15-213/18-213/15-513/14-513, Fall 2018
Shell Lab: Writing Your Own Linux Shell

Assigned: Friday, October 19, 2018 at 12:00 AM EDT
Due: Tuesday, October 30, 2018 at 11:59 PM EDT
Submissions close: Thursday, November 1, 2018 at 11:59 PM EDT

1 Introduction

The purpose of this assignment is to help you become more familiar with the concepts of process control and signalling. You'll do this by writing a simple Linux shell program, `tsh` (tiny shell), that supports a simple form of job control and I/O redirection. Please read the whole writeup before starting.

2 Hand Out Instructions

Create your GitHub Classroom repository at `https://classroom.github.com/a/Qv3Se6Bc`. Then do the following on either an Andrew Linux or a Shark machine:

- Clone the repository that you just created using the `git clone` command. *Do not download and extract the zip file from GitHub.*
- Type your name and Andrew ID in the header comment at the top of `tsh.c`.
- Type the command `make` to compile and link the driver, the trace interpreter, and the test routines.

Looking at the `tsh.c` file, you will see that it contains a skeleton of a simple Linux shell. It will not, of course, function as a shell if you compile and run it now. To help you get started, we've provided you with a helper file, `tsh_helper.{c,h}`, which contains the implementation of routines that manipulate a job list, and a command line parser. Read the header file carefully to understand how to use it in your shell.

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 — that's a good thing).

- `eval`: Main routine that parses, interprets, and executes the command line. [300 lines, including helper functions]
- `sigchld_handler`: Handles `SIGCHLD` signals. [80 lines]
- `sigint_handler`: Handles `SIGINT` signals (sent by `Ctrl-C`). [15 lines]
- `sigtstp_handler`: Handles `SIGTSTP` signals (sent by `Ctrl-Z`). [15 lines]

When you wish to test your shell, type `make` to recompile it. To run it, type `tsh` to the command line:

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

3 General Overview of Linux 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 pathname 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 word is assumed to be the pathname of an executable program. In this case, the shell forks a child process, then loads and runs 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 Linux pipes. However, the shell you write in this lab need not support pipes.

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. Typing the command line

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

runs the `/bin/ls` program in the foreground. By convention, the shell will ensure that when `/bin/ls` 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"
```

On the other hand, typing the command line

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

runs the `/bin/ls` program in the background.

Linux 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. For example,

- 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.

Linux shells also provide various built-in commands that support job control. For example:

- `jobs`: List the running and stopped background jobs.
- `bg` *job*: Change a stopped background job into a running background job.
- `fg` *job*: Change a stopped or running background job into a running foreground job.

Linux shells also support the notion of **I/O redirection**, which allows users to redirect `stdin` and `stdout` to files. For example, typing the command line

```
tsh> /bin/ls > foo
```

redirects the output of `ls` to a file called `foo`. Similarly,

```
tsh> /bin/cat < foo
```

will redirect the input of the `/bin/cat` command to the `foo` command, which will print the contents of the file `foo`.

4 The `tsh` Specification

Your `tsh` shell should have the following features:

- The prompt should be the string "`tsh> `".
- 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.

  Otherwise, `tsh` should assume that `name` is the path of an executable file, which it should load and run in the context of an initial child process. (In this context, the term *job* refers to this initial child process). If you are running system programs like `ls`, you will need to enter the full path (in this case `/bin/ls`), because your shell will not support search paths.
- `tsh` need not support pipes (`|`), but **MUST** support I/O redirection ("<" and ">"). For example:

```
tsh> /bin/cat < foo > bar
```

Your shell must support both input and output redirection in the same command line.

- Typing `Ctrl-C` or `Ctrl-Z` should cause your shell to send a `SIGINT` or `SIGTSTP` signal, respectively, 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.
- Each job can be identified by either a process ID (PID) or a job ID (JID). The latter is a positive integer assigned by `tsh`. JIDs are denoted on the command line with the prefix "`%`". For example, "`%5`" denotes a JID of 5, and "`5`" denotes a PID of 5.
- 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. When starting a background job, `tsh` should print out the command line, prepended with the job ID and the process ID. For example:

