15-213/15-513, Summer 2024
Shell Lab: Writing Your Own Linux Shell

Assigned: Tues, Jul 9
Due: Fri, Jul 19, 11:59PM
Last Possible Handin: Mon, Jul 22, 11:59PM

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 Logistics

This is an individual project. All handins are electronic. You must do this lab assignment on a class shark machine.

To get your lab materials, click "Download Handout" on Autolab. Clone your repository *on a Shark machine* via HTTPS by running:

```
shark> git clone https://github.com/cmu15213-s24/tshlab-s24-<USERNAME>.git
```

or via SSH by running:

```
shark> git clone git@github.com:cmu15213-s24/tshlab-s24-<USERNAME>.git
```

3 Overview

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.

- `eval`: Main routine that parses, interprets, and executes the command line.
- `sigchld_handler`: Handles `SIGCHLD` signals.
- `sigint_handler`: Handles `SIGINT` signals (sent by `Ctrl-C`).

- `sigtstp_handler`: Handles `SIGTSTP` signals (sent by `Ctrl-Z`).

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

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

4 General Guidelines for Writing Your Shell

This section provides an overview of how you can start writing your shell. You should read Section 4: The `tsh` Specification, for a list of everything your shell should support and the format of all shell output.

- 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.

  Each command consists of one or more words, the first of which is the name of an action to perform. This may either be the path to an executable file (e.g., `tsh> /bin/ls`), or a **built-in command**—a word with special meaning to the shell—(e.g., `tsh> quit`). Following this are command-line arguments to be passed to the command.

- Built-in commands run within the shell's process. Looking at the handout code, you may notice that it's difficult to exit the program. Try making it respond to the word `quit`.

- So as not to corrupt its own state, the shell runs each executable in its own child process. You should recall from lecture the sequence of three library calls necessary to create a new process, run a particular executable, and wait for a child process to end. Try to make your shell correctly respond to `/bin/ls`, without breaking the existing `quit` command. If this works, try passing `ls` a particular directory to make sure your shell is passing the arguments along.

- The child processes created as a result of interpreting a single command line are known collectively as a **job**. We just saw one type of job, a **foreground job**. However, sometimes a user wants to do more than one thing at once: in this case, they can instruct the shell *not* to wait for a command to terminate by instead running it as a **background job**. Looking back at the sequence of calls you made to implement foreground jobs, what do you think you would do differently to spawn a background job?

- Given that your shell will need to support both types of job, consider refactoring your existing code to minimize the amount of duplication that will be necessary.

- Try implementing the execution of background jobs, which your shell should do whenever the command line ends with an `&` character. To test this feature, try executing `tsh> /usr/bin/sleep 5` and comparing against `tsh> /usr/bin/sleep 5 &`. In the latter case, the command prompt should appear immediately after running the command. Now you can run multiple sleeps at once!

- 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. The `sigchld_handler` might be a good place to reap **all** your child processes.

- The shell might want to track in-flight jobs and provide an interface for switching their status i.e. background to foreground, etc. Now might be a good time to read the api in `tsh_helper.{c,h}` and start maintaining a job list.

- Typing `Ctrl-C` or `Ctrl-Z` causes a `SIGINT` or `SIGTSTP` signal, respectively. Your shell should catch the signals and forward them to the *entire* process group that contains the foreground job. If there is no foreground job, then these signals should have no effect.

- 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 a possible workaround: After the `fork`, but before the `execve`, you may want to think of a way to put the child in a new process group whose group ID is identical to the child's PID. This would ensure that there will be only one process, your shell, in the foreground process group. Hint: `man setpgid`. [1]

- Remember that signal handlers run concurrently with the program and can interrupt it anywhere, unless you explicitly block the receipt of the signals. Be very 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.

Aside from these guidelines, you should use the trace files to guide the development of your shell. The trace files are in order of difficulty so it might not be the best to attempt a trace before passing all traces up to it.

5 Incremental Development

In this section, we describe in more detail about the traces and give you a mental roadmap as to how you can approach the lab.

Trace 0 should work out of the box. trace 1 is about the builtin command `quit`. Traces 2-4 are about creating a foreground job with possibly different flavors for arguments or environment variables. Fundamentally, you should be able to create a child process, pass on the arguments and the environment variables to it. Recall from lecture the `fork+exec` model to create the child process. How do you reap a child after it's done running? You could use the obvious method to wait for the foreground job and then reap it, but you will have to change it since there could be simultaneous foreground and background jobs. Nevertheless, it is a good starting point to get everything working till trace 4.

Starting in trace 5, there are "additional" foreground jobs running with `bin/echo` that print messages out to the terminal. At this point, you need to have an understanding for what a foreground job means, how is it different from a background job means, and how your shell can differentiate between them. Reading our helper code would be helpful.

Remember the part where we described simultaneous foreground and background jobs? Well here is it, trace 5. This is tricky, so let's break it down into simpler pieces. Whenever a child process terminates, the kernel sends a `SIGCHLD` signal to the parent, which triggers the appropriate signal handler. However, when simultaneous jobs are running and terminating, multiple `SIGCHLD` signals would be **coalesced**. Therefore, it

[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.

is important to reap **all** child processes which have terminated while handling the `SIGCHLD` signal, specifically trace 6 and trace 8 test if your shell is capable of doing this.

How would you go about doing this? Think about where would you reap the children, as described above, `SIGCHLD` handler is a good place to do it. You would need to make sure that **all** exited child processes are reaped, some flavor of `waitpid` will help you doing this. One more question that you might wonder is if `waitpid` is moved into the `SIGCHLD` handler, how would you wait for the foreground job within `eval`? `sigsuspend` and certain functions from the job interface should be helpful in achieving this. Reading man-pages and the helper functions from job interface is highly recommended.

