COMP 321: Introduction to Computer Systems

Project 1: Unix Shell
Assigned: 1/28/25, **Due: 2/12/25, 11:55 PM**

Important: This project must be done individually. Be sure to carefully read the course policies for assignments (including the honor code policy) on the assignments page of the course web site:

`http://www.clear.rice.edu/comp321/html/assignments.html`

Overview

You will write a simple Unix shell program that supports job control. The goals of this project are as follows:

- Work with a realistic system application.
- Become familiar with the concepts of process control.
- Become familiar with the concepts of signaling.

General Overview of Unix Shells

A *shell* is an interactive command-line interpreter that runs programs on behalf of the user. A shell repeatedly prints a prompt, waits for a *command line* on `stdin`, and then carries out some action, as directed by the contents of the command line.

The command line is a sequence of ASCII text words delimited by whitespace. The first word in the command line is either the name of a built-in command or the name of an executable file. The remaining words are command-line arguments. If the first word is a built-in command, the shell immediately executes the command in the current process. Otherwise, the shell assumes that the first word is the name of a program to execute. In this case, the shell forks a child process, then attempts to load and run the program in the context of the child. 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.

If the first word in the command line starts with a directory, such as "`.`", "`/`", or the name of a subdirectory in the current directory, the word is assumed to be the path name of an executable program, for example, "`./tsh`", "`/usr/bin/ls`", or "`comp321/count/count`". Otherwise, the first word is assumed to be the name of an executable that is contained in one of the directories in the shell's search path, which is an ordered list of directories that the shell searches for executables.

If the command line ends with an ampersand ("`&`"), then the job runs in the *background*, which means that the shell does not wait for the job to terminate before printing the prompt and awaiting the next command line. Otherwise, the job runs in the *foreground*, which means that the shell waits for the job to terminate before awaiting the next command line. Thus, at any point in time, at most one job can be running in the foreground. However, an arbitrary number of jobs can run in the background.

For example, typing the command line

```
tsh> jobs
```

causes the shell to execute the built-in `jobs` command.

In contrast, typing the command line

```
tsh> /bin/ls -l -d
```

runs the `ls` program in the foreground. By convention, the shell ensures that when the program begins executing its main routine

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

the `argc` and `argv` arguments have the following values:

- `argc == 3`,
- `argv[0] == "/bin/ls"`,
- `argv[1] == "-l"`,
- `argv[2] == "-d"`.

Alternatively, typing the command line

```
tsh> /bin/ls -l -d &
```

runs the `ls` program in the background.

Unix shells support the notion of *job control*, which allows users to move jobs back and forth between the background and foreground, and to change the process state (running, stopped, or terminated) of the processes in a job. Typing `ctrl-c` causes a 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 SIGTSTP is to place a process in the stopped state, where it remains until it is awakened by the receipt of a SIGCONT signal. Unix shells also provide various built-in commands that support job control. For example:

- `jobs`: List the stopped jobs and running background jobs.
- `bg <job>`: Change a stopped job to a running background job.
- `fg <job>`: Change a stopped or running background job to a running job in the foreground.

The Tiny Shell (`tsh`) Specification

Your `tsh` shell should have the following features:

- The prompt should be the string "`tsh> `".
- The search path should be the string contained by the `PATH` environmental variable. The search path is a string that contains a list of directories, where each directory is separated by the "`:`" character. If the search path contains an empty string, this should be interpreted as the current directory being located in the search path. The search path can contain an empty string if the entire path is an empty string, if there are two "`:`" characters in a row, or if the path starts or ends with the "`:`" character. The search path should be loaded when the program is initialized.

- The command line typed by the user should consist of a `name` and zero or more arguments, all separated by one or more spaces. If `name` is a built-in command, then `tsh` should handle it immediately and wait for the next command line. Otherwise, `tsh` should assume that `name` is the name of an executable file, which it loads and runs in the context of an initial child process. This initial child process is called a *job*.

  If `name` does not start with a directory and the search path is not `NULL`, then `tsh` should assume that `name` is the name of an executable file contained in one of the directories in the search path. In this case, `tsh` should search the directories in the search path, in order, for an executable named `name`. The first executable named `name` found in the search path that `tsh` can execute is the program that `tsh` loads and runs. If `name` starts with a directory or the search path is `NULL`, `tsh` should assume that `name` is the path name of an executable file.

- `tsh` must use the `execve` function. In other words, `tsh` may not use any other member of the `exec` family, such as `execvp`.

