THE UNIVERSITY OF WESTERN AUSTRALIA

School of Computer Science and Software Engineering

CITS2002 Programming and Systems

CITS2002 Programming and Systems - Project 2 2015

See also: **Project 2 clarifications**

A *shell* is a command-interpreter used widely on Unix-based systems. Most shells receive character-based input, typically read from a keyboard or a textfile, parse their input into a token sequence forming the syntax recognised by the shell (command names, command arguments, and control-flow delimiters), and then attempt to execute the token sequence. The shell used by most CSSE students under OS-X and Linux is named *bash* (the Bourne Again Shell); a former CSSE graduate is the author of a very popular book describing the Bash shell.

The goal of this project is to implement a program named *mysh* supporting a *small subset* of the features of a standard Unix-based shell. Successful completion of the project will develop your understanding of some advanced features of the C99 programming language, and your understanding of the role of a number of system-calls responsible for process invocation and monitoring, and input and output management using file and pipe descriptors.

NOTE - this is a systems-specific project. You will find this project significantly easier to complete on OS-X or Linux. You will likely have great difficulty completing it on Windows, and it is recommended that you don't try. As for the 1st project, your submission will be tested and marked on CSSE OS-X laboratory machines.

Suggested steps

A number of C99 source code files are provided as a starting skeleton for the project. The most difficult part of the project, the command parser, has been written for you, and should be used without modification (and need not be understood). The parser returns a pointer to a tree of commands which the remainder of your shell will traverse and execute.

In summary, the function *parse_cmdtree(FILE *fp)* accepts a *FILE* pointer as its only input, and attempts to read and parse its input. If successful, function *parse_cmdtree()* returns a pointer to a user-defined datatype named *CMDTREE*, a self-referential structure holding all information necessary to execute the requested command-sequence. Your role is to execute the commands in the structure by (implementing and) calling your own function named *execute_cmdtree()*.

The project only requires implementation of a *small subset* of a traditional Unix shell. However, that subset will be quite faithful to standard shells. For this reason, this project description does not specify all details in great detail. If unsure about the role and action of shell feature listed in these steps, you should experiment with the same feature in your standard shell, *bash* (or obviously ask on *help2002*).

The approach taken to develop the *sample solution* has been broken into 10 independent steps, presented below. There's no requirement for you to follow these steps, but they are described here to provide an obvious pathway to a solution, and to explain the project's required features.

Step 0. Develop a *Makefile*

Develop a *Makefile*, employing variable definitions and automatic variables, to compile and link the project's files.

Step 1. Execute simple commands

1. execute commands without arguments, such as */bin/ls*
2. execute commands with arguments, such as */usr/bin/cal -y*

Helpful C99 functions and system-calls: *fork, execv, exit, wait.*
[added] **DO NOT** use the functions: *system,*, *popen*, or *execvp* anywhere in your project.

Step 2. Search path

The global variable *PATH* points to a character string which is interpreted as a colon separated list of directory names.

If the user enters a command name which does not contain a '/' character, then the members of the search path list are considered as directories from which to attempt to execute the required command. Once the required command is found and executed, the search terminates.

Helpful C99 functions and system-calls: *strchr.*

Step 3. Execute *internal* commands

1. If the name of the command to be executed is *exit*, then *mysh* should terminate itself, by calling the function *exit()*. When an additional argument is provided, it should be interpreted as the numeric exit-status. When requested without any arguments, the exit-status of the recently executed command should be used.

2. If the name of the command to be executed is *cd*, then *mysh* should attempt to change from the current working (default) directory to the directory specified as the first argument to the *cd* command. If the command *cd* is given without arguments, then the variable *HOME* should be used as the new directory.

   As with the role of *PATH* for command execution, the variable *CDPATH* points to a character string which is interpreted as a colon separated list of directory names. If the user enters a directory-name which does not contain a '/' character, then the members of *CDPATH* are considered as directories to locate the required directory.
3. If the name of the command to be executed is *time*, then *mysh* should time the execution of "remainder" of the command. The command's execution time should be reported in milliseconds (e.g. 84msec) to *mysh*'s *stderr* stream.

