CSCI 352 - UNIX Software Development

Spring 2014 – Assignment 4

200 Points

Due: Friday, May 9, 2014

For this assignment, we will continue work on the mini-shell. Start with the code base you have for the last assignment. (ASSIGNMENT-3 tagged sources.)

This assignment's work consists of the following steps:

- Correct any problems you know about from Assignment 3.
- (100 Points) Add argument processing for arguments to msh's main. This is not argument processing for arguments produced by arg_parse, it is for the arguments passed to your shell when your shell starts. (The parameters to msh's "int main (int mainargc, char **mainargv) ...". The name mainargc and mainargv could be any name, I just named them that to show you that they are the argc and argv that are parameters to main() for your shell.) This would be the arguments used when your shell is run. For example, your msh could be run like:

  ```
  msh filename arg1 arg2 arg3 arg4 arg5
  ```

  instead of

  ```
  msh
  ```

  To add argument processing for main's arguments, do the following:

  - If the shell is started with one command line argument (the shell's name), all input comes from the standard input. This is considered an interactive execution. (This is what your msh currently does. You start your msh by giving the command "msh" or "./msh".)
  - If the shell is started with more than one command line argument, shell commands are read from the file named in the first argument after the shell's name. (The first argument is the script file name. For example, you would start msh running with the command "msh script arg1 arg2" and msh would read commands from the file named "script".) If msh can not open the file for reading, msh should print an error message and exit with the value of 127.

Note: Again, these are not command lines processed by msh, this is how msh is started. You will need to use mainargc and mainargv as passed into the main function of your msh program.

- If, while reading a script file, the shell encounters an end of file, exit with the value of 0.
- Do not print a prompt when reading from a script file. An interactive run still must print a prompt to standard error.
- Lines from either standard input or from the file are processed exactly the same way. They are sent to "processline()" for processing. (In fact, the only change in the main loop should be which file is read by fgets(). The best way would be to have only *one* fgets(). Remember, variables can vary and you can have stream variables. (Type FILE *.))
- Add processing to "expand()" that recognizes the pattern $n, where n is an integer greater than or equal to 0. "expand()" should replace the $n by the the shell's command line argument n+1. (Yes, n may be a multi-digit integer.) When run with multiple command line arguments, $0 should be replaced by the script name. If n is larger than the number of command line arguments, replace the $n with the empty string. (The empty string really means that the $n pattern in the command is just removed from the command.) When the shell is run without any arguments, that is during an interactive execution, $0 should be replaced with the name of the shell. All other $n values are replaced by the empty string.
- "expand()" should also replace $# by the base 10 ascii representation of the number of arguments. That is, the number of arguments indexed from $0 to $n-1 where n is the value of $#. For example, if your msh was run using the command:

  ```
  msh script_name a b c d
  ```

  $0 should be replaced by "script_name", $1 by "a" and so forth. Also $# should be replaced by 5. (Note: the argc parameter to msh's main() function would be 6.)
- Implement a new built-in command with the syntax:

  shift [n]

  that shifts all arguments starting with $1 by n values. That is, parameter i+n now becomes parameter i and for i = 1 to $#-n.

Also, $# is also decremented by n. If n is not given, n is 1. Note, $0 is still the script name and $# should never be less than 1. If a shift command requests a shift that can not be done, that is n $\geq$ $#, give an error message and do not shift the arguments. Note, arguments $1 through $n are no longer available to use in $i replacement.

– Implement a new built-in command with the syntax:

  unshift [n]

  that does the opposite of shift. If n is not specified, unshift should completely undo any effects of all earlier shift commands. If n is specified, unshift the number of arguments specified by n. If n is greater than the number of shifted arguments, return an error message and do not shift the arguments.

  As an example, assume you have a file called "showshift" that contains the following commands:

```
echo showshift is named $0
echo Number of arguments is $#.
echo Argument 1 is $1.
echo Argument 2 is $2.
echo Argument 3 is $3.
echo Argument 4 is $4.
shift 3
echo Number of arguments is $#.
echo Argument 1 is $1.
echo Argument 2 is $2.
echo Argument 3 is $3.
echo Argument 4 is $4.
unshift 1
echo Number of arguments is $#.
echo Argument 1 is $1.
unshift
echo Number of arguments is $#.
echo Argument 1 is $1.
```

  Now assume you give the command "msh showshift a b c d e f". The output should be:

```
showshift is named showshift
Number of arguments is 7.
Argument 1 is a.
```

```
Argument 2 is b.
Argument 3 is c.
Argument 4 is d.
Number of arguments is 4.
Argument 1 is d.
Argument 2 is e.
Argument 3 is f.
Argument 4 is .
Number of arguments is 5.
Argument 1 is c.
Number of arguments is 7.
Argument 1 is a.
```

Note: Good implementations of shift and unshift will not copy data in the mainargv variable or change the mainargv variable.

