Final Project

`1730sh` - A Unix Shell

Pair Programming Project

DUE THU 2016-12-08 @ 6:00 PM

CSCI 1730 – Fall 2016

Contents

1 Project Description

For this project, you and your partner are going to implement a Unix shell (similar to bash) in C++ that supports background/foreground job control, pipelining, and IO redirection. Your code will make use of low-level system calls such as `fork`, `exec`, `pipe`, and related functions. You are **NOT** allowed to use the `system`, `popen`, and `pclose` system calls or functions.

This project is a natural extension of labs 13 and 14. Students should read the entire project description before starting and plan appropriately. I recommend tackling the project in phases: i) complete lab 13; ii) complete lab 14; then iii) complete the project. Both lab descriptions were released on eLC at the same time as this project description.

1.1 Functional Requirements

Here you will find detailed information about the functional requirements for your shell.

- **Prompt (20 points):**

  Your shell should read input from the prompt and divide it into words and operators, employing quoting rules (see below) to define the meaning of each character of input. Then these words and operators are translated into commands and other constructs, which return an exit status available for inspection or processing.

  ```
  1730sh:~/cs1730/p4/$ cat filename
  ```

  Using double quotes, the literal value of all characters enclosed is preserved, except for the backslash character. The backslash retains its meaning only when followed by another backslash or newline. Within double quotes, backslashes are removed from the input stream when followed by a double quote character. In other words, a double quote may be quoted within double quotes by preceding it with a backslash. The following example would have `argc == 2`:

  ```
  1730sh:~/cs1730/p4/$ echo "my \"awesome\" shell"
  ```

Your shell's prompt should include the text `"1730sh:"` followed by the present working directory, the `$` symbol, and a single space. If the present working directory is under the user's home directory, then a tilde (~) should replace the absolute path of the home directory in the prompt.

Note: Your shell should ignore the job control signals, however jobs executed using your shell should not. For example, typing `C-c` to send `SIGINT` should not terminate your shell.

- **Job Control (20 points):**

  In addition to job control facilitates that you have via your `kill` implementation (see Project 2), your shell needs to support launching jobs in both the foreground and the background. Your shell also needs to facilitate job bookkeeping.

  - *Bookkeeping:* Each job needs to have an associated job identifier (JID). I recommend placing all the processes associated with a job into the same process group, then using the process group id (PGID) as the JID. When a job encounters a status change (e.g., termination, stopped, continued, etc.), your shell needs to display a message letting the user know about the status change. This output should be similar to the output of the `jobs` builtin command (detailed later in this document). Here are some examples of status change messages:

    ```
    1137 Stopped           less &
    2245 Continued         cat /dev/urandom | less &
    2343 Exited (0)        emacs
    2345 Exited (SIGKILL)  emacs
    ```

  - *Foreground Jobs:* Nothing special needs to be done in order to launch a job in the foreground. When a foreground job is launched, your shell must wait until the job encounters a status change before re-prompting the user.

  - *Background Jobs:* A job may be launched in the background by appending an ampersand (&) after the job in your prompt. When a job is run in the background, your shell must **NOT** wait before re-prompting the user.

- **Pipelining (20 points):**

  Your shell needs to support pipelined jobs, where the standard outputs and standard inputs of the processes in the job are chained together using a Unix pipe (see `pipe(2)`). Remember, you are not allowed to use `popen` or `pclose`. A user should be able to execute a pipelined job using your shell by separating the individual commands by the vertical bar (|) operator. For example:

  ```
  1730sh:~/cs1730/p4/$ cat file | grep // | less
  ```

  In the example above, there are two pipes. The first pipe chains the standard out of the first process to the standard input of the second process, and the second pipe chains the standard out of the second process to the standard input of the third process.

- **IO Redirection (20 points):**

  Your shell needs to support the redirection of standard input, standard output, and standard error. In the examples below `COMMAND` may be a simple job or a pipelined job.