You also have to update the job interface to add/delete jobs or change their status. It is critical that the job list is protected against possible race conditions. To ensure this, you can block for the 3 signals (`SIGCHLD, SIGINT, SIGTSTP`) in the handlers. As an exercise to the reader, figure out why these 3 signals are required to block while accessing the job list.

Traces 9-21 handle signals. You would have to implement the `SIGINT` and `SIGTSTP` handlers. It is also critical to understand the behavior of `execve` – it resets the signal handlers to their default actions in the child processes. This should make sense to you if you think about what code is running in the child process when `execve` succeeds. Additionally, use `tshref` to understand the output expected and replicate it while you reap child processes in your shell. For `SIGTSTP`, read the man pages of `waitpid` to understand how can the shell be informed about a "stopped" process (one which has not terminated) using `SIGCHLD`.

Traces 17-18 test if your shell forwards the signals to all processes in the same process group as the running job. This would require setting the process group id of the child before `execve` is called upon (Hint : `man setpgid`). When forwarding signals, think about what it means when your shell is running. It is a child process of another shell, the linux shell. Does it's status change when receiving signals? Traces 19-21 are checks to ensure your shell is able to handle race conditions and edge cases.

Traces 22-25 are for builtins `fg` and `bg`. You need to parse and extract the job id (either directly or through the process id) and change the state of the corresponding jobs in the job list (under signal safety to avoid races). It might also be insightful to think what these commands do in terms of changing the shell's local view and also the view of the OS w.r.t. the pid/jid. Traces 26 and 27 also stress test your shell to see if it is functional under various race conditions involving the builtins and the command line inputs.

Traces 28-32 involves I/O redirection and error handling. Recall that after forking a child process, you can change it's file descriptors (Hint : `dup` and `dup2` would be useful). Do not leak file descriptors and make sure you do not edit the shell's file descriptors otherwise you might stop getting output on your terminal.

Finally, for error handling, you can extract the error from the `errno` variable whenever a system call fails. You can get information from the man-pages of the system calls for finding these error conditions. Remember that errors are "handled" in three ways: silent ignore, notify the user, fail / abort / exit. In your shell, you may have instances of each. Running the `tshref` on the traces to figure out the required error handling mechanism is recommended.

6 The `tsh` Specification

Your `tsh` shell should have the following features:

- 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.

- `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 resumes *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 resumes `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.
- 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 &
```

- Your shell should be able to handle `SIGINT` and `SIGTSTP` appropriately. If there is no foreground job, then these signals should have no effect.
- `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
```

- `tsh` should support I/O redirection (See Appendix C for more details). For example:

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

  Your shell must support both input and output redirection in the same command line.
- 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`.
- Your shell **does not** need to support pipes.

7 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:

```
shark> ./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:

```
shark> ./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, and compares its output to the output produced by the reference shell.

```
shark> ./sdriver -h
Usage: sdriver [options] [trace [trace...]
Test driver for Shell lab. Run this program to test your shell the
same way Autolab will.
...
```

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 *all* of the runs to get credit for a particular trace:

```
shark> ./sdriver
Running 3 iters of 'traces/trace00.txt'
Iteration 1/3...
```

```
Iteration 2/3...
Iteration 3/3...
Running 3 iters of 'traces/trace01.txt'
Iteration 1/3...
Iteration 2/3...
Iteration 3/3...

...

Running 3 iters of 'traces/trace32.txt'
Iteration 1/3...
Iteration 2/3...
Iteration 3/3...

Summary: 32/32 correct traces
```

To test just one trace file, or several (but still not all of them), list their numbers on the command line:

```
shark> ./sdriver 6
Running 'traces/trace06.txt'...

Summary: 1/1 correct iterations
```

You can also give file names of traces to run. This is convenient for running traces you wrote yourself. File names are relative to the current working directory, *not* the traces directory—if you want to run trace 12 by name you must use `traces/trace12.txt` and not just `trace12.txt`.

You can use the `-i` option to control the number of times the driver runs each trace file:

```
shark> ./sdriver -i 1
Running 'traces/trace00.txt'...
Running 'traces/trace01.txt'...
Running 'traces/trace02.txt'...

...

Running 'traces/trace32.txt'...

Summary: 32/32 correct traces
```

If a test fails, sdriver will print out a comparison between what your shell does for that test, and what the reference shell (`tshref`) does. Here's what that comparison looks like, for an extra-buggy shell that doesn't even pass trace 2:

```
shark> ./sdriver 02
Running 'traces/trace02.txt'...
FAIL: Test and reference outputs for traces/trace02.txt differed.
Comparison of output follows.

    Lines like this were output by both shells.
-ref- Lines like this were output only by the reference shell.
+you+ Lines like this were output only by your shell.
```

```
-ref- ./runtrace -s ./tshref -f traces/trace02.txt
+you+ ./runtrace -s ./tsh -f traces/trace02.txt

      #
      # trace02.txt - Run a foreground job that prints an environment variable
      #
      # IMPORTANT: You must pass this trace before attempting any later
      # traces. In order to synchronize with your child jobs, the driver
      # relies on your shell properly setting the environment.
      MY_ENV=42
+you+ traces/trace02.txt: Runtrace timed out waiting for next shell prompt.
+you+ Child shell still running.

Summary: 0/1 correct iterations
```

If possible, the comparison will be color-coded