Project 1: tsh - A tiny shell

EECS-343 - Fall 2014

Important Dates

Out: *Wednesday, September 24th, 2014.*

Due: *Friday, October 10th, 2014 (11:59 PM CDT).*

Project Overview

For this project you are asked to write a simple command line interpreter or shell. Your shell should behave much like any other Unix shell (`sh`, `csh`, `tcsh`, etc) allowing a user to background and foreground jobs, pipe output, and redirect jobs' standard input/output.

You are to work in groups of two (2) people for this project. While groups of three (3) are allowed, these groups **must complete all of the required and extra credit cases for full credit**. Skeleton code to begin is provided on the course website. The only hand-in will be electronic. Any clarification and/or revision will be posted on the course web page.

Educational Objectives

This project aims at reinforcing your understanding of processes, process control and signaling, as well as making you more familiar with the developing environment and the set of available tools (such as `gcc`, `gdb` and `make`).

The `tsh` Specification

For this project you are asked to write a **C** program that will act as a shell command line interpreter for Unix. Your shell program should use the same style as the Bourne shell for running programs.

In particular, your shell should have the following features:

- The prompt should be the empty string. For debugging purpose, you might find it helpful to set it to a string such as "`tsh>` ".
- When the user types a line such as

  `command argument_1 argument_2 ... argument_n`

  the framework's code will generate `argv` for you. `tsh` should search the directory system (in the order specified by the `PATH` environment variable) for a file with the same name as `command` (which may be a relative filename or a full pathname). If the file is found, then it should be executed with the optional parameter list, as is done with `sh`. Use an `execv()` system call (rather than any of the `execl()` calls) to

execute the file that contains the command. You will need to become familiar with the `fork()` and `wait()` functions and the `execv` family of system calls (a set of interfaces to the `execve` system call).

- Typing `ctrl-c` should cause a SIGINT signal to be sent to the current process, as well as any descendants of that process (e.g. any child processes that it forked).
- Typing `exit` at a prompt should cause your shell to cleanly terminate.
- Finally, `tsh` should reap all of its zombie children. If any job terminates because it receives a signal that it didn't catch, then `tsh` should recognize this event and print a message with the job's PID and a description of the offending signal.
- Typing `ctrl-z` should cause a SIGTSTP signal to be sent to the current foreground job, as well as any descendants of that job (e.g. any child processes that it forked). If there is no foreground job, then the signal should have no effect.
- `tsh` should reap all of its zombie children. If any job terminates because it receives a signal that it didn't catch, then `tsh` should recognize this event and print a message with the job's PID and a description of the offending signal.
- Putting a given job in the background (terminating the command string with **"&"**). The shell should print a message including a sequential job number.
  Example: `eleuthera % xemacs &` runs xemacs and returns a prompt to the user immediately.
- Three additional builtins for job control:
  - `fg` should return a backgrounded job to the foreground. It should take an optional job number as an argument, defaulting to the most recently backgrounded job.
  - `bg` should send SIGCONT to a backgrounded job, but should not give it the foreground. (i.e., The user should immediately be able to issue further commands.) It takes an optional argument that works the same way as `fg`.
  - `jobs` should print out a table of all currently running jobs.

  For each of these, you should mimic `bash`. Try things out there to see exactly what your shell should print.

For Extra Points

There are two ways to get extra points in this project: you can implement the `alias` command or implement **both** pipes and IO redirection.

Alias command (+10 Points)

Real shells implement a long list of additional builtin commands, two of the most commonly used are the *alias* and *unalias* commands. Aliases allow a string to be substituted for a word when it is used as the first word of a simple command. The shell maintains a list of aliases that may be set and unset with the *alias* and *unalias* commands. For full extra credit, you must implement aliases to the following specification:

- The first word of each simple command, if unquoted, must be checked to see if it has an alias. If so, that word must be replaced by the text of the alias. The characters '/', '$', '=' and any other shell metacharacters or quoting characters may not appear in an alias name. The replacement text may contain any valid shell input, including shell metacharacters.
- A word that is identical to an alias being expanded must not be expanded a second time. This means that if we alias *"ls"* to *"ls -F"*, for instance, your shell should not try to recursively expand the replacement text.

- If the last character of the alias value is a space or tab character, then the next command word following the alias must also be checked for alias expansion.

