$Id: project3.html,v 1.12 2014/11/24 02:38:02 csci243 Exp $

Mechanics of Programming

Project: mysh: MY SHell

```
Revision History:
Fri Nov 21 17:37:04 EST 2014
-- first release
Sun Nov 23 21:36:46 EST 2014
-- clarifications
```

Due Date: Tuesday, 12/9/2014, 11:59:59 PM

Introduction

Goals and Objectives

The goal of this project is to build a simple interactive shell program and learn how to spawn sub-processes, implement some internal commands and handle errors. It builds upon skills of dynamic memory management, text I/O, string manipulation, and the design and use of structures.

Overview

The **mysh** shell is a simple interactive shell capable of executing simple *UNIX* commands and some internal commands specified later in this document.

You will implement a program that gets each line of input from the terminal console, formulates that input into a user command, and creates a *child* process to execute that command. The mysh shell continues to process each line entered until any one of these events occur:

- the user issues the command to terminate the shell;
- the user signals the shell that there is no more input;
- the shell encounters the end of input (e.g. end of file input); or
- the shell encounters a *fatal error* reading input(see later definition).

Examples

Below are several examples of mysh shell interactions with explanations of features they illustrate. In these examples, lines starting with the '#' character are comments that explain more about the example interaction.

The first example shows the user starting mysh and issuing commands following the mysh prompt, and the prompt contains a sequence number inside the *[]*.

```
$ mysh
mysh[1]> date
Fri Nov 21 13:33:53 EST 2014
mysh[2]> echo hello there universe
hello there universe
mysh[3]> cat header-c99.mak
CXXFLAGS =      -ggdb -Wall
CFLAGS =        -ggdb -Wall -std=c99
CLIBFLAGS =     -lm
CCLIBFLAGS =
mysh[4]> which grep
/bin/grep
mysh[5]>
$
# At prompt number 5 the user entered CONTROL-D.
# The program terminated and returned the user to the terminal shell.
```

This example shows what happens when a user enters an unknown command.

```
$ mysh
mysh[1]> bogus
bogus: No such file or directory
command status: 1
mysh[2]>
$
# The attempt to execute bogus produced an error code.
# The shell detected and printed the error.
# Since the execution status was non-zero from the child process,
# the shell printed the status it received.
```

This example shows starting the shell in diagnostic, verbose mode.

```
$ mysh -v
mysh[1]> ls MissingFile
        command: ls MissingFile

        input command tokens:
        0: ls
        1: MissingFile
        wait for pid 1244: ls
        execvp: ls
ls: cannot access MissingFile: No such file or directory
command status: 2
mysh[2]> ^D
$
```

The shell maintains a history of the last few commands it has processed. The next example shows how the command history lists works. Study the numbering used by the history command to implement the correct ordering and numbering.

```
# The user has already run 9 commands in this session.
mysh[10]> history
     1: date
     2: ls mysh.c
     3: banana
     4: cat header.mak
     5: ls
     6: gcc
     7: gprof
     8: valgrind
     9: gdb nothing
    10: history
mysh[11]> head Makefile
#
# Created by gmakemake (Ubuntu Jul 25 2014) on Fri Nov 21 12:33:09 2014
#

#
# Definitions
#

.SUFFIXES:
.SUFFIXES:      .a .o .c .C .cpp .s .S
mysh[12]> history
     3: banana
     4: cat header.mak
     5: ls
     6: gcc
     7: gprof
     8: valgrind
     9: gdb nothing
    10: history
    11: head Makefile
    12: history
mysh[13]>
# Notice how the history numbering changed as the commands were entered.
```

The shell supports a shortcut to execute a command out of the command history list; this is known as the *bang* command. The next example shows how the bang command syntax works. The bang command only works for the commands that exist in the history list.

```
mysh[12]> history
     3: banana
     4: cat header.mak
     5: ls
     6: gcc
     7: gprof
     8: valgrind
     9: gdb nothing
    10: history
    11: head Makefile
    12: history
mysh[13]> !11
head Makefile
#
# Created by gmakemake (Ubuntu Jul 25 2014) on Fri Nov 21 12:33:09 2014
#

#
# Definitions
#

.SUFFIXES:
.SUFFIXES:      .a .o .c .C .cpp .s .S

mysh[14]>
```

Note that, entry of `!5` next would re-execute the *ls* command, but entry of `!3` would fail because that command (i.e. 'banana') has fallen off the history list.

Requirements

Your solution must fulfill all these functional requirements to get full functionality credit. This outline of required behavior identifies all the major functionality of `mysh`.

Processing External Commands

1. The shell runs a loop waiting for user commands. The user may issue a command to the shell or a *Control-D* to tell the shell that there is no more input.
2. A user command may have a relative or absolute path. For example, `grep`, `/bin/grep`, or `../myProgram` should be accepted as commands, whether or not they refer to valid, executable programs on the system.
3. The prompt for each command contains its sequence number, 1, 2, and so on. The number of a previously-entered command will not change. If the same command is entered again, that command will have a new number.
4. The shell gets a line of text containing commands and their arguments.
5. The shell parses the line into tokens.
6. Parsing handles *unnested, quoted arguments* containing matching, single or double quotes. Each quoted string shall be treated as a *single argument* of the command passed to the shell.

