CSCI 352 - UNIX Software Development

Fall 2014 – Assignment 3

200 points

Due: Tuesday, October 21, 2014

For this assignment, you will continue work on the mini-shell. Start with the code you turned in for assignments 1 and 2.

This assignment's work consists of the following steps:

- Read the CVS book. If you have not purchased the book, you can download a pdf version of the book. See http://cvsbook.red-bean.com. Read sections on: *An Overview of CVS* (Ch 2), *CVS Repository Administration* (Ch 3). Notice especially the section on *Starting A Repository.* Reading chapter 1 would be good also!

- Start your own CVS repository. (Read chapter 3 of your CVS book FIRST.) It should be the directory 'cs352f14/CVSrep' in your home directory on the lab Ubuntu machines. (Full path would be like /home/phil/cs352f14/CVSrep, where 'phil' is replaced with your user name.) (Hint: read about the init command to cvs.)

- Make sure that "others" has access to your repository and any files in the repository. Make sure that the group student has no privileges to your repository. Again, use chmod(1) to make sure that **ALL** your directories have no group privileges and that others have r-x access to your home directory and rwx access to all directories in your CVS repository. **This is important.** Also, you need give others rw- access to the files "history" and "val-tags" in the CVSROOT directory under your CVSrep directory.

- Read the CVS book on how to import sources. Start with assignment 1 which should be the file 'msh.c'. Import it to a "module" called msh. On your import your vendor tag must be your user name and the release tag must be the string "ASSIGNMENT-1". *Note:* this must be your Assignment 1 program, not after modifications for Assignment 2 or Assignment 3. (The best way to do this is to cd to your cs352f14/a1 directory and import from there.) You have now created new directories in your CVSrep tree and need to make sure you have the correct permissions on the new directory. Please note that the release tag is important. It *must* be the string specified above.

- Your CVSrep should be ready to use. If you happen to mess up your import, don't worry. If you have to, remove your entire CVSrep tree and start again. (The work to this point is worth 25 points.)

- Learn how to check out a copy of your msh module. *You should never work from inside your CVSrep directory. Only the program cvs should do any changes to the files in the CVSrep tree other than permission changes.* An easy way to do this is to create a new directory called

"work352", cd to it and then check out msh. Note, you should also learn how to check out your msh across the network. This can make it easier to work at home. Cygwin, OS X and all free UNIXes come with a cvs client. The book explains two remote access methods, pserver and ext. You want to use the ext method since you have ssh access to the lab machines.

- Read the CVS book about adding a new file.

- (20 points) It is now time to add your assignment 2 files to your repository. Since, in a project, you would add new files rather than import them, I want you to add the files of assignment 2 that did not exist in assignment 1 to your repository. For the file msh.c, you just copy the assignment 2 version over the assignment 1 version. Read the documentation on how to add them to the repository. You need to start in a checked out working copy of your msh project and *copy* the files from your cs352f14/a2 directory into your working msh directory. Add the new files using cvs. *Note: You should only add source code to your repository. Do not add any .o file or any executable files.*

- Read the CVS book about committing (check-in). Notice, adding files doesn't actually put them into the repository, you must commit them to get them in your repository. Commit the new files you added in the previous step. You should now have 4 files, msh.c, arg_parse.c, builtin.c, and proto.h. After you have committed them, make sure you can check out a new copy of your msh module.

- (5 points) Once you are sure you have assignment 2 in your repository, you need to tag the the files with the tag "ASSIGNMENT-2". Read about tagging and how to tag the files. Note that you can delete tags or move tags if you make a mistake. Also, you should NOT make a branch. Finally, the tag must be *EXACTLY* "ASSIGNMENT-2". Character case and special characters are important.