Helpful C99 functions and system-calls: *exit, chdir, gettimeofday.*

Step 4. Sequential execution

There are three forms of sequential execution, and thus three subparts to this step. The three forms differ in the action they take when a command *fails*. By convention, a command which fails will return a non-zero exit status.

1. A sequence of commands separated by the ";" token requires *mysh* to execute each command sequentially, waiting until each command has finished before beginning the next. In this form, *mysh* should continue to the next command regardless of the result of earlier commands. The sequence *ls ; date monday ; ps* would execute *ls*, then on its completion execute *date monday* (which will fail), then execute *ps* (even though the previous command failed). The exit value of sequential execution is the exit status of the command sequence to the right of the ";" token.
2. If the commands are separated by the token "&&" then *mysh* should continue with further commands only if the command sequence to the left of the token returns an exit value indicating success. For the command sequence *a && b*, *mysh* should wait for command *a* to terminate, test its return value and execute *b* if and only if *a*'s return value indicates success. The exit value of an "&&" command sequence is the exit status of the last command executed.
3. If the token "||" is used to separate commands then *mysh* should continue to the next command only if the first command sequence returns an exit value indicating failure. This is the converse of part *2*. For the command sequence *a || b*, *mysh* should wait for command *a* to terminate, test its return value and execute *b* if and only if *a*'s return value indicates failure. The exit value of a "||" command sequence is the exit status of the last command executed.

Step 5. Background execution

The token "&" causes the preceding command to be executed without *mysh* waiting for it to finish (asynchronously). Thus the sequence *ls ; ps & date* should start the command sequence *ls*, once this has completed start *ps* and immediately proceed to the command *date*. The exit value of background execution is success unless the fork call fails, in which case the exit value should indicate failure.

Step 6. Subshell execution

The sequence *(commands)* should cause *commands* to be executed in a subshell. A subshell is created by forking a copy of *mysh* to execute the commands. The exit value for a subshell is the exit value of the command sequence inside the parentheses.

Step 7. stdin and stdout file redirection

Commands may redirect their standard input and standard output from and to files, respectively, before the command is executed.

1. The sequence *command < infile* requests that the command use *infile* as its standard input.
2. The sequence *command > outfile* requests that the command use *outfile* as its standard output. If the file *outfile* does not exist then it is created. If it does exist it is truncated to zero then rewritten.
3. The sequence *command >> outfile* requests that the command appends its standard output to the file *outfile*. If *outfile* does not exist it is created.

NOTE: in all of the above cases *command* could be a subshell. It is valid to have both input and output redirection for the same command. Thus the sequence *(sort ; ps) < junk1 >> junk2* would take input from the file *junk1* and append output to the file *junk2*.

Helpful C99 functions and system-calls: *open, close, dup2.*

Step 8. Pipelines

The sequence *command1 | command2* requests that the *stdout* of *command1* be presented as *stdin* to *command2*. By default, the *stderr* output of *command1* is not redirected and appears at its default location (typically the terminal). With reference to this example, the *mysh* parser will not permit the *stdout* of *command1* to be redirected to a file (with >), nor the *stdin* of *command2* to be received from a file (with <). Note that the sequence *command1 | command2 | command3* requests that a pipeline of two different pipes be established.

Helpful C99 functions and system-calls: *pipe, dup2.*

Step 9. Shell scripts

Once an executable file name has been found (possibly using the search path) the standard function *execv()* can be called to try to execute it. If *execv()* fails, you should assume that the file is a *shell script* (a text file) containing more *mysh* commands. In this case you should execute another copy of *mysh* to read its input from the shell script. No specific filename extension is required.

Helpful C99 functions and system-calls: *access, fopen, fclose.*

Step 10. Setting *internal* variables

The final step is to allow the user to alter the internal variables *HOME*, *PATH*. and *CDPATH*.

To alter the *PATH* variable the user should type *set PATH list* where the list is a semi-colon separated list of directory names. For example *set PATH .:/bin:/usr/local/bin* The syntax for setting *HOME* is similar: *set HOME directoryname*

The amount of code to be written for this project is similar to that of the 1st project, although less needs to be designed "from scratch" because you're extending an (incomplete) code skeleton.