Modify your interpreter so that it creates and lists aliases with the *alias* command, and removes aliases with the *unalias* command. For example, when the user types:

```
% alias ls_alias='ls -l'
```

the command 'ls_alias' should be automatically substituted by its equivalent 'ls -l' next time it is used. Similarly, in order to remove this alias, the user should be allowed to type:

```
% unalias ls_alias
```

Finally, typing the command:

```
% alias
```

without any arguments should print the list of all the aliases defined by the user.

Pipes (+15 Points)

With the pipe symbol (|) you can have the output of one command fed (piped) to the input of another. Example: `eleuthera % w | more` shows who is in the system one screen at a time.

Much of the power of the Unix command line comes from the ability to string together complex instructions using simple tools connected by pipes. Your program should not have any particular limit on how many syntactically correct pipes the user can put into one command.

I/O redirection (+10 Points)

Your shell should support input (<) and output (>) redirection. When in doubt, check how `bash` does it and replicate that behavior. Example: `eleuthera % w > who.txt` saves a list of the users logged into `eleuthera` and what they are doing in the file `who.txt`.

Testing your code

Your project will be graded based on its ability to run correctly on the virtual machine image distributed through the class. This is due to the minor differences in compilation and program behavior across different operating systems, gcc and system libraries. For this course, your projects will be graded on a distributed virtual machine image of 32-bit Ubuntu 14.04 LTS; the instructions for retrieving this image are available at `http://aqualab.cs.northwestern.edu/classes/EECS343/343-files/vm_instructions.html`.

We will use a set of test cases to evaluate your code against a reference implementation. The test cases included are the **only** test cases we will use to evaluate your submission. To support sending signals (e.g. SIGINT and SIGTSTP) via an automated test harness to your executable, we are using a script located in the testsuite directory called sdriver.pl.

Testing Locally To run an individual test case outside of the test harness, go to the "testsuite" directory. Run: `./sdriver.pl -t testXX.in -s .././tsh` Similarly, you can run the reference implementation over that test case by replacing ".././tsh" with "./tsh- orig". Note that the testsuite runs a script that makes a set of files in a temporary directory, so the output of a particular testcase may differ depending on whether or not you are running it from the testsuite. You can also generate your own test cases following the format of the testsuite/test*.in files. A line with "INT" or "TSTP" sends the interrupt or stop signals to the shell. ``SLEEP N'' will cause the driver to wait for `N` seconds before continuing. All other lines are sent to the shell as input.

Testing on VM Testing on the virtual machine guarantees that there are no inconsistencies with your code and our testing environment. Once the virtual machine has been correctly installed following the instructions above, start the vm using the command `make start-vm` (the vm can be killed with `make kill-vm`). The make command `make test-vm` copies test code onto the virtual machine, and runs the test cases from within the VM.

Evaluation, deliverables, and hand-in instructions

The project is graded out of 100 points. You can earn up to 35 extra points, for a total of 135 points, by passing the extra credit test cases (7 test cases at 5 points each). We will deduct points for the following reasons:

- 100 points!: it doesn't compile when we type make
- up to 10 points: no documentation (tsh.1 man page file) or poorly self-documented/commented code
- 4 points per failed test case (there are 20 basic test cases)
- 2 points per compiler warning
- 1 point per memory leak warning
- 5 points per definitive memory leak

If you hand in your project late, we will deduct 10% from your final score per day or portion thereof. We will not accept submissions that are more than three days late.

When grading your project, we will use the same set of test cases that is included with the skeleton. This should avoid any surprises with the grading process. The deliverable for this project should include:

1. The source code for `tsh: tsh.{c,h}, interpreter.{c,h}, io.{c,h}, runtime.{c,h}, config.h,` any other files you have added, and the Makefile.
2. A mini man page describing your shell: `tsh.1`. Please include a brief (less than a page) description of any important design decisions you made while implementing tsh. We ask that you format your documentation as a "man" page, following the template in the `tsh.1` file. For more information, see this HOWTO guide for man pages: `http://www.schweikhardt.net/man_page_howto.html`. We should be able to execute `'groff -man -Tascii tsh.1'` and see your formatted man page.
3. If you attempted to pass the extra credit test cases, please make a note of it in your man page.

To hand in your project:

- Change the working directory to the project directory.
- Set your team name in Makefile: replace 'whoami' with "yourNetid1+yourNetid2" for the TEAM variable.
- Invoke `make test-reg`, which builds the deliverable tar.gz and runs the test cases.
- **If and only if the test cases terminate, you may submit the handin**.
- Submission will be done through a dedicated webpage (check the project section of the class web site for the URL).
- After running the handin procedure, if you discover a mistake and want to submit a revised copy, follow the same procedure and resubmit. The submission page keeps track of all your submissions and considers the last one as your final submission.
- A few minutes after submitting your handin on the submission site, you will receive an email confirming your submission, with the output of the testsuite running on our submission server. If you havent received a confirmation within an hour, contact the TA to verify that your submission was received.