  ```
  [1] (32757) /bin/ls &
  ```

- `tsh` should support the following built-in commands:
  - The `quit` command terminates the shell.
  - The `jobs` command lists all 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.
- Your shell should be able to redirect the output from the built-in `jobs` command. For example,

  ```
  tsh> jobs > foo
  ```

  should write the output of `jobs` to the `foo` file. The reference shell supports output redirection for all built-ins, but you are only required to implement it for `jobs`.
- `tsh` should reap all of its zombie children. If any job terminates or stops because it receives a signal that it didn't catch, then `tsh` should recognize that event and print a message with the job's JID and PID, and the offending signal number. For example,

  ```
  Job [1] (1778) terminated by signal 2
  Job [2] (1836) stopped by signal 20
  ```

5 Checking Your Work

Running your shell. The best way to check your work is to run your shell from the command line. Your initial testing should be done manually from the command line. Run your shell, type commands to it, and see if you can break it. Use it to run real programs!

Reference solution. The 64-bit Linux executable `tshref` is the reference solution for the shell. Run this program (on a 64-bit machine) 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, which change from run to run. (See the Evaluation section.)

Once you are confident that your shell is working, then you can begin to use some tools that we have provided to help you check your work more thoroughly. These are the same tools that the autograder will use when you submit your work for credit.

Trace interpreter. We have provided a set of trace files (`trace*.txt`) that validate the correctness of your shell. Each trace file tests a different shell feature. For example, does your shell recognize a particular built-in command? Does it respond correctly to the user typing a `Ctrl-C`?

The `runtrace` program (the trace interpreter) interprets a set of shell commands in a single trace file:

```
linux> ./runtrace -h
Usage: runtrace -f <file> -s <shellprog> [-hV]
Options:
  -h            Print this message
  -s <shell>    Shell program to test (default ./tsh)
  -f <file>     Trace file
  -V            Be more verbose
```

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:

```
linux> ./runtrace -f trace05.txt -s ./tsh
#
# trace05.txt - Run a background job.
#
tsh> ./myspin1 &
[1] (15849) ./myspin1 &
tsh> quit
```

The lower-numbered trace files do very simple tests, while the higher-numbered trace files do increasingly more complicated tests. The appendix contains a description of each of the trace files, as well as each of the commands used in the trace files.

Please note that `runtrace` creates a temporary directory `runtrace.tmp`, which is used to store the output of redirecting commands, and deletes it afterwards. However, if for some reason the directory is not deleted, then `runtrace` will refuse to run. In this case, it may be necessary to delete this directory manually.

Shell driver. After you have used `runtrace` to test your shell on each trace file individually, then you are ready to test your shell with the shell driver. The `sdriver` program uses `runtrace` to run your shell on each trace file, compares its output to the output produced by the reference shell, and displays the `diff` if they differ.

```
linux> ./sdriver -h
Usage: sdriver [-hV] [-s <shell> -t <tracenum> -i <iters>]
Options
        -h           Print this message.
        -i <iters>   Run each trace <iters> times (default 4)
        -s <shell>   Name of test shell (default ./tsh)
        -t <n>       Run trace <n> only (default all)
        -V           Be more verbose.
```

Running the driver without any arguments tests your shell on all of the trace files. To help detect race conditions in your code, the driver runs each trace multiple times. You will need to pass each of the runs to get credit for a particular trace:

```
linux> ./sdriver
Running 3 iters of trace00.txt
1. Running trace00.txt...
2. Running trace00.txt...
3. Running trace00.txt...
Running 3 iters of trace01.txt
1. Running trace01.txt...
2. Running trace01.txt...
3. Running trace01.txt...
Running 3 iters of trace02.txt
1. Running trace02.txt...
2. Running trace02.txt...
3. Running trace02.txt...

...