- Now that you have both assignment 1 and 2 in your repository, you will be doing active development on the files in your repository. You will work in a directory with a checked out copy of your repository. As you edit your file or add new files you have changes that need to be committed. So, at various times during your development, commit your current revision. Especially do this if you are at a point where things are working to that current point and you are getting ready to start to add a new feature or change a lot of code. Also, for this class, a good rule of thumb is to check-in your work daily. This may not work as well in an environment where multiple people are working on the same source tree and you should not check-in non-working code. But for your own work, the checking in your code after each day of work is good. *I will have very little sympathy for someone who loses 3 days work just because they didn't check-in their work.* Finally, make it a priority to check-in your code BEFORE any major restructuring so you can go back the previous revision if you need to. You should also learn how to check out a previous revision of your code.

- (15 points) Add a new file, "Makefile", that makes the executable "msh" from your sources. Just giving the command "make" in your directory should make the program. You should also have a "clean" target that removes all object files and the final executable. (You may need to read about the "make" program to get this working correctly.) You must have your Makefile written so that adding a file that needs to be compiled *requires exactly one line to be changed* (the change must be just adding a file name once) and possibly adding a dependencies line for the new file. You *must* utilize the built-in suffix rules for compiling C programs. At the end of this step, you should be able to make a working minishell. This is a good point to check-in your work.

- (10 points) Add a RCS id line to the beginning of every file. For the C files, the line initially looks like:

  ```
  /*    $Id: $    */
  ```

  For the Makefile, it should look like:

  ```
  #    $Id: $
  ```

  Capitalization **is** important. Using "ID" or "id" will not work correctly. These are REQUIRED since the RCS id will be used to check that your repository revision and the revision you printed and turned in are the same revision. Now, commit all your files at this point. All your files should be at revision 1.2 except msh.c and the Makefile. msh.c should be at revision 1.3. The Makefile should be at revision 1.2 or later.

- **All the work up to this point must be done before doing anything past this point.**

- Now, correct any problems you know about in your assignment 2 and commit the changes. **Do not change any files in your assignment 2 turn in directory. Fix you problems in your working directory for assignment 3.**

- (70 points) Add environment variable processing. Do this by:

  - In the function processline(), you will add a call to a new function called "expand". The prototype should be: "int expand (char *orig, char *new, int newsize);". (new should be a pointer to a fixed size array similar to buffer in main(). newsize is the number of characters in the new array. *The parameter* **newsize** *is very important. This is an "input parameter". It tells your expand() function the maximum numbers of characters that may safely be put into the parameter* **new**. *Your code must make sure that expand() does not overflow the* **new** *array by using the* **newsize** *parameter.* The return int should be a value that means "expand was successful" or "expand had an error and processline should stop processing this line".) It would be best to put this function in a new file. (Don't forget

your RCS id line at the top of your new file and add the proper entry in your Makefile.) "expand()" should not change the original line. It processes the original line as explained below and produces a new line. This call to expand must be called before arg_parse() because the new line is what must be passed to arg_parse. Notice, as a result of this, the original line passed to processline() is not modified. This is different than assignments 1 and 2 where the line was modified. Also, you should just declare an array in processline() for the new line. Do not malloc() space for this new line.

- expand() is to process the original line exactly once, starting with the first character (orig[0]) and looping to the last character. It should do this loop exactly once and do all the required processing during this one loop. When expand finds text that needs replacing, it replaces the text with the required new text and continues processing the original text at the point after the replaced text. Do not expand the new text. The basic processing done by expand() is to copy each character, one at a time, to the new string unless it detects that the current character is part of text that needs to be replaced. In that case, expand() adds the new text (expansion text) to the new string as specified by the replacement. Be careful to not write code that is order n squared. Also, remember that strlen(3) is an order n operation.
- In expand(), replace ${NAME} in the original string with the value of the environment variable of the same name. (Yes, the braces are part of the syntax. $NAME is not processed as an environment variable.) This value of the environment variable is placed in the new string. If the name does not exist in the environment, then it is replaced with the null string. This is done for all ${..}s found in the original string. In ALL cases the ${NAME} will not appear in the new string. If you find a ${ and do not find a closing }, print an error message and stop processing the line. Read about the library function getenv(3).