Code Review and Walkthrough

As part of every project grading a subset of 2-8 teams will be randomly selected to conduct a walkthrough of their code for the TAs and instructor. A walkthrough is a form of software peer review in which a designer or programmer leads participants through the software product, while the participants ask questions and make comments on code functionality, style, compliance to standards, etc [1].

Note: To be granted full credit for a project you must be able to carry the walkthrough and unquestionably show a full understanding of your own code.

A few points on the topic:

- For each of the four projects during the quarter, between 2-8 teams will be selected to conduct a walkthrough.
- The selection will be random with replacement; in other words, your team could be selected multiple times over the quarter.
- The walkthrough must be done in the first week immediately following submission.
- All team members must be present; one of the members will conduct the walkthrough at a time while the other(s) take note of the questions and comments.
- All other participants, TAs, instructor and members of other teams can ask question about and comment on the code.
- All member of the team conducting the code-review must drive the walkthrough at different times during the meeting.
- Members of other teams present during a code review session are require to provide comments on the code being presented (besides their own). They can submit their comments to the TA after the session for a total of 2.5% extra (from total) points.

Background Information

A command line interpreter is a mechanism by which each interactive user can issue commands to the operating system. Whenever a user has successfully logged into the computer, the operating system causes the user process (assigned at login) to execute a particular shell which provides the default, text-based, human-computer interface.

Once initialized, the shell prints a prompt at the beginning of each line:

```
(10:22am) ~/classes %
```

The command line is normally of the form: `command argument_1 argument_2 ... argument_n`

where the first word is the command to be executed and the remaining words are arguments expected by the command. The actual number of arguments depends on the given command.

By convention the command specified on the command line is either the name of a file that contains an executable program (e.g. `perl`, `ls`, `gcc`, ...) or a built-in command implemented by the shell (e.g. `cd, echo, history`, ...). If the command is a built-in command, the shell immediately executes it in the current process. Otherwise, the command is assumed to be the pathname of an executable program and the shell will fork a child to execute it. The child processes created as a result of interpreting a single command line are known collectively as a *job*. In general, a job can consist of multiple child processes connected by Unix pipes.

[1] Additional information on code reviews and code walkthroughs can be found at `http://en.wikipedia.org/wiki/Software_walkthrough` and `http://saltlane.co.uk/Resources/fagan%20inspection.HTML`

Many shell programs are used within Unix variants, including the original Bourne shell (sh), the C shell (csh), the enhanced C shell (tcsh), the Korn shell (ksh) and the standard Linux shell (bash). Despite the differences, all follow a similar set of rules for command line syntax. The Bourne shell is described in Ritchie and Thompson's original Unix paper [2].

Consider the detailed steps that a shell must take to accomplish its job:

- Print a prompt. Normally there is a hard coded default prompt string, but the user can customize this to print the machine name, the current directory, etc.
- Get the command line. To get a command line, the shell performs a blocking read operation so that the process that executes the shell will be blocked until the user types a command line in response to the prompt. When the command has been provided by the user (and terminated with a NEWLINE character), the command line string is returned to the shell.
- Parse the command. The syntax for the command line is trivial. The parser begins at the left side of the command line and scans until it sees a white space character (such as space, tab, or end-of-line). The first word is interpreted as the command and the rest as parameter strings.
- Find the file. The shell will search for the file to execute and run it with the given parameters. The shell provides a set of environment variables, normally defined in the user's `.login`. One of them, `PATH`, is an ordered list of absolute pathnames that specifies where the shell should search for command files. For example, if `PATH` is set to `(/bin:/usr/bin/:/usr/local/bin)`, then the shell will first look in `/bin`, then in `/usr/bin`, and finally in `/usr/local/bin`. If no file of that name can be found in any of the specified directories, the shell will report the error to the user.
- Prepare the parameters. The shell simply passes the string parameters to the command as the `argv` array of pointers to strings.
- Execute the command. Finally the shell must execute the command by creating a new process for it. In the execution of the command, a common set of calls needed are: `fork()`, `execve()`, and `wait()`.

