CSCI 352 - UNIX Software Development

Fall 2014 – Assignment 2

125 points

Due: Wednesday, October 8, 2014.

Start with the code you turned in for assignment 1. You should have a single file "msh.c" which has three primary functions, *arg_parse()*, *processline()* and *main()*.

This assignment's work consists of the following steps:

- In a *new work directory*, start with a copy of your msh.c file. *Do not work in your cs352f14/a1 turn in directory. This will make you lose points on Assignment 1.* For safety, chmod your msh.c in your a1 directory to have only read permissions. Then you can't accidentally change it.

- (25 points) Your first job is to move your arg_parse() function to a new file and a change of prototype to arg_parse(). This must be done before doing any other work on this assignment.

  - Create a new file called "arg_parse.c". Move all code for the function arg_parse() to this new file. Any helper functions should be moved and declared "static". Also, create a new file called "proto.h" that contains the proper prototype of arg_parse() so both your main program and arg_parse.c can #include the file to get the proper prototype definition. (Note: msh.c should #include "proto.h", not "arg_parse.c". Also, proto.h should *NOT* #include "arg_parse.c". Both "msh.c" and "arg_parse.c" need to #include "proto.h".) Note: you can still compile both files on one command line by giving the command:

    ```
    cc -g -o msh -Wall msh.c arg_parse.c
    ```

  - Change arg_parse() to have the prototype "int arg_parse(char *line, char ***argvp)" where arg_parse() returns the number of parameters in the argv, not including the NULL sentinel value. argvp is a pointer to the variable where the pointer to the malloced structure (the argv you are building in arg_parse) must be stored. You should not have to change very much in the implementation of arg_parse() unless you are fixing errors from Assignment 1. Also, you should not change any variable declarations in processline() or arg_parse() for the prototype change. *The prototype change does not change what the function does. In most cases, this prototype change requires changing at most three lines of code. If you change more than three lines of code or starting adding new variables or changing the types of variables you currently use, you are most likely doing it incorrectly.* It just changes how processline() calls arg_parse() and how arg_parse() returns the resulting pointer. This is the C equivalent to passing reference parameters ("out parameters" in Ada.), except C doesn't have reference parameters and so must pass a pointer to a variable in the calling function. You should not have any variables in your program declared of type "char ***". The *only* place for the type "char ***" is in the parameter list for arg_parse(). If you have "char ***" in any other place except the argument list and prototype you will lose points.

  - Make sure your argv is correctly built with only one malloc. It must have the correct size. Also, make sure that arg_parse() is called by the parent shell before your fork(2) and you properly free the argument array in the parent. Also, remember to correctly free your argument array if you return from processline() in places other than at the end of the function. For example, if fork(2) fails, you return. You should make sure you free the argument array before the return.

- (35 points) Change arg_parse() to also look for quotes. (Do this AFTER you have a working arg_parse() with the new prototype.) If a quote is found, look for a matching quote and everything between the two quotes is considered to be consecutive characters in the same argument. For example:

  ```
  prog "this is a single arg"
  ```

  has 2 arguments and

  ```
  prog this" "is" "a" "single" "arg
  ```

  also has exactly 2 arguments. And again,

  ```
  prog thi"s is a s"ingl"e a"rg
  ```

  has exactly 2 arguments and is identical in result to the above 2. Also, you can make a program name with spaces like:

  ```
  "prog name with spaces"
  ```

  Note: a quote may appear in the middle of an argument as you can see by the second example above. Also, the actual quote character is removed from the final version of the argument. If you find an odd number of quotes in the line, print an error message to "standard error" and stop processing the current line.

- (5 points) Change processline() so that if arg_parse() finds zero arguments, processline() just ignores that line. (Does nothing!)