  - Redirect Standard Input using `<`

    ```
    1730sh:~/cs1730/p4/$ COMMAND < myfile.txt
    ```

  - Redirect Standard Output using `>` (truncate) or `>>` (append)

    ```
    1730sh:~/cs1730/p4/$ COMMAND > myfile.txt
    1730sh:~/cs1730/p4/$ COMMAND >> myfile.txt
    ```

  - Redirect Standard Error using `e>` (truncate) or `e>>` (append)

    ```
    1730sh:~/cs1730/p4/$ COMMAND e> myfile.txt
    1730sh:~/cs1730/p4/$ COMMAND e>> myfile.txt
    ```

  Your shell should support multiple redirections at once. For example, the following is considered to be valid:

  ```
  1730sh:~/cs1730/p4/$ COMMAND < input.txt > output.txt e>> log.txt
  ```

For the purposes of this project, you may assume the following order for redirects will be respected: `<`, `>`, `>>`, `e>`, `e>>`. You may also assume that the truncate and append redirects for a particular file are mutually exclusive.

- **Additional Builtins (20 points):**

  Your shell needs to support the following builtin commands:

  - `bg JID` – Resume the stopped job JID in the background, as if it had been started with `&`.
  - `cd [PATH]` – Change the current directory to `PATH`. The environmental variable `HOME` is the default `PATH`.
  - `exit [N]` – Cause the shell to exit with a status of `N`. If `N` is omitted, the exit status is that of the last job executed.
  - `export NAME[=WORD]` – `NAME` is automatically included in the environment of subsequently executed jobs.
  - `fg JID` – Resume job JID in the foreground, and make it the current job.
  - `help` – Display helpful information about builtin commands.
  - `jobs` – List current jobs. Here is an example of the desired output:

    ```
    JID  STATUS       COMMAND
    1137 Stopped      less &
    2245 Running      cat /dev/urandom | less &
    2343 Running      ./jobcontrol &
    ```

  - `kill [-s SIGNAL] PID` – The `kill` utility sends the specified signal to the specified process or process group `PID` (see `kill(2)`). If no signal is specified, the `SIGTERM` signal is sent. The `SIGTERM` signal will kill processes which do not catch this signal. For other processes, it may be necessary to use the `SIGKILL` signal, since this signal cannot be caught. If `PID` is positive, then the signal is sent to the process with the ID specified by `PID`. If `PID` equals 0, then the signal is sent to every process in the current process group. If `PID` equals -1, then the signal is sent to every process for which the utility has permission to send signals to, except for process 1 (init). If `PID` is less than -1, then the signal is sent to every process in the process group whose ID is `-PID`. When the `-s SIGNAL` option is used, instead of sending `SIGTERM`, the specified signal is sent instead. `SIGNAL` can be provided as a signal number or a constant (e.g., `SIGTERM`).

1.2 Nonfunctional Requirements

Your submission needs to satisfy the following functional requirements (points presented will be deducted if requirement is not met):

- **Directory Setup (5 points):** Make sure that all of your files are in a directory called `LastNameA-LastNameB-p4`, where `LastNameA` and `LastNameB` are replaced with the actual last names of you and your pair programming partner.
- **Libraries (50 points):** You are allowed to use any of the C or C++ standard libraries. When reading or writing to a file are concerned, you need to use low-level calls to `read(2)` and `write(2)` and related functions. Whenever possible, program output should be unbuffered. You are NOT allowed to use the following system calls in your implementation: `popen(3)` and `system(3)` (or related functions). Failure to adhere to this non-functional requirement will result in an automatic 50 point deduction.
- **Documentation (5 points):** All classes, structs, and functions must be documented using Javadoc (or Doxygen) style comments. Use inline documentation, as needed, to explain ambiguous or tricky parts of your code.
- `Makefile` **File (5 points):** You need to include a `Makefile`. Your `Makefile` needs to compile and link separately. That is, make sure that your `Makefile` is setup so that your `.cpp` files each compile to individual `.o` files. This is very important. The resulting executable names should correspond to the ones presented in the functional requirements.
- **Standards & Flags (5 points):** Make sure that when you compile, you pass the following options to `g++` in addition to the `-c` option:

  ```
  -Wall -std=c++14 -g -O0 -pedantic-errors
  ```