- `tsh` need not support pipes (`|`) or I/O redirection (< and >).

- Typing `ctrl-c` (`ctrl-z`) should cause a SIGINT (SIGTSTP) signal to be sent to the current foreground job, as well as any descendents of that job (e.g., any child processes that it forked). If there is no foreground job, then the signal should have no effect.

- If the command line ends with an ampersand (`&`), then `tsh` should run the job in the background. Otherwise, it should run the job in the foreground.

- Each job can be identified by either a process ID (PID) or a job ID (JID), which is a positive integer assigned by `tsh`. JIDs should be denoted on the command line by the prefix '`%`'. For example, "`%5`" denotes JID 5, and "`5`" denotes PID 5. (We have provided you with all of the routines you need for manipulating the job list.)

- `tsh` should support the following built-in commands:

  - The `quit` command terminates the shell.
  - The `jobs` command lists the stopped jobs and running background jobs.
  - The `bg <job>` command restarts `<job>` by sending it a SIGCONT signal, and then runs it in the background. The `<job>` argument can be either a PID or a JID.
  - The `fg <job>` command restarts `<job>` by sending it a SIGCONT signal, and then runs it in the foreground. The `<job>` argument can be either a PID or a JID.

- `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 of the form

  ```
  Job [1] (11639) terminated by signal SIGINT
  ```

  Where the job ID is 1 and the process id is 11639 and the job terminated because of a SIGINT signal. The provided `signame` array is useful for converting signal numbers into names. Specifically, use the signal number as the index for selecting an element of the array. The element will be the address of a string containing the symbol for that signal, without the "SIG" prefix.

Getting Started

Please visit the web page at `https://classroom.github.com/a/4QDedgCa`. (If you are copying this URL, do not include the period at the end of the sentence, as it is not part of the URL.) This page should say "RICE-COMP321-S25-classroom" and "Accept the assignment — Shell". Moreover, it should have a green button labeled "Accept this assignment"[1]. Please accept the assignment.

Upon accepting the assignment, you will be redirected to another web page. This page will confirm that you have accepted the assignment, and it will eventually (after you click refresh) provide you with a link to your personal repository for the assignment. Click this link to go to your personal repository.

The web page for your personal repository has a green button labeled "Code". Click this button. You should now see a text field with a URL. Copy or remember this URL.

Login to the `CLEAR9` system if you have not already done so. Type the following:

`git clone` [Copy the URL for your repo here]

You will be prompted for your `github` username and password.

Once the clone operation is complete, you will have a directory named

`shell-`[YOUR `github` ID]

Please `cd` into this directory, and do the following:

- Type the command `make` to compile and link a skeleton shell and some simple programs you can run from within your shell to test it.
- Type your name and NetID in the header comment at the top of `tsh.c`.

Looking at the `tsh.c` (*tiny shell*) file, you will see that it contains a functional skeleton of a simple Unix shell. To help you get started, we have already implemented the less interesting functions. Your assignment is to complete the remaining empty functions listed below. As a sanity check for you, we've listed the approximate number of lines of code for each of these functions in our reference solution (which includes lots of comments).

- `main`: Performs the read-evaluate/execute loop. Most of this function's implementation is provided. The only changes that you should need to make are to the `sigaction` calls. [6-12 lines]
- `initpath`: Performs any necessary initialization of the search path. The complexity of this function affects the complexity of the `eval` function. Specifically, a more complicated `initpath` function may allow for a simpler `eval` function. [1–20 lines]
- `eval`: Main routine that parses and interprets the command line. [90–110 lines]
- `builtin_cmd`: Recognizes and interprets the built-in commands: `quit`, `fg`, `bg`, and `jobs`. [25 lines]
- `do_bgfg`: Implements the `bg` and `fg` built-in commands. [50 lines]
- `waitfg`: Waits for a foreground job to complete. [20 lines]
- `sigchld_handler`: Catches SIGCHILD signals. [80 lines]

[1] You may have to login to GitHub to see this page. If so, you will be prompted for your GitHub username and password.

- `sigint_handler`: Catches SIGINT (`ctrl-c`) signals. [15 lines]
- `sigtstp_handler`: Catches SIGTSTP (`ctrl-z`) signals. [15 lines]

Each time you modify your `tsh.c` file, type `make` to recompile it. To run your shell, type `tsh` to the command line:

```
UNIX% ./tsh
tsh> [type commands to your shell here]
```

Checking Your Work

We have provided some tools to help you check your work.