- `fork`. This system call creates child process which is an identical copy of its parent with the exception of its process identification and its set of pointers to shared kernel entities such as file descriptors.
- `execve`. This system call is used to change the program that the process is currently executing. When a process encounters the `execve` system call, the next instruction it executes will be the one at the entry point of the new executable file specified by its first argument. There are various versions of `execve` available at the system call interface; they differ in the way parameters are specified.
- `wait`. This system call is used by a process to block itself until the kernel signals the process to execute again. When the `wait` call returns as a result of a child process terminating, the status of the terminated child is returned as a parameter to the calling process.

Putting a process in the background

In the normal paradigm to execute a command, the parent process (the shell) creates the child and waits for it to terminate. If the "&" operator is used to terminate a command line, however, the shell is expected to execute concurrently with the child. While the child executes the command, the parent prints another prompt to standard output and waits for the user to enter another command line.

When a child process is created and starts executing its own program, both the child and the parent expect their standard input stream to come from the user via the keyboard and for their standard output stream to be

[2] "The Unix Time-Sharing System", *Communication of the ACM*, 17(7), pp 1897–1920, July 1974. You can find a pointer to the paper from the course webpage.

written to the character terminal display. Notice that if multiple child processes are running concurrently the user will not know which child process will receive data on its standard input or in which order the output will be written to the terminal display.

I/O redirection

A process has three default file identifiers: `stdin`, `stdout` and `stderr`. Normally `stdin` is mapped to the keyboard and `stdout` and `stderr` are mapped to the terminal display. The user can redefine these mappings whenever a command is entered, making use of "<" and ">"). It is the shell's job to set up the child processes' file descriptors.

The shell can redirect I/O by manipulating the child process's file descriptors. A newly created child process inherits the open file descriptors of its parent, specifically the same keyboard for `stdin` and the terminal display for `stdout` and `stderr`. The shell can change this by mapping the file descriptors 0, 1 and 2, respectively, to the preferred files.

A code fragment such as the following:

```
fid = open(foo, O_WRONLY | O_CREAT);
close(1);
dup(fid);
close(fid);
```

is guaranteed to create a file descriptor, `fid`, to duplicate the entry and to place the duplicate in fd[1] (the usual `stdout` entry in the process's file descriptor table). As a result, characters written by the process to `stdout` will be written to the file `foo`. This is the key to redirecting both `stdin` and `stdout`.

Shell pipes

The pipe is the main IPC mechanism in uniprocessor Unix. A pipe employs asynchronous send and blocking (could be changed to non-blocking) receive operations. Pipes are FIFO buffers designed with an API that resembles as closely as possible the file I/O interface. A pipe may contain a system-defined maximum number of bytes at any given time, usually 4KB.

A pipe is represented in the kernel by a file descriptor. A process that wants to create a pipe calls the kernel with a call like

```
int pipeID[2];
...
pipe(pipeID);
```

The kernel creates the pipe as a kernel FIFO data structure with two file identifiers. In this example code, pipeID[0] is a file pointer to the read end of the pipe and pipeID[1] is a file pointer to the write end of the pipe.

For two or more processes to use pipes for IPC a common ancestor of the processes (i.e. the shell) must create the pipe prior to creating the processes. Because the `fork` command creates a child that contains a copy of the open file table, the child inherits the pipes that the parent created and so both can communicate.

```
...
pipe(pipeID);
if (fork() == 0) {  /* child process */
  ...
```

```
  read(pipeID[0], childBuf, len);
  /* process the message in childBuf */
  ...
} else {  /* parent process */
  ...
  /* send a message to the child */
  write(pipeID[1], msgToChild, len);
  ...
}
```

As an example, look at how pipes can be used to implement concurrent processing between processes A and B, where A performs some computation, send a value x to B, performs some computation, reads a value y from B, etc.

```
int A_to_B[2], B_to_A[2];

main ()
{
  pipe(A_to_B);
  pipe(B_to_A);
  if (fork() == 0) { /* this is the first child process */
    close(A_to_B[0]); /* a process that does not intent to use a pipe
                         should close it so that EOF conditions can be
                         detected */
    close(B_to_A[1]);
    execve(``prog_A.out'', ...);
    exit(1);
  }
  if (fork() == 0) { /* this is the second child process */
    close(A_to_B[1]);
    close(B_to_A[0]);
    execve(``prog_B.out'', ...);
    exit(1);
  }
  /* this is the parent process code */
  wait ...
  wait ...
  ...
}

proc_A()
{
  while(TRUE) {
    write(A_to_B[1], x, sizeof(int));  /* using the pipe to send info */
    <compute A1>
    read(B_to_A[0], y, sizeof(int));   /* using the pipe to get info */
    <compute A2>
  }
}

proc_B()
{
  while(TRUE) {
```

```
    read(A_to_B[0], x, sizeof(int));
    <compute B1>
    write(B_to_A[1], y, sizeof(int));
    <compute B2>
  }
}
```