Starting files

Start your project by downloading, reading, and understanding the files in the archive **mysh.zip**.

- mysh.h - provides the definition of *mysh*'s user-defined datatypes and the declaration of global variables.
- mysh.c - provides the *main()* function, and calls the *parse_cmdtree()* function.
- globals.c - defines global variables, including *mysh*'s internal variables (Step 10), and the helpful *print_cmdtree0()* function.
- execute.c - where you define your *execute_cmdtree()* function.
- parser.c - defines the *parse_cmdtree()* and *free_cmdtree()* functions, which should be used without modification (and need not be understood).

You may modify any file or add any additional files to your project.

Program requirements

1. Your project, and its executable program, **must** be named *mysh*.
2. Your project **must** be developed using multiple C99 source files and **must** employ a *Makefile*, employing variable definitions and automatic variables, to compile and link the project's files.
3. If any error is detected during its execution, your project **must** use *fprintf(stderr, ...)* or *perror()* (as appropriate) to print an error message.
4. **It is anticipated** that a successful project will need to use (at least) the standard C99 and POSIX functions: *perror()*, *chdir()*, *exit()*, *gettimeofday()*, *fork()*, *execv()*, *dup2()*, *wait()*, *open()*, and *close()*.
5. Your project **must** employ sound programming practices, including the use of meaningful comments, well chosen identifier names, appropriate choice of basic data-structures and data-types, and appropriate choice of control-flow constructs.

Assessment

This project is worth **20% of your final mark** for CITS2002. It will be marked out of 40. The project may be completed **individually or in teams of two**. You are **strongly** encouraged to work with someone else - this will enable you to discuss your initial design, and to assist each other to develop and debug your joint solution.

During the marking, attention will obviously be given to the correctness of your solution. However, a correct and efficient solution should not be considered as the perfect, nor necessarily desirable, form of solution. Preference will be given to well presented, well documented solutions that use the appropriate features of the language to complete tasks in an easy to understand and easy to follow manner. That is, do not expect to receive full marks for your project simply because it works correctly. Remember, a computer program should not only convey a message to the computer, but also to other human programmers.

Up to half of the possible marks will come from the correctness of your solution. The remaining marks will come from your programming style, including your use of meaningful comments, well chosen identifier names, appropriate choice of basic data-structures and data-types, and appropriate choice of control-flow constructs.

Your project will be marked on the computers in CSSE Lab 2.01, using the OS-X environment. No allowance will be made for a program that *"works at home"* but not on CSSE Lab 2.01 computers, so be sure that your code compiles and executes correctly on these machines before you submit it.

Submission requirements

1. The deadline for the project is **12noon Friday 30th October (end of week 13)**.
2. Your submission will be compiled, run, and examined using the OS-X platform on computers in CSSE Lab 2.01. Your submission must work as expected on this platform. While you may develop your project on other computers, excuses such as *"it worked at home, just not in the lab!"* will not be accepted.
3. Your submission's C99 source files should each begin with the lines:

```
/*
   CITS2002 Project 2 2015
   Name(s):             student-name1 (, student-name2)
   Student number(s):   student-number-1 (, student-number-2)
   Date:        date-of-submission
```

```
*/
```

4. **You must submit your project electronically using *cssubmit*. No other method of submission is allowed.** You should submit all C source-code (*.c) and header (*.h) files and a *Makefile* that you wish to be assessed. You do not need to submit any additional testing scripts or files that you used while developing your project. If working as a team, only one team member should make the team's submission. The *cssubmit* facility will give you a receipt of your submission. You should print and retain this receipt in case of any dispute. Note also that the *cssubmit* facility does not archive submissions and will simply overwrite any previous submission with your latest submission.

5. You are expected to have read and understood the University's guidelines on academic conduct. In accordance with this policy, you may *discuss* with other students the general principles required to understand this project, but the work you submit must be the result of your own efforts. **All projects will be compared using software that detects significant similarities between source code files. Students suspected of plagiarism will be interviewed and will be required to demonstrate their full understanding of their project submission.**

Good luck!

Chris McDonald.