You may assume that `MAXLINE`, `MAXARGS`, `MAXJOBS`, and `MAXJID` are the maximum number of characters in any command line, the maximum number of arguments on any command line, the maximum number of jobs at any time, and the maximum job ID, respectively. You do not need to check to ensure that these maximums will be met, rather you may assume that no one will type any input into the shell that would cause them to be exceeded.

Reference solution. The executable `tshref` is the reference solution for the shell. Run this program to resolve any questions you have about how your shell should behave. *Your shell should emit output that is identical to the reference solution* (except for PIDs, of course, which change from run to run).

Shell driver. The `sdriver.pl` program executes a shell as a child process, sends it commands and signals as directed by a *trace file*, and captures and displays the output from the shell.

Use the -h argument to find out the usage of `sdriver.pl`:

```
UNIX% ./sdriver.pl -h
Usage: sdriver.pl [-hv] -t <trace> -s <shellprog> -a <args>
Options:
  -h            Print this message
  -v            Be more verbose
  -t <trace>    Trace file
  -s <shell>    Shell program to test
  -a <args>     Shell arguments
  -g            Generate output for autograder
```

We have also provided sample trace files (`trace*.txt`) that you can use in conjunction with the shell driver to test the correctness of your shell. The lower-numbered trace files do very simple tests, and the higher-numbered tests do more complicated tests.

You can run the shell driver on your shell using trace file `trace01.txt` (for instance) by typing:

```
UNIX% ./sdriver.pl -t trace01.txt -s ./tsh -a "-p"
```

(the `-a "-p"` argument tells your shell not to emit a prompt), or

```
UNIX% make trace01.test
```

Similarly, to compare your result with the reference shell, you can run the trace driver on the reference shell by typing:

```
UNIX% ./sdriver.pl -t trace01.txt -s ./tshref -a "-p"
```

or

```
UNIX% make trace01.rtest
```

The neat thing about the trace files is that they generate the same output you would have gotten had you run your shell interactively (except for an initial comment that identifies the trace). For example:

```
UNIX% make trace07.test
./sdriver.pl -t trace07.txt -s ./tsh -a "-p"
#
# trace07.txt - Forward SIGINT only to foreground job.
#
tsh> ./myspin 4 &
[1] (27778) ./myspin 4 &
tsh> ./myspin 5
Job [2] (27780) terminated by signal SIGINT
tsh> jobs
[1] (27778) Running ./myspin 4 &
UNIX%
```

Trace file format. The trace file consists of text lines that are blank lines, comment lines, driver commands, or shell commands. Blank lines are ignored. Comment lines begin with "#" and are echoed without change to stdout. Driver commands are interpreted by the driver and are not passed to the child shell. All other lines are shell commands and are passed without modification to the shell, which reads them on stdin. Output produced by the child on stdout/stderr is read by the parent and printed on its stdout.

The driver commands are:

Command	Description
TSTP	Send a SIGTSTP signal to the child
INT	Send a SIGINT signal to the child
QUIT	Send a SIGQUIT signal to the child
KILL	Send a SIGKILL signal to the child
CLOSE	Close Writer (sends EOF signal to child)
WAIT	Wait() for child to terminate
SLEEP n	Sleep for n seconds

Notes

- Read every word of Chapter 8 (Exceptional Control Flow) in your textbook. **Significant changes were made to this chapter in the third edition of the textbook.** In particular, the material in Section 8.5.5 on safe signal handling did not exist in the previous edition.
- Read every word of this project handout.
- Use the trace files to guide the development of your shell. Starting with `trace01.txt`, make sure that your shell produces the *identical* output as the reference shell. Then move on to trace file `trace02.txt`, and so on.
- The functions `waitpid`, `kill`, `fork`, `execve`, `setpgid`, and `sigprocmask` will be the fundamental building blocks of your shell. Make sure that you understand what these functions do and how to use them. The WUNTRACED and WNOHANG options to `waitpid` will also be useful.
- When you implement your signal handlers, be sure to send `SIGINT` and `SIGTSTP` signals to the entire foreground process group, using "`-pid`" instead of "`pid`" in the argument to the `kill` function. The `sdriver.pl` program tests for this error.

- One of the tricky parts of the assignment is deciding on the allocation of work between the `waitfg` and `sigchld_handler` functions. We recommend the following approach:
  - In `waitfg`, use a loop around the `sigsuspend` function that tests whether or not the specified process is still running in the foreground. (How to use the `sigsuspend` function is described in Section 8.5.7.)
  - In `sigchld_handler`, use `waitpid` to reap zombie children.