Reading multiple input streams

As any other file, the read end of a pipe can be configured, with an `ioctl()` call, to use nonblocking semantics. After the call has been issued on the descriptor, a read on the stream returns immediately, with the error code set to `EAGAIN`. In addition, `read()` returns 0 indicating that it did not read any information into the buffer. Alternatively, the program can determine whether `read()` succeeded by checking the length value to see if it is nonzero.

The `select()` command allows a process to poll all of its open input streams to determine which contain data. It then can execute a normal blocking read on any stream that contains data. Note that if multiple processes are reading the pipe and each uses a `select()`, then a race condition can result. Read the documentation in detail if you want to use this approach in your solution.

Job control in Unix

Unix shells support the notion of *job control*, which allows users to move jobs back and forth between background and foreground, and to change the process state (running, stopped, or terminated) of the processes in a job. Typing `ctrl-c` causes SIGINT signal to be delivered to each process in the foreground job. The default action for SIGINT is to terminate the process. Similarly, typing `ctrl-z` causes a SIGTSTP signal to be delivered to each process in the foreground job. The default action for SGTSTP is to place a process in the stopped state, where it remains until it is awakened by the receipt of a s SIGCONT signal. Unix shells also provide various built-in commands that support job control. For example: `jobs`, `bg` $< job >$, `fg` $< job >$ and `kill`.

Hints and a suggested plan of attack

Here is a suggested plan of attack including a number of debugging points for your project. **The first and most important thing to do is to familiarize yourself with the provided code and test cases.** Having a high level understanding of how the existing skeleton code works will make it much easier to begin to implement shell functionality. Being aware of the specific test cases you have to pass should make it easier to approach this project one step at a time, following this plan of attack. Please take a look at the file named README.api for a list of commands that may turn out to be useful as you work in your project. As always, you can consult the man page for any function to learn more about its use and effects. In addition, "Advanced Programming in the Unix Environment" (information on the course site) is a valuable reference for more detailed information on many relevant topics.

Suggested Plan of Attack

1. Organize the shell to initialize variables and then perform an endless loop until standard input detects an EOF condition, such as `ctrl-d` character or an exit message. Your first version should just print the prompt, read the command line and print it to standard output.

2. The provided framework already parses the command line on your behalf. It determines the strings on the command line and puts them into a `char *argv[]` array, computing the value for `int argc` on the way. To see it in action just print out the contents of `argv[]` and `argc`.

   (a) The provided framework also locates executable files in the current directory, by absolute path and specified in the `PATH` environmental variable.

3. Read the man pages for `fork, execv, wait(2)` and `exit(2)`.
4. Write some code to experiment with them.
5. Now implement the command line execution functionality using `fork/exec/wait`. Use `execvp` first and then get it working with `execv`, implementing the PATH search capability.
6. Modify your code to allow putting a job in the background by appending an `&` to a command.
7. Implement `bg`, `jobs`, and `fg`. You may find that order to be easiest.
8. Ensure your shell is able to handle signals properly (i.e. SIGTSTP, SIGINT, SIGCONT).
9. (Extra Credit) Implement pipes. Think ahead - if you do this correctly, multiple pipes won't be much more complicated than single ones.
10. (Extra Credit) Implement I/O redirection. Input and output are approximately equal in complexity.
11. (Extra Credit) Implement `alias` command within shell.
12. Finish the details and update the man page.
13. Test to exhaustion.
14. Submit and go celebrate.

A stripped-down shell will look something like this:

```
struct command_t {
  char *name;
  int argc;
  char* argv[];
  ...
};

int
main (int argc, char *argv[])
{
  ...
  struct command_t *command;

  ...

  /* shell initialization */

  while (TRUE) { /* repeat forever */
     /* print prompt */

     /* read command line and parse it */
```

```
    /* find the full pathname for the file */

    /* create a process and execute the command */

    /* parent wait for child finishing */
  }

  /* shell termination */

} /* end main */
```

Hints

- The `waitpid`, `kill`, `fork`, `execve`, `setpgid`, and `sigprocmask` functions will come in very handy. The WUNTRACED and WNOHANG options to `waitpid` will also be useful.
- When you implement your signal handlers, be sure to send `SIGINT` signals to the entire foreground process group, using "`-pid`" instead of "`pid`" in the argument to the `kill` function.
- One of the tricky parts of the project is deciding on the allocation of work between the `waitfg` and `sigchld_handler` functions. We recommend the following approach:
  - In `waitfg`, use a busy loop around the `sleep` function.
  - In `sigchld_handler`, use exactly one call to `waitpid`.