  Other compiler/linker options may be needed in addition to the ones mentioned above. The expectation is that the grader should be able to type `make clean` and `make` in the following to clean and compile/link your submission, respectively.
- `README` **File (5 points):** Make sure to include a `README` file that includes the following information presented in a reasonably formatted way:

- Your Name and 810/811#
- Instructions on how to compile and run your program.

Make sure that each line in your `README` file does not exceed 80 characters. Do not assume line-wrapping. Please manually insert a line break if a line exceeds 80 characters.

- **Compiler Warnings (5 points):** Since you should be compiling with both the `-Wall` and `-pedantic-errors` options, your code is expected to compile without `g++` issuing any warnings. Failure to adhere to this non-functional requirement will result in an automatic 5 point deduction.
- **Memory Leaks (5 points):** Since this project may make use of dynamic memory allocation, you are expected to ensure that your project implementation does not result in any memory leaks. We will test for memory leaks using the `valgrind` utility. Failure to adhere to this non-functional requirement will result in an automatic 5 point deduction.

1.3 Collaboration Policy

Students are required to work in groups of two for this project. Furthermore, each group needs to engage in pair programming. Pair programming is an agile software development technique in which two programmers work together at one workstation. One, the driver, writes code while the other, the navigator, reviews each line of code as it is typed in. The two programmers switch roles frequently.

Yes, this involves physically meeting with your partner. Not being able to find time is **NOT** an excuse for not participating. If you need a place to meet and work on the project then I suggest the 307 lab in Boyd. The iMacs are already setup with everything you need. You login to them with your Nike account, open up the "Terminal" application and SSH into your Nike account.

1.3.1 Some Pair Programming Guidelines

- You and your partner should work together as much as possible, with the stipulation that at most 25% of your total time coding, testing, and debugging on the assignment can be done alone.
- When the pair gets back together after either partner has worked on the code alone, review, line by line, the work done alone before doing any new work. This is really easy if each person maintains their own branch and commits as they work.
- You and your partner should alternate driving and navigating, spending roughly equal amounts of time in each role.
- After the project is completed, a pair programming survey will be made available for both partners to complete.

1.3.2 GitHub Repositories

Private Git repositories on GitHub are available upon request for each team. Please contact your instructor to have a repository created for your team.

1.4 Fork Bomb Policy

Fork bombing Nike will hurt your grade. If I find out that you have fork bombed Nike, there will be a 25-point deduction from your project for the first offense. Log in to one of the `cf` cluster nodes to avoid this. Also, if you are forking processes within a loop, use a simple counter to make sure the loop doesnt execute more than a handful of times while you are still debugging. It is better to be safe than sorry.

1.5 Submission

Before the due date, you need to submit your code via Nike. Make sure your work is on `nike.cs.uga.edu` in a directory called `LastName1-LastName2-p4`. From within the parent directory, execute the following command:

```
$ submit LastName1-LastName2-p4 cs1730a
```

It is also a good idea to email a copy to yourself. To do this, simply execute the following command, replacing the email address with your email address:

```
$ tar zcvf LastName1-LastName2-p4.tar.gz LastName1-LastName2-p4
$ mutt -s "p4" -a LastName1-LastName2-p4.tar.gz -- your@email.com < /dev/null
```

2 Frequently Asked Questions (FAQ)

2.1 Why does `waitpid(2)` return `-1` when I explicitly ignore `SIGCHLD`?

Please read the following from `wait(2)`:

> A child that terminates, but has not been waited for becomes a "**zombie**". The kernel maintains a minimal set of information about the zombie process (PID, termination status, resource usage information) in order to allow the parent to later perform a wait to obtain information about the child. As long as a zombie is not removed from the system via a wait, it will consume a slot in the kernel process table, and if this table fills, it will not be possible to create further processes. If a parent process terminates, then its "zombie" children (if any) are adopted by `init(8)`, which automatically performs a wait to remove the zombies.