  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.

- When you implement your signal handlers, be sure to follow the rules described in Section 8.5.5. In particular, per rule *G1*, you must not call functions such as `printf` or `fprintf` to print messages.

- The textbook is too quick to dismiss the direct use of `sigaction` by applications, like your shell, using the weak argument that having to initialize and pass a structure to `sigaction` makes it "unwieldy" to use. (See page 775.) However, that structure's `sa_mask` field provides for a much simpler approach to addressing the problem discussed in Section 8.5.5's rule *G3*. Rather than explicitly calling `sigprocmask` within a handler, as suggested by rule *G3*, you should use `sigaction`'s `sa_mask` to automatically block and unblock any signals that should not be allowed to preempt the signal 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 which signals are *blocked* by their parents, the child must be sure to then unblock `SIGCHLD` signals before it execs the new program. Moreover, before the child unblocks signals, it should call `signal` to reset the handling of `SIGINT` and `SIGTSTP` to their defaults, for example, `signal(SIGINT, SIG_DFL)`, so that these signals do not result in the execution of their handlers by the child before it calls `execve`.

  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`, `/bin/echo`, and the included test programs (such as `myspin`).

- 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. There should only be one process, your shell, in the foreground process group. When you type `ctrl-c`, your 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).

  Here is the workaround to ensure that your shell is the only process in the foreground process group and that it can reliably forward signals to the foreground job: After the `fork`, but before the call to `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. However, that call by itself is not sufficient.

The (parent) shell and child run concurrently, so there is no guarantee that the child will perform the `setpgid(0, 0)` call before the shell might attempt to forward a signal to the child's process group. To ensure that sending a signal from the shell to the child's process group does not fail because the group does not yet exist, the parent should also call `setpgid(pid, pid)` to set the child's process group, where `pid` is the child's PID. This call should occur in the (parent) shell after the `fork` and before unblocking `SIGCHLD` signals. Similarly, there is no guarantee that the shell will perform the `setpgid(pid, pid)` call before the child runs, calling `execve` to start the new program, so both calls to `setpgid()` are necessary, even though the one that actually occurs second will be redundant.

- As described, your shell will not be able to handle programs that read from the standard input, and you are not required to do so. Note that this means you will not be able to run your shell from within your shell. Handling programs that read from the standard input requires your shell to give control of the terminal to its foreground child, which is beyond the scope of this project. We suggest you stick to the simple text based programs mentioned above and to the provided test programs (such as `myspin`).
- A string is guaranteed to start with a directory if the string contains a "`/`".
- The function `strtol` can be used to convert a string containing decimal digits into a (long) integer. Compared to the related function `atoi`, the function `strtol` better supports error checking on its input.

Testing

You should write test trace files for your shell in addition to the provided ones. Your test trace files **must** be named "traceXX.txt", where XX is replaced by a number starting with 13 (past the 12 provided test trace files).

All of your test trace files should be documented with comment lines clearly stating what the trace is testing. Also, make sure you commit all your test trace files to your repository. You can run your test traces on your shell and on the reference shell as described above. For instance, you can run your trace37.txt file on your shell as follows:

```
UNIX% make trace37.test
```

Turning in Your Assignment

To turn in your code and test trace files, you ***must*** use `git push` to copy your work to the `github` remote repository. As a precaution against accidental loss of your code or test trace files, we encourage you to `push` periodically.

The *only* files that you need to turn in are `tsh.c` and your test trace files (`traceXX.txt`). In other words, these are the only files on which you should ever perform `git add`.

For grading your submission, we will use the `Makefile` that was originally provided to compile your code. Therefore, your code should not rely on any modifications to the `Makefile` for correct compilation.

As a final sanity check, you should use your web browser to visit your assignment repo. Make sure that what you see in the browser is consistent what you think you have pushed.

Evaluation

The project will be graded as follows:

- Testing: 15%
- Style: 15%
- Correctness: 70%

Your solution will be tested for correctness on a CLEAR9 machine, using the same shell driver and trace files that were provided (along with additional traces that were not provided). Your shell should produce **identical** output on these traces, with only two exceptions:

- The PIDs can (and will) be different.
- The output of the `/bin/ps` commands will be different from run to run. However, the running states of any processes run from within your shell in the output of the `/bin/ps` command should be identical.

For each test, you will only receive credit for that test if the output of your shell is identical to the output of the reference shell (other than the exceptions listed above).