Running 3 iters of trace30.txt
1. Running trace30.txt...
2. Running trace30.txt...
3. Running trace30.txt...
Running 3 iters of trace31.txt
1. Running trace31.txt...
2. Running trace31.txt...
3. Running trace31.txt...

Summary: 32/32 correct traces
```

Use the optional `-i` argument to control the number of times the driver runs each trace file:

```
linux> ./sdriver -i 1
Running trace00.txt...
Running trace01.txt...
Running trace02.txt...

...

Running trace30.txt...
Running trace31.txt...

Summary: 32/32 correct traces
```

Use the optional `-t` argument to test a single trace file:

```
linux> ./sdriver -t 06
Running trace06.txt...
Success: The test and reference outputs for trace06.txt matched!
```

Use the optional `-V` argument to get more information about the test:

```
linux> ./sdriver -t 06 -V
Running trace06.txt...
Success: The test and reference outputs for trace06.txt matched!
Test output:
```

```
#
# trace06.txt - Run a foreground job and a background job.
#
tsh> ./myspin1 &
[1] (10276) ./myspin1 &
tsh> ./myspin2 1

Reference output:
#
# trace06.txt - Run a foreground job and a background job.
#
tsh> ./myspin1 &
[1] (10285) ./myspin1 &
tsh> ./myspin2 1
```

6 Warnings

- Start early! Leave yourself plenty of time to debug your solution, as subtle problems in your shell are hard to find and fix.
- There are a lot of helpful code snippets in the textbook. It is OK to use them into your program, but make sure you understand *every line of code* that you are using. Please do not build your shell on top of code you do not understand!
- **Race conditions with child processes.** Remember that you cannot make any assumptions about the order of execution of the parent and child after forking. In particular, you cannot assume that the child will still be running when the parent returns from the `fork`. In fact, our driver has code that purposely introduces non-determinism in the order that the parent and child execute after forking.
- **Race conditions in signal handlers.** Remember that signal handlers run concurrently with the program and can interrupt it anywhere, unless you explicitly block the receipt of the signals. The driver uses techniques to amplify the occurrence of incorrect behaviour caused by race conditions.

  Be especially careful about race conditions on the job list. To avoid race conditions, you should block any signals that might cause a signal handler to run any time you access or modify the job list. To enforce this, we have built checking code into the helper functions in `tsh_helper.c` to make sure these signals are blocked.
- Remember that simply passing the tests multiple times does not prove the correctness of your shell. We will deduct correctness points (up to 20 percent!) if there are race conditions in your code, so it is in your best interest to find them before we do, through thorough inspection.
- **Saving/restoring `errno`.** Signal handlers should always properly save/restore the global variable `errno` to ensure that it is not corrupted, as described in Section 8.5.5 of the textbook. The driver checks for this explicitly, and it will print a warning if `errno` has been corrupted.
- **Busy-waiting.** It is forbidden to spin in a tight loop while waiting for a signal (e.g. "`while (1);`"). Doing so is a waste of CPU cycles. Nor is it appropriate to get around this by calling `sleep` inside a tight loop. Instead, you should use the `sigsuspend` function, which will sleep until a signal is received. Refer to the textbook or lecture slides for more information.

- **Reaping child processes.** When children of your shell die, they must be reaped within a bounded amount of time. This means that you should **not** wait for a running foreground process to finish or for a user input to be entered before reaping.

  You should not call `waitpid` in multiple places. This will set you up for many potential race conditions, and will make your shell needlessly complicated.

- **Async-signal-safety.** Many commonly used functions, including `printf`, are not async-signal-safe; i.e., they should not be invoked from within signal handlers. Within your signal handlers, you must ensure that you only call syscalls and library functions that are themselves async-signal-safe.

