# Joe Citizen, s1234567 - Operating Systems Project 1
# CompLab1/01 tutor: Fred Bloggs
xsh: xsh.c utility.c xsh.h
	gcc -Wall xsh.c utility.c -o xsh
```

   The program xsh is then generated by just typing make at the command line prompt.

   Note: The fourth line in the above makefile **MUST** begin with a tab.

Implementation Suggestions

This is a very complex project. You should consider attacking it in stages. I would recommend that you take skeleton code (see lecture notes), and add all of the internal commands into it first, without implementing them yet (except exit should be implemented first). I would then implement commands and features in roughly the following order:

1. the **exit** command.
2. the **clr** command.
3. the **echo** command.

4. the **show** command.
5. the environment list and the commands **environ**, **export**, **unexport**, **set**, and **unset**.
6. the notion of the current directory, and the internal commands **dir** and **chdir**.
7. signal dispositions for all signals.
8. the history list and the **history** command
9. the **repeat** command
10. the **kill** command
11. forking to spawn processes.
12. implement the use of waitpid to handle process status changes
13. I/O redirection to redirect input and output to/from a file.
14. commands with a single pipe
15. commands with multiple pipes
16. use of the background character ! on the command line
17. the **pause** command
18. the **wait** command
19. the **help** command, which displays the man page to the screen
20. implement the **`–f file`** option on the command line along with support for reading commands from input shell script file and executing them (note: this is batch execution mode and as such your shell should not read any commands from the terminal – although output from those commands should be displayed on the terminal display)

Notes

- You can assume that all command line arguments (including the redirection symbols, <, >, & >> and the background execution symbol, !) will be delimited from other command line arguments by white space - one or more spaces and/or tabs.
- Programming style matters. I will reject projects if they demonstrate poor programming style.
- You are required to use a makefile
- You are expected to use copious comments throughout your code.
- You are expected to provide a man page for your xsh shell. This man page should provide the required information for invoking/executing your shell, and for interacting with your shell. This man page can be in the same form as the UNIX/Linux man pages for csh/tcsh/ksh/bash, etc., except that you only need to worry about the features your shell actually provides (the man pages for csh/tcsh/ksh/bash include many features you are not required to implement).
- Turn in details will be supplied on the course web page at a later date.