CSCI 352 - UNIX Software Development

Fall 2014, Assignments 5 and 6

Assignment 5 Due: Tuesday, Nov 18, 2014. 250 points. (The earlier you finish assignment 5, the more time you have on assignment 6. I'll open the grading script as soon as someone needs it.)
Assignment 6 Due: Friday, Dec 5, 2014. 325 points.

For assignment 5, you will continue your work on the mini-shell. Start with the code base you have for assignment 4. (ASSIGNMENT-4 tagged sources.)

This assignment's work consists of the following steps: (Complete each step before going on to the next step. I recommend you check in your sources after completing each step and before continuing on the next step. Be aware of the requirements for both assignments so if you make a change to your existing code base, so it would be consistent with what you need in further work.)

- Correct any problems you know about from assignment 4.
- (10 points) Add comments to your shell. All characters on an input line starting at and after a pound sign (#) are comments. Of course, a pound sign inside double quotes does not denote the start of a comment. Also, $# is not the start of a comment. It must continue to be processed as a standard variable. This should be done *first* in processing a line. It may be done in a function that originally reads the input line from the file or standard input.
- (15 points) The prompt should be the value of the environment variable P1. If no such variable exists, then use the string '% ' as the prompt. (Note, that is the percent sign followed by one space.) Also, make sure your prompt is printed to standard error and is not printed when reading commands from a script file.
- (20 points) Add processing for the tilde character (~) to expand(). It should be done in the following way. If it is the first character after white space, replace it with the home directory name of an appropriate user. ~ by itself is the home directory of the user running the shell. Examples include ~ and ~/path. ~name should be replaced by the home directory path of the named user. Examples include ~name and ~name/path. You must use the password information to do these expansions. *You may not use environment variables.* Also, notice that the home directory of some users is not "/home/loginname". Try these in your login shell.
- (140 points) Add command output expansion to expand(). In expanding a command line, if you find a $(, find the matching) and everything between the $(and the matching) is considered a shell command and executed. Notice, count every (in your

matching, not just the ones preceded by $. The characters `$(cmd ...)` are replaced with the standard output of the command "cmd ..." after you replace each newline (`\n`) character in the command output with a space. If the final character in the command output is a newline character, it should not be replaced with a space, it should be deleted. Do not do expansion on the output of the command. Notice, all standard processing for the command inside the $() must be done, including all expansion and argument processing. The easiest way to process the $() command is to find the terminating) and replace it with the null character ('`\0`'), call processline() on the command text and after the call, put back the) into the command string. (This implies that the expansion function will be indirectly called recursively.) You must allow for at least 200000 characters in the final command expansion. (This may be a compile constant.) (Warning: Be careful when allocating a 200000 character local variable in functions that may be called recursively. If you use more than one of these big arrays in processline and expand, you may have your program crash on you when using recursion. On Linux, you have 8 Megabytes for your stack size so that gives you the maximum of about 40 of these 200000 character arrays.)

As an example, consider the following:

```
% pwd
/home/phil
% envset PWD "$(pwd)"
% echo ${PWD}
/home/phil
% echo The current working directory is $(pwd).
The current working directory is /home/phil.
% envset VAR 1
% envset VAR $(expr ${VAR} + 1)
% echo ${VAR}
2
```

Note, $() inside double quotes still run the command in the $(). Double quotes inside the $() are for the command inside the $().

Also note, some shells also recognize this expansion with the syntax `‘cmd ...‘`. You are to implement only the $() version. Also, notice that the $(...) version allows for "embedded commands". For example,

```
% aecho $(ls -ld $(dirname $(pwd)))
```

Finally, you must use pipes to implement $() and you must use only one pipe for each $() and have at most two processes running at any time in processing $() and this includes nested $($()) forms. Builtin commands in $() must be run by the shell without forking and you do not need to worry about deadlock with builtin commands for this assignment. $() commands also sets the value to use for $? expansion. And if you use popen(3) to implement $(), you will lose 150 points of the possible 250 points.

- (20 points) Add signal processing so that your shell is not terminated by SIGINT. Warning! Some system calls are interrupted by a signal and return with errno set to EINTR. You may need to deal with these depending on how you set up things. Also, your signal handler should *not* just siglongjmp(3) back to main. The best way to do this is to have your signal handler send SIGINT to any process on which your shell is wait(2)ing and set a global variable that says a sigint happened. This method works better with assignment 6 where statements are implemented and SIGINT should stop the processing of statements.

- (20 points) Add a new builtin command "read name". It reads a line of input from standard input and defines the environment variable "name" with input line, minus the final newline character, as the value of the variable.

- (25 points) Points for assignment 5 turn in, style and so forth. Turn in should be similar to assignment 4.

This is the end of the features for assignment 5. **Read what you need to do for assignment 6 and the specification for the order of operations.** This should help you design your data structures and functions for assignment 5. When you are done, tag your sources with the tag "ASSIGNMENT-5" and then continue work. Try to finish assignment 5 as early as possible. The earlier you finish it the more time you will have for assignment 6. Again, for assignment 5 turn your printed source code and a printed sample test session showing these features.

To complete assignment 6, add the following features.