Note: While the above script has the $n variables near the end of the line, they may appear anywhere in the line and there may more than one replacement needed.

- (50 points) Add the following other special expansion processing to "expand()".
  - Expand $? by replacing it with the base 10 ascii equivalent of the exit value of the last command. If the command did not exit normally, use 127 for the exit value. (This should already done by the "exit(127)" after the execvp().) Note, if the last command was a successful built-in command, $? is replaced by 0. If the last command was an unsuccessful built-in command, $? is replaced by 1.
  - Add a limited wildcard expansion capability. The wildcard character for your shell is '*'. In the simple version, the '*' appears by itself. White space (or the beginning or end of the line) must be on either side of the '*'. In this case, replace the '*' by names of all the files in the current working directory who's name do not start with a period. In the more complex version, if the '*' appears with a leading white space and trailing non-white space, that is something like " *xyz ", replace the '*' and the following characters with all file names in the current directory that end with the context characters. The context is terminated by white space, end of string or a double quote character. (A double quote

character is not part of the context.) Next, if the line does not match either of the two cases above, just copy the '* to the expanded string. Finally, the sequence `'\*'` should put only the '*' in the expanded string.

Any wildcard expansion should do not use file names that start with a period.) The file names placed into the expanded string must be separated by a single space character between file names. You should not have leading or trailing spaces. Note: If the trailing context characters include a slash (/), generate an error message and stop the processing of the current line. Also note: If no file names are matched, the pattern must be put in the expanded string. A final note: A double quote before a star should be treated as having a leading space. Finally, don't use a regular expression matching library to implement wildcard expansion. You are to use opendir(3), readdir(3) and closedir(3). Do not use scandir(3).

- (25 points) Add a new built-in command "sstat file [file ...]" that prints the information from stat(2) system call for each file on the command line. Your output for each file must be as follows:
  - the file name
  - the user name (not the uid)
  - the group name (not the gid)
  - the permission list including file type as printed by ls(1).
  - the number of links
  - the size in bytes
  - the modification time (use localtime(3) and asctime(3))

  These fields should be separated by a single space between the fields. Each file should be listed on a separate line. If there is no associated user name or group name, print the uid or gid in numerical form. (Hint: man strmode)

- The remaining 25 points are for style, turn in points, following directons and so forth.

- Add no other features.

- If you add a new file to your repository, don't forget the RCS id in your new file and to follow my style guide.

- Tag your files ASSIGNMENT-4.
- Again, I will grade your assignment by checking out your sources, running make and then testing your shell.
- Turn a printed copy of your sources and a printed copy of your own tests.

Notes:

I'd like to give you a few pointers. I believe these can reduce the time required to do this assignment.

- Do things as simply as possible.
- *Have only one main loop.* Figure out a way to get your line from the proper source instead of having a loop for script processing and another loop for interactive processing.
- Continue having processline() your primary function for running commands. processline() should not care whether you are doing an interactive run or are processing a script.
- Use global variables if information is required in many places. C tends to need global variables more than other languages. Don't make all variables global, but do use globals if it is the easiest way to get data where it is needed. For example, it might be useful to have global variables for access to mainargv and mainargc from the main function. main() can initialize the global variables from its parameters.
- The library function assert(3) can help you find bugs in your logic. Read about it in the man pages.
- Add one feature at a time and get it working. For example, processing of commands from a script file, then $n, $#, shift and unshift, then the new special variable $? and finally your limited wildcard processing. *When you get a feature working check-in your working sources.*
- man 3 isdigit
- You should be able to execute a similar set of commands as in assignment 3 except using the tag ASSIGNMENT-4 in the cvs checkout. Again, I'll make my test script available sometime before the assignment is due.

- If you want to run the testa3 scripts on assignment four, you can cd to the testa3 directory and run "try" with the environment variable N set to 4. For sh or bash, you can run the one line "N=4 ./try". For csh you need to say "setenv N 4 ; ./try".

- *What to turn in:* Turn in your source code and some tests of your own to show your shell is working. Please make sure you include the source of your msh testing scripts along with the output. Also, include a run of my test script with your output.