  Note: Replacing one string in the original line with another string in the new line is a task that will be done several more times during this project. It would be best to write helper functions to make it easy in expand() to add similar functionality in a slightly different context.
- Add a built-in command with the syntax:

  envset NAME value

  that sets the environment variable of the same name to the given value.
- Add a built-in command with the syntax:

  envunset NAME

  that removes the variable NAME from the current environment.

For implementing these builtin functions, read about the library functions setenv(3) and unsetenv(3).

- (10 ponts) Prepare for other special variable processing in expand(). Write your code to easily add other kinds of expansion. To show that it is successful, add the following expansion.

  $$ is replaced by the base 10 ASCII equivalent of the shell's pid.

- (20 points) Add the following built-in command.

  cd [dir] - set the working directory to dir. Give an appropriate error message if you can't connect to that directory. If the directory dir is not given, use the value of the environment variable "HOME". If neither the "HOME" variable is set or the dir is given, generate an error message. (hint: man 2 chdir)

- When you have completed with all of your modifications and have them all working and *checked in*, tag your msh project with the tag "ASSIGNMENT-3". Again, it must be *EXACTLY* that tag. Character case is important. You will lose points if my scripts can't check out your sources. **Warning: Before you tag your assignment you should see if it passes the grading script.** Do this tag only after you are satisfied that your program passes as much of the grading script as you are going get it to pass.

Some final notes:

- The final 25 points are available in style, assignment submission contents and quality, error checking, following my coding standards and so forth.

- Your shell may need to print error messages from time to time. These should always be printed to the standard error file. Use perror(3) only to report errors returned in errno. User input errors like a missing } should *not* use perror(3) to report the error.

- Converting strings to integers is easily done by atoi(3). It can also be done by using sscanf(3) or strtol(3). Converting integers to strings is easily done by using snprintf(3).

- Your assignment will be graded by checking out your sources, running make and then testing your shell. This is done by a script and it expects your shell to be named *msh*.

- **What to turn in?** Print your sources and make sure that your printed sources has the "$ Id: $" line filled in by CVS. *They must have the revision number that was tagged.* If they are missing or are not filled in by CVS or are not the same revision number as you tagged, you will lose points. **It is best** to *carefully* check in your final revision, tag your final revision and then print your final revision. If you are doing this 10 minutes before class starts, expect to forget something that will cost you points. Again, I want output that is at least as nice as the output of a2ps. Just printed text files will lose points. Also turn in a printed *sample run* where *you* tested your shell with your own tests and a run of grading script I provide to you. You will get graded on the quality of your tests you turn in. You should be able to do the following commands by hand and have them work. "username" should be replaced by your login name.

```
mkdir newdir
cd newdir
cvs -d /home/username/cs352f14/CVSrep get -rASSIGNMENT-3 msh
cd msh
make
./msh
make clean
exit
```

- Again, you should recognize you can use your repository from other machines on the network. For example, you could be on your home computer and use your repository on the Lab Ubuntu machines. (Yes, you could even do it under Windows ... but you should use the cygwin environment for your UNIX work.) Use ssh as your "transport agent". (Set CVS_RSH environment variable to ssh.) If you have several copies of your program checked out, like one here at WWU and one on your home computer, you will need to learn how to use the cvs command "update."

- I will make my test script available in the directory /home/phil/public/cs352/testa3 on Ubuntu. (It should be usable by at least the Thursday before the assignment is due.) As before, to use use the script, cd to the /home/phil/public/cs352/testa3 directory and give the command *./try* when there. (**You do not need to tag your assignment to run the try script. In fact, it best to not tag your assignment until after it is complete and ready to turn in.** It will work with the tag there, but if your assignment is not tagged it will check out HEAD. If you have it tagged, it will always check out your tagged version, even if you change a file and commit. If you tag your work and then delete your tag, the try script won't find any files.) It will create a directory /home/username/testa3 where it will test your work. It is possible that not all tests done by my real grading script are done by the public one. The public one does most of them. If your program passes all my tests, the script says

  ```
  Big script output same
  ```