- (115 points) Input and output redirection of the forms:

  > path

  2 > path

  >> path

  2 >> path

  < path

  where there may or may not be a space between the redirection operator and the path name. Redirection operators inside double quotes are parts of parameters and not a redirection operator. Redirections inside of $() apply to the command inside the $().

Note that redirection of stdout inside $() would produce no output and thus would be an empty replacement, but is a legal construct. Note, all of the following forms are legal, but this not a complete list.

```
% command <file
% command<file
% command < file
% command > file1 < file2
% command < file1 > file2
% < file1 command > file2
% > file1 < file2 command
```

Remember, > redirection sends all of standard output to the named file. All previous contents of the file are deleted. >> appends standard output to the end of the named file. 2> will redirect standard error to the named file. 2>> appends standard error to the end of the named file. Also, the '2' in stderr redirection must follow a space or be the first character on the line and must must be followed by the >. In all output redirection, if the file doesn't exist, it needs to be created. < attaches an existing file to the standard input.

Final note: You must not use fopen(3) to open your redirection files.

- (115 points) Add pipelines to your shell. For example:

  ```
  % finger | tail -n +2 | cut -c1-8 | uniq | sort
  ```

  should print out a sorted list of user names of all logged in users. The return value for the shell variable $? is the exit value of the last command in the pipeline. Redirections in a pipeline are for the command without a `|` between the command and the redirection. Pipes are set up first and then redirections are done. Pipes do not pipe standard error, it only connects the standard output of the command on the left of the `|` to the standard input of the command on the right.

  Note: Make sure your pipelines do not generate zombies. You must properly "wait()" (or wait3, wait4 or waitpid) on all children of a pipeline.

Note: The order of processing for your shell shall be:

1. Comment removal
2. statement identification (see below)
3. Expansions: Variable, $(), tilde, ... (left to right)

4. Pipeline identification
5. Redirection processing (each element of the pipeline)
6. Argument processing (each element of the pipeline)
7. Command execution (each element of the pipeline)

This will do something different than standard UNIX shells do. For example:

```
% envset X "> JUNK"
% echo Hello ${X}
```

should create a file called JUNK that contains the word Hello. Most standard shells would just echo "Hello > JUNK" and not create the file.

And for those truly committed to getting the most possible points, do the following for the final 75 points of assignment 6.

- (50 points) Add an "if statement built-in". The syntax as follows:

  ```
  if <command>
    <commands>
  else
    <commands>
  end
  ```

  where things in <> describe what goes there. The characters < and > do not appear in the command line unless they are used as the redirection characters. The "else" and the following commands are optional. If the command on the "if" line exits with an exit value of zero, execute the "then" commands. If it exits with an exit value of non-zero, execute the "else" commands, if present. For example:

  ```
  if grep -q myword myfile
    echo Yes
  else
    echo No
  end
  ```

  should print Yes if "myword" is found in "myfile" and print "No" if it is not found. Also, note redirections may not come before the "if" key word. The key words "else" or "end" must be on a line by themselves after comment removal.

- For interactive executions, you need to prompt for the lines that need to be read until you find the matching "end". (The only difference between interactive and non-interactive shells is that interactive shells print prompts.) You need to read all those lines first and then when all the lines are read, then you execute the lines. If the environment variable "P2" is set, use that string for the prompt. If it is not set, then use the default prompt of '> '. (Greater-than space)

- (25 points) Add a "while" statement. The syntax as follows:

  ```
  while <command>
    <commands>
  end
  ```

  While the initial command reports success, do the commands over and over again. Note, you must retain the original text of the commands so variable expansion and other processing can be done again and again as the while statement continues. This is especially important that the while command is processed (variable expansion ...) each time the while line is executed. For example, consider the following statements:

  ```
  envset X 1
  while test ${X} -lt 10
    echo ${X}
    envset X $(expr ${X} + 1)
  end
  ```

  This should print 1 to 9. Without processing the while command every time, this would be an infinite loop. Again, redirections may not come before the "while" key word.

  Finally, note that a SIGINT delivered to your shell should stop the while loop (or an if statement) but not terminate the shell!

When you have completed as much of the above as you can or will do, then check in your final versions and tag them ASSIGNMENT-6. Again, please do not add other features except these. You may create files to help you organize your code. I suggest that you put all prototypes in a "proto.h" file so they are in only one place. You can then include that file in any file that needs access. Again, turn in your printed sources and printed scripts of your own tests of your minishell.

Notes:

- 20 points of assignment 6 are for the turn in, style and so forth.
- Again, I will grade your assignments by running a script that checks out your sources, runs make and then runs a series of scripts.
- As usual, the student versions of the grading scripts will be available in the directories /home/phil/public/cs352/testa5 and /home/phil/public/cs352/testa6. If they are not readable two days before the assignment is due, send me e-mail to remind me to make them readable.
- Please turn in your sources on paper for both assignments. Don't forget scripts with your own tests of your shell and a run of the grading script. You may lose points if you don't show enough tests of your devising.
- Notice where common tasks are done and make them functions!
- Use recursion where it helps you!
- Recursion for pipelines can be difficult, but it can work.
- Start early, work often. Don't wait to start these assignments.
- Pipelines and redirections can appear in the if or while command!
- Pipelines and redirections can appear in the $() commands!
- If you have a question of what does a particular command do, try it using my shell. It is found at /home/phil/.bin/msh on the lab machines. It implements assignment-6.