- (45 points) Start a new file for built-in commands, "builtin.c", and implement builtins as follows:

  - Build a framework for executing built-in commands. These commands are done directly by the shell and are not forked to another process to do. Your processline() function should call arg_parse() and then call a routine to check to see if the command is a built-in

command. This function may execute the built-in if it is a built-in before returning to processline(). You may also have two functions, one to check if the command is a built-in and one to execute it. Add the prototype definitions of the one or two functions to proto.h. After processline() determines it is not a built-in command, that is the time to call fork(). If you execute a built-in command, no fork() is called. Remember, your "shell" process must free the argument list after it is done using it, regardless of whether it uses fork() or not. Note: in later assignments you will be required to add more built-in commands. A good "framework" will provide a very easy way to add new built-in commands. While a series of "if (strcmp(...,...)...) ... else if (strcmp(...,...)...) ..." will work, try to think of a way to to use strcmp only once (in a loop) to determine which built-in command needs to be executed. (Function pointers are really nice to use here with having each built-in command in its own function. If you have never worked with function pointers, look at the files named "funcptr.c" and "funcptr1.c" in the directory /home/phil/public/cs352.)

- All the identification and execution of built-in commands must be implemented in code in your builtin.c file. You should have at most two functions that processline() calls directly to implement your built-in commands. If you have two functions, one should answer the question "is the command described by this argv a built-in function" and the other one should do the built-in command. A single function would combine the functionality of the two into a single function. Add the prototype for your built-in commands function(s) to your "proto.h".

  Note: if you are calling strcmp() in processline, you are not implementing this as requested and will lose points. Also, if you put definitions in msh.c to help you with built-in commands, you are also not doing as requested. *All code implementing built-in commands must be in builtin.c. This includes global variables for built-in commands. These global variables must be static. No prototypes of "helper functions" in builtin.c should be in proto.h, just the routines that are called from processline().*

- Implement exactly the following two built-in commands. (The brackets ([...]) in the descriptions do not appear in the real command, they just show that that part is optional.)

  * exit [value] - exit the shell with the value. If value is not given, exit with value 0. This built-in command should call the exit(3) library call. Hint: man 3 atoi
  * aecho [-n] [arguments] - echo the command arguments with a single space **between** arguments. After all the arguments are echoed, a newline should be printed. (Not a space and a newline.) If the -n flag is given, the newline should not be printed. The word "aecho" and the "-n" flag should not be printed. "aecho" with no arguments just prints a newline.

    Hint: Plan for later modifications. In this assignment, you are to print the output to standard out (file descriptor 1). In a later assignment, you may need to send it out to a different file descriptor. Creating a string using snprintf(3) and writing that string using write(2) may be more difficult now, but will later save some problems on

future assignments and may save you time on the current assignment. Also, consider dprintf(3) as it may save you even more time.

- (15 points) The final 15 points are available in following the style guidelines and how you turn in your assignment. Here are some final notes for this assignment and the specification of how to turn in your assignment.
  - *None of your source files should #include a .c file and no executable code may appear in your .h file.* You can compile all three files on one command like:

    ```
    cc -g -Wall -o msh msh.c arg_parse.c builtin.c
    ```

    You can also compile using several commands like:

    ```
    cc -g -c -Wall msh.c
    cc -g -c -Wall arg_parse.c
    cc -g -c -Wall builtin.c
    cc -g -o msh msh.o arg_parse.o builtin.o
    ```

  - Again, make your source code available to me online like you did for assignment 1. Put all your source files in the directory "cs352f14/a2" relative to your home directory. Make sure the permissions are correctly set or you will lose points. I will grade your assignment by getting your sources and compiling them.
  - **What to turn in:** Also, you must turn in your sources on paper and turn in a *sample run* where *you* tested your shell. This test must not be running my grading script on your shell. You will get graded on the quality of the tests you turn in. Also, you must turn in a script of you running the grading script. Make sure it is easy to determine which are your tests and what is the grading script.
  - My reference executable is available for you to run. On the lab machines, you may run the program /home/phil/.bin/msh. If you have a question like "What should the shell do if I type .....", ask my shell instead of asking me. Of course, if you don't understand what it does, ask me.
  - Finally, I will make my test script available sometime on or after Monday, October 6 in the directory /home/phil/public/cs352/testa2 on lab machines. To use use the script, cd to the /home/phil/public/cs352/testa2 directory and give the command *./try* when there. It will create a directory /home/username/testa2 where it will test your work. (username is your user name.) 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

    ```
    Script output same
    Exit values correct
    ```