  For the `printf` function specifically, the CS:APP library provides `sio_printf` as an async-signal-safe replacement, which you may wish to use in your shell. (See Section 8.5.5 in the textbook for information on async-signal-safety, and see the appendix for information about the functions provided by the CS:APP library.)

- Don't use any system calls that manipulate terminal groups (e.g. `tcsetpgrp`), which will break the autograder.

7 Hints

- Read and understand every word of Chapter 8 (Exceptional Control Flow) and Chapter 10 (System-level I/O) in the textbook.

- Read the code in `tsh.c` and `tsh_helper.h` carefully before you start. Understand the high-level control flow; get familiar with the defined global variables and the helper routines. The tsh helper maintains a job list, which you should use, but does not expose its implementation. You should use the provided functions to access the job list.

- Play with your shell by typing in commands directly. Don't make the mistake of running the trace generator and driver immediately. Develop some familiarity and intuition about how your shell works before testing it with the automated tools.

- Only after you have tested your shell directly from the command line and are fairly confident that it is correct should you start testing with `runtrace` and the driver programs.

- Use the trace files to guide the development of your shell. Starting with `trace00.txt`, make sure that your shell produces *identical* output as the reference shell. Then move on to trace file `trace01.txt`, and so on.

- The `waitpid`, `kill`, `fork`, `execve`, `setpgid`, `sigprocmask`, and `sigsuspend` functions will come in handy. The `WUNTRACED` and `WNOHANG` options to `waitpid` will also be useful. Use `man` and your textbook to learn more about each of these functions.

- 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 driver program specifically tests for this error.

- One of the tricky parts of the assignment is deciding on the allocation of work between the `eval` and `sigchld_handler` functions when the shell is waiting for a foreground job to finish.

- In `eval`, the parent must use `sigprocmask` to block `SIGCHLD`, `SIGINT`, and `SIGTSTP` 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 `add_job`. Since children inherit the `blocked` vectors of their parents, the child must be sure to then unblock these signals before it `exec`s the new program. The child should also restore the default handlers for the signals that are ignored by the shell.

  The parent needs to block signals in this way in order to avoid race conditions. For example, the child could be reaped by `sigchld_handler` (and thus removed from the job list) *before* the parent calls `add_job`). Section 8.5.6 explains some possible race conditions and how to avoid them by blocking signals explicitly.

- Programs such as `top`, `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/cat`, `/bin/ls`, `/bin/ps`, and `/bin/echo`.

- When you run your shell from the standard Linux 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 created by your shell. Obviously, this 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 process group that contains the foreground job. [1]

- Your shell needs to handle error conditions appropriately, which depends on the error being handled. For example, if `malloc` fails, then your shell might as well exit; on the other hand, your shell should not exit just because the user entered an invalid filename. (See the section on style grading.)

  An additional, ungraded trace file `trace32.txt` has been provided that tests the error handling of your shell. However, it is not scored, since your shell does not need to output the same error messages as the reference shell. For error messages, you may find the `perror` function useful, which is what the reference shell uses.

- `trace23.txt` uses builtin commands like `bg` and `fg` on job/process ids that are invalid. Your shell should be able to handle this gracefully, and the autograder will expect to see the same error message from your shell as the reference shell. For example, if job id 5 and process id 6 are both invalid, we'd expect this output from your shell:

```
tsh> fg %5
%5: No such job
tsh> fg 6
(6): No such process
```

[1] With a real shell, the kernel will send `SIGINT` or `SIGTSTP` directly to each child process in the terminal foreground process group. The shell manages the membership of this group using the `tcsetpgrp` function, and manages the attributes of the terminal using the `tcsetattr` function. These functions are outside of the scope of the class, and you should not use them, as they will break the autograding scheme.

8 Evaluation

Your score will be computed out of a maximum of 108 points based on the following distribution:

96 Correctness: 32 trace files at 3 pts each. In addition, if your solution passes the traces but is not actually correct (you hacked a way to get it to pass the traces), we will deduct correctness points during our read through of your code.

The most common thing we will be looking for is race conditions that you have simply plastered over, often using the `sleep` call. In general, your code should not have races, even if we remove all `sleep` calls.

12 Style points. We expect you to follow the style guidelines posted on the course website. For example, we expect you to check the return value of EVERY system call and library function, and handle any error conditions appropriately.

We expect you to break up large functions such as `eval` into smaller helper functions, to enhance readability and avoid duplicating code. We also expect you to write good comments. Some advice about commenting:

- Do begin your program file with a descriptive block comment that describes your shell.
- Do begin each routine with a block comment describing its role at a high level.
- Do preface related lines of code with a block comment.
- Do keep your lines within 80 characters.
- Don't simply comment each line.

You should also follow other guidelines of good style, such as using a consistent indenting style (don't mix spaces and tabs!), using descriptive variable names, and grouping logically related blocks of code with whitespace.

Finally, you should ensure that you regularly commit your code *and* push it to your GitHub repository. Some of the style points for this lab will be assigned based on your usage of Git and your commit history.

As for every lab, be sure to review the style guide for this class, which you can find at `https://www.cs.cmu.edu/~213/codeStyle.html`.

Your solution shell will be tested for correctness on a 64-bit machine (the Autolab server), using the same driver and trace files that were included in your handout directory. Your shell should produce **identical** output on these traces as the reference shell, with only two exceptions:

- The PIDs can (and will) be different.
- The output of the `/bin/ps` commands in `trace26.txt` and `trace27.txt` will be different from run to run. However, the running states of any `mysplit` processes in the output of the `/bin/ps` command should be identical.

The driver deals with all of these subtleties when it checks for correctness.

9 Hand In Instructions

- Make sure you have included your name and Andrew ID in the header comment of `tsh.c`.
- Hand in your `tsh.c` file for credit by uploading it to Autolab. You may hand in as often as you like. You will be graded on the **last** version you hand in.
- After you hand in, it takes a minute or two for the driver to run through multiple iterations of each trace file.
- As with all our lab assignments, we'll be using a sophisticated cheat checker. Please don't copy another student's code. Start early, and if you get stuck, come see your instructors for help.

Good luck!

Appendix: Trace Files

The trace driver runs an instance of your shell in a child process and communicates with the shell interactively in a way that mimics the behavior of a user. To test the behavior of your shell, the trace driver reads in trace files that specify shell line commands that are actually sent to the shell, as well as a few special synchronization commands that are interpreted by the driver when handling the shell process. The trace files may also reference a number of shell test programs to perform various functions, and you may refer to the code and comments of these test programs for more information.

The format of the trace files is as follows:

- The comment character is `#`. Everything to the right of it on a line is ignored.
- Each trace file is written so that the output from the shell shows exactly what the user typed. We do this by using the `/bin/echo` program, which not only tests the shell's ability to run programs, but also shows what the user typed. For example:

  ```
  /bin/echo -e tsh\076 ./myspin1 \046
  ```

  Note: `\076` is the octal representation of `>`, and `\046` is the octal representation of `&`. These are special shell metacharacters that need to be escaped in order to be passed to `/bin/echo`. This command will echo the string `tsh> ./myspin1 &`.

- There are also a few special commands which are used to synchronize the job (your shell) and the parent process (the driver) and to send Linux signals from the parent to the job. These are handled in your shell by the wrapper functions in `wrapper.c`.

  A wrapper is a function injected at link time around calls to a function. For instance, where your code calls `fork`, the linker will replace this call with an invocation of `__wrap_fork`, which in turn calls the real `fork` function. Some of those wrappers are configured to signal the driver and resume execution only when signaled.

WAIT	Wait for a sync signal from the job over its synchronizing UNIX domain socket.
SIGNAL	Send a sync signal to the job over its synchronizing UNIX domain socket.
NEXT	Read and print all responses from the shell until you see the next shell prompt. This command is essential for synchronizing with the shell and mimics the way people wait until they see the shell prompt until they type the next string. It also automatically signals the shell when receiving a signal from the shell.
SIGINT	Send a SIGINT signal to the job.
SIGTSTP	Send a SIGTSTP signal to the job.
SHELLSYNC *function*	Sets an environment to indicate that synchronization in *function* is enabled. Currently supported values of *function* are: `kill`, `get_job_pid`, and `waitpid`. See `wrapper.c` for details.
SHELLWAIT	Wait for a wrapper in the shell to signal `runtrace` over the shell synchronizing domain socket.
SHELLSIGNAL	Tell the wrapper to resume execution over the shell synchronizing domain socket.
PID *name* fg/bg	Calls the shell builtin command `fg` or `bg`, passing the PID of the process *name*.

The following table describes what each trace file tests on your shell against the reference solution.

NOTE: this table is provided so that you can quickly get a high level picture about the testing traces. The explanation here is over-simplified. To understand what exactly each trace file does, you need to read the trace files.

trace00.txt	Properly terminate on EOF.
trace01.txt	Process built-in quit command.
trace02.txt	Run a foreground job that prints an environment variable.
trace03.txt	Run a synchronizing foreground job without any arguments.
trace04.txt	Run a foreground job with arguments.
trace05.txt	Run a background job.
trace06.txt	Run a foreground job and a background job.
trace07.txt	Use the jobs built-in command.
trace08.txt	Check that the shell can correctly handle reaping multiple process
trace09.txt	Send fatal SIGINT to foreground job.
trace10.txt	Send SIGTSTP to foreground job.
trace11.txt	Send fatal SIGTERM (15) to a background job.
trace12.txt	Child sends SIGINT to itself.
trace13.txt	Child sends SIGTSTP to itself.
trace14.txt	Run a background job that kills itself
trace15.txt	Forward SIGINT to foreground job only.
trace16.txt	Forward SIGTSTP to foreground job only.
trace17.txt	Exit the child in the middle of sigint/sigtsp handler
trace18.txt	Signal a job right after it has been reaped.
trace19.txt	Forward signal to process with surprising signal handlers.
trace20.txt	Process bg built-in command (one job).
trace21.txt	Process bg built-in command (two jobs).
trace22.txt	Check that the fg command waits for the program to finish.
trace23.txt	Process fg builtin command (many jobs, with PID and JID, test error message)
trace24.txt	Signal and end a background job in the middle of a fg command
trace25.txt	Forward SIGINT to every process in foreground process group.
trace26.txt	Forward SIGTSTP to every process in foreground process group.
trace27.txt	Restart every stopped process in process group.
trace28.txt	I/O redirection (input).
trace29.txt	I/O redirection (output)
trace30.txt	I/O redirection (input and output).
trace31.txt	I/O redirection (input and output, different order, permissions)
trace32.txt	Error handling (not scored on Autolab)

NOTE: `trace32.txt` will not be graded. It's been included anyway as it may be useful to think about some of the error cases that can be easy to overlook while implementing the rest of the shell.

Appendix: CS:APP library

The `csapp.c` file contains a number of useful functions described in the textbook. This code will be linked with your code, and so you can make use of any of these functions.

One useful type of function in `csapp.c` is wrapped versions of many of the functions you will use. These are named with an upper-case character as the first letter. For example, `Fork` is the wrapped version of the `fork` syscall. A wrapped function calls its core function and checks for errors. If any error is detected, it prints an error message and exits the program.

This exit-on-failure behavior is acceptable for some, *but not all*, of the code you will be writing. In particular, if running a certain command in your shell fails, then that should not terminate your entire shell. You must decide when it is appropriate to use the wrapped functions, and make sure to handle errors appropriately when necessary.

Another use of `csapp.c` is that it provides the SIO series of functions, which are async-signal-safe functions you can use to print output. The main function of interest is the `sio_printf` function that you can use to print formatted output, which you can use the same way you use the `printf` function. However, it only implements a subset of the format strings, which are as follows:

- Integer formats: `%d`, `%i`, `%u`, `%x`, `%o`, with optional size specifiers `l` or `z`
- Other formats: `%c`, `%s`, `%%`, `%p`

For this lab, we have **removed** the `sio_puts` and `sio_putl` functions that are used in the textbook. Instead, we encourage you to use the `sio_printf` family of functions for async-signal-safe I/O, which should help you write more readable code.