The next paragraph is of particular interest:

> `POSIX.1-2001` specifies that if the disposition of `SIGCHLD` is set to `SIG_IGN` or the `SA_NOCLDWAIT` flag is set for `SIGCHLD` (see `sigaction(2)`), then children that terminate do not become zombies and a call to `wait(2)` or `waitpid(2)` will block until all children have terminated, and then fail with `errno` set to `ECHILD`. (The original `POSIX` standard left the behavior of setting `SIGCHLD` to `SIG_IGN` unspecified. Note that even though the default disposition of `SIGCHLD` is "ignore", explicitly setting the disposition to `SIG_IGN` results in different treatment of zombie process children.) Linux 2.6 conforms to this specification. However, Linux 2.4 (and earlier) does not: if a `wait(2)` or `waitpid(2)` call is made while `SIGCHLD` is being ignored, the call behaves just as though `SIGCHLD` were not being ignored, that is, the call blocks until the next child terminates and then returns the process ID and status of that child.

In the description provided by Lab 14, you're instructed to ignore certain job control signals, one of which is `SIGCHLD`. As described in the manual, this signal is ignored by default. This means that you do NOT need to explicitly set the signal disposition to `SIG_IGN` unless you want to prevent the creation of zombies. This may or not be the behavior you desire/need for the child processes created by your shell, so keep this in mind.

2.2 Why do I get weird hanging issues with pipes?

Please read the `pipe(7)` manual page... Here is one paragraph of particular interest (as of this writing):

> If all file descriptors referring to the write end of a pipe have been closed, then an attempt to `read(2)` from the pipe will see end-of-file (`read(2)` will return 0). If all file descriptors referring to the read end of a pipe have been closed, then a `write(2)` will cause a `SIGPIPE` signal to be generated for the calling process. If the calling process is ignoring this signal, then `write(2)` fails with the error `EPIPE`. An application that uses `pipe(2)` and `fork(2)` should use suitable `close(2)` calls to close unnecessary duplicate file descriptors; this ensures that end-of-file and `SIGPIPE/EPIPE` are delivered when appropriate.

In other words, if you're experiencing weird "hanging" issues related to pipes, it's probably because you haven't closed out your file descriptors.

2.3 How do I interpret the program argument syntax for builtin commands?

The notation used for the arguments of a utility imposes requirements on the implementors of the builtin and provides a simple reference for the application developer or system user.

1. Arguments or option-arguments enclosed in the `'['` and `']'` notation are optional and can be omitted. Conforming applications shall not include the `'['` and `']'` symbols in data submitted to the utility.
2. Arguments separated by the `'|'` (`<vertical-line>`) bar notation are mutually-exclusive. Conforming applications shall not include the `'|'` symbol in data submitted to the utility.
3. Ellipses (`"..."`) are used to denote that one or more occurrences of an operand are allowed. When an option or an operand followed by ellipses is enclosed in brackets, zero or more options or operands can be specified.

2.4 How do I parse the program arguments for builtin commands?

The `getopt(3)` function can be used to handle options and operands that conform to the command argument descriptions provided in this project description. In addition to the manual page for `getopt(3)`, a description of the function and its parameters as well as an example of how to use it are provided APUE 17.6.

2.5 How do I achieve unbuffered output?

The best way to guarantee that output is unbuffered (i.e., characters at the destination as soon as possible) is to directly call `write(2)`. If you are using `printf(3)`, you should disable output buffering using `setvbuf(3)`. If you are using C++ output streams (e.g., `cout`), then you should disable output buffering using `setf` and `unitbuf` .