   Each quote mark will have its correct, matching type in the input, and there will not be any cases of quoted sequences inside other quoted sequences. (Test input will not have unbalanced or unmatched quotation characters; the input of quotation marks will be balanced and matching in all tests.)
7. The shell shall maintain a set of *internal commands that it does not fork*. When the shell recognizes an internal command name, it will execute a function that fulfills the functionality associated with that name.
8. The shell detects and handles internal commands (e.g. help) before the *fork point*.
9. The shell *forks and execs* any *non-empty*, external command entered by the user.
10. The shell waits for command completion. Note that there is more to do than simply call `wait`. The child may have exited normally or not. The specifics on return codes that wait returns are in the page *man -s 2 wait*, where you will find some additional functions that process the returned status based on whether the child exited or was terminated by a signal. These functions post-process the raw exit value to extract the most typical error codes.
11. The shell reports errors in execution (see Errors).
12. The shell shall provide a command line option for a *verbose mode* of operation. When invoked with a *-v* command line argument, the shell will display the command, the results of parsing a non-empty command, and the major operations that occur in processing the command. See the

Examples.

Internal Commands

The internal commands and their behavior are listed below in alphabetical order. Note that each internal command *should ignore any extra arguments beyond those that are required* for their proper execution.

- `!N`: Re-execute the *Nth* command in the history list. This involves looking up the command in the list, fetching the command and all its arguments, and rerunning it to create the process another time. Remember that the *N* must be valid and within the range of the current history. If the *N* value is out of range, the shell should issue a message to `stderr` that describes what happened.

  This *bang* command works only if it is the first token on the command line.

- `help`: Print a list of internal commands. The output format should be a *reasonable design of your own* that provides suitable information.

- `history`: Print a list of commands executed so far, including their arguments. The shell should number the history list *starting from 1*. The output format should be a *reasonable design of your own* that provides suitable information.

  By default, the shell should *remember only the last 10 commands entered*. The history command should display up to the last 10 commands that were entered. If less than 10 have been entered, display all of them.

  If the user has changed the default number of remembered history using a command line option, then the value of 10 will be different (See the Interface Requirements).

  The numbering must be in old to new command order, top to bottom. The first command listed in the history must be the 'oldest' command, and the last command listed is the most recent.

  As the number of commands grows monotonically, the numbering of the commands in the history shall grow with it. The earlier section named *Examples* shows how the numbering sequence should change as the user enters more commands.

- `quit`: Clean up memory and gracefully terminate the shell.

- `verbose on/off`: Turn the shell verbose on or off based on the argument passed to the verbose command. When verbose is on, the shell should print a table or list of commands and their arguments as shown in the examples. After the command executes, the verbose mode should print the execution status value returned to the shell by the command. This is also shown in the examples.

  The output format should be a *reasonable design of your own* that provides suitable information.

  If the user enters something other than 'on' or 'off', the shell should print a usage message like this:

  ```
  usage: verbose [on|off]
  ```

Interface Requirements

Note: These requirements affect the ability to test the submitted solution. **Test failures due to non-conformance to these requirements are treated the same as failing functionality requirements.**

1. The user starts the shell following this pattern:

   ```
   $ ./mysh [-v] [-h pos_num]
   ```

   The text inside the *[]* are optional command line arguments, and these options can be in the order shown or the reverse order.

   The *-v* option turns on the verbose mode, which the user can turn off inside the shell using the *verbose* internal command.

   The *-h pos_num* option sets the size of the history list to be the value of *pos_num*, which must be a positive integer. For example, if the user enters this invocation:

   ```
   $ ./mysh -h 5
   ```

   then the shell maintains a history list that is only 5 commands long. The length of the history list cannot be changed after the shell starts.

   If the user entered a number that is 0 or negative for the history length, then the shell should issue the following usage message to `stderr`, and terminate with a failure status. Here is an example:

   ```
   $ ./mysh -h 0
   usage: mysh [-v] [-h pos_num]
   ```

   To process command line options such as these, refer to the man page on `getopt` using the command *man -s 3 getopt*, where you will find instruction on the `getopt` library function and examples on processing standard Linux command line options.

2. The `main()` function must be in a file named `mysh.c`. The function prototype or signature shall begin this way:

   ```
   int main( int argc, char * argv[] ) { // ...
   ```

   with the parameters following the usual naming conventions. Failure to match this may prevent `gmakemake` from generating a *Makefile* that will allow compiling, linking, testing and submission.

3. The prototype for *all of mysh's internal commands* shall be:

   ```
   int mysh_command_name( int argc, char * argv [] );
   ```

   where `mysh_command_name` is the name of the internal function (e.g. `verbose`).

4. The compilation and linking of your program shall produce no warnings when compiled with these compiler flags: `-Wall`, `-Wextra` and `-pedantic`.

5. It is possible for a user to enter an arbitrarily long command line; your design will need to take that into account. For input, you may want to use the `getline` function. To use the `getline` function with the `gcc` compiler in *c99* standard mode, you need to define the symbol `_GNU_SOURCE` when you compile.

Other Requirements

Prompts and Commands

While the prompt for each command contains a sequence number, that number must increase only if the last command was *not* an empty command. In other words, if the user hits the *ENTER* key producing an empty string command, *the command sequence number must not increase*.

The only commands that the shell may fork are non-empty commands and non-internal commands. When the shell receives an internal command, it executes that command as part of the shell process itself.

Termination Behavior (and Control Signals)

When the user issues a *Control-D* at the prompt, it tells the shell that there is no more input; the shell must then terminate, free all its dynamically allocated memory and exit. Note that the *Control-D* must be the value entered immediately at the prompt, before any other character; otherwise it does not cause the shell process to terminate.

With production shells, such as `bash`, it is possible for the user to enter *Control-C* into a terminal window to terminate a process that the `bash` shell started. While the operating system delivers the signal to the shell as well as the process, the `bash` shell does not terminate.

The `mysh` shell should not program signal handlers. A *Control-C* entered into a terminal window running your shell will terminate both a child process of the `mysh` shell and the `mysh` shell itself. This is the *expected, acceptable behavior*.

Errors

A **fatal error** occurs when the shell encounters an error reading input. The shell can no longer continue since it cannot get input to process. As a result, the shell should clean up memory allocations and terminate with an error message reporting what happened. To report fatal errors, the shell should issue a message using the `perror` system function.

A **non-fatal error** occurs when the shell tries to execute a user-entered command that fails or does not exist. The shell must continue since it can get the next command to process. To report non-fatal errors, the shell should issue a message to `stderr` that describes what happened.

Documentation and Style

Your solution must conform to these requirements to get full documentation and style credit.

1. The code shall follow the coding conventions established by the course.
2. Each file shall contain an `Author` entry using standard documentation generation tags. The value shall be the full name of the author.
3. All source files shall be controlled by a version management and control facility. The submitted source files shall identify the version control mechanism used **and the repository location (path) if applicable**. Acceptable version control facilities include:

- `RCS`
- `CVS`
- `Subversion (SVN)`
- `git`: the repository must be **private**!

Implementation Notes

This section provides suggestions, guidelines, sources of information for research and other helpful hints.

Sometimes you want to return more than one value from a function. Choices include returning those values through function parameters (which must be *pointers*) or creating a structure that the function may return.

The shell needs to check the results of calling system/library functions and report errors appropriately. The specifics on return codes appear in the *man pages* of the subject functions. Useful functions for error reporting include: `fprintf`, `feof`, `ferror`, and `perror`.

Designing the Implementation

Major approaches to designing are *top-down* and *bottom-up*. Each has its advocates and detractors, and each has its plusses and minuses. Of course, there is also the combined approach -- alternating a little of both ways.

Top-down Design

One way to start is top-down. For this problem, the requirements provide an outline of tasks that you could follow to decompose the problem into smaller ones. You could follow this process recursively to smaller and smaller problems until it is easy to translate the smaller problems into a function.

You could start with the *prompting* and *looping* behavior first. Then add input retrieval. Then add parsing. And so forth. As you find repeated functionality, you can *factor out* that behavior into utility functions to call from multiple places in the code.

As top-down design and implementation proceed, the program function emerges as you integrate components.

Bottom-up Design

Another way to start is bottom-up. In this case, you identify and write lower-level functions. Examples of functions you might write bottom-up include the internal functions (e.g. `help`), and memory allocation/deallocation utility functions.

For bottom-up development, a *test harness* is important because you do not have the high-level, skeleton program structure that top-down design typically creates first. This test harness is a program that starts testing one function and evolves to test more and more functions as you develop them.

As you write each function, you add test operations to the test harness that will test the function independently of others. When you reach the 'top' level from the bottom, you create a new program -- the end product -- that integrates all the functions together. This way you get two programs: the test harness and the product.

Submission

This project is due on the date shown on the course schedule.

You must include a **readme.txt** file with your submission to describe your design and identify problems that you know about. This file must be *plain text*.

Use this command pattern to submit your program:

```
try grd-243 project3-1 readme.txt *.h mysh.c [other .c .h files needed]
```

The main function *must* be in a file called `mysh.c`. Otherwise your submission will be rejected, and you will fail the project.

Grading

This project has the following grade percentage breakdown:

- 5% : Startup (prompting and looping to handle input)
- 20% : Handling command line input and errors (parsing operations)
- 25% : External command processing (fork, exec and wait operations)
- 25% : Internal command processing (bang, history, verbose, help)
- 5% : Design Quality (reasonable file, function and module structuring) and Build Cleanliness (no make warnings)
- 10% : Documentation (publishable documentation for all functions)
- 10% : Style (consistent indentation, spacing, line length for readability)

This breakdown does not include deductions for failure to manage memory allocations properly or initialize variables to eliminate `valgrind`-detected errors.