  While other solutions are possible, such as calling `waitpid` in both `waitfg` and `sigchld_handler`, these can be very confusing. It is simpler to do all reaping in the handler.
- In `eval`, the parent must use `sigprocmask` to block `SIGCHLD` signals before it forks the child, and then unblock these signals, again using `sigprocmask` after it adds the child to the job list by calling `addjob`. Since children inherit the `blocked` vectors of their parents, the child must be sure to then unblock `SIGCHLD` signals before it execs the new program.

  The parent needs to block the `SIGCHLD` signals in this way in order to avoid the race condition where the child is reaped by `sigchld_handler` (and thus removed from the job list) *before* the parent calls `addjob`.
- Programs such as `more`, `less`, `vi`, and `emacs` do strange things with the terminal settings. Don't run these programs from your shell. Stick with simple text-based programs such as `/bin/ls`, `/bin/ps`, and `/bin/echo`.
- When you run your shell from the standard Unix shell, your shell is running in the foreground process group. If your shell then creates a child process, by default that child will also be a member of the foreground process group. Since typing `ctrl-c` sends a SIGINT to every process in the foreground group, typing `ctrl-c` will send a SIGINT to your shell, as well as to every process that your shell created, which obviously isn't correct.

  Here is the workaround: After the `fork`, but before the `execve`, the child process should call `setpgid(0, 0)`, which puts the child in a new process group whose group ID is identical to the child's PID. This ensures that there will be only one process, your shell, in the foreground process group. When you type `ctrl-c`, the shell should catch the resulting SIGINT and then forward it to the appropriate foreground job (or more precisely, the process group that contains the foreground job).
- The `waitpid`, `kill`, `fork`, `execve`, `setpgid`, and `sigprocmask` functions will come in very handy. The WUNTRACED and WNOHANG options to `waitpid` will also be useful.
- When you implement your signal handlers, be sure to send `SIGINT` signals to the entire foreground process group, using "`-pid`" instead of "`pid`" in the argument to the `kill` function.

- One of the tricky parts of the project is deciding on the allocation of work between the `waitfg` and `sigchld_handler` functions. We recommend the following approach:
  - In `waitfg`, use a busy loop around the `sleep` function.
  - In `sigchld_handler`, use exactly one call to `waitpid`.

  While other solutions are possible, such as calling `waitpid` in both `waitfg` and `sigchld_handler`, these can be very confusing. It is simpler to do all reaping in the handler.

- In `eval`, the parent must use `sigprocmask` to block `SIGCHLD` signals before it forks the child, and then unblock these signals, again using `sigprocmask` after it adds the child to the job list by calling `addjob`. Since children inherit the `blocked` vectors of their parents, the child must be sure to then unblock `SIGCHLD` signals before it execs the new program.

  The parent needs to block the `SIGCHLD` signals in this way in order to avoid the race condition where the child is reaped by `sigchld_handler` (and thus removed from the job list) *before* the parent calls `addjob`.

- Programs such as `more`, `less`, `vi`, and `emacs` do strange things with the terminal settings. Don't run these programs from your shell. Stick with simple text-based programs such as `/bin/ls`, `/bin/ps`, and `/bin/echo`.

- When you run your shell from the standard Unix shell, your shell is running in the foreground process group. If your shell then creates a child process, by default that child will also be a member of the foreground process group. Since typing `ctrl-c` sends a SIGINT to every process in the foreground group, typing `ctrl-c` will send a SIGINT to your shell, as well as to every process that your shell created, which obviously isn't correct.

  Here is the workaround: After the `fork`, but before the `execve`, the child process should call `setpgid(0, 0)`, which puts the child in a new process group whose group ID is identical to the child's PID. This ensures that there will be only one process, your shell, in the foreground process group. When you type `ctrl-c`, the shell should catch the resulting SIGINT and then forward it to the appropriate foreground job (or more precisely, the process group that contains the foreground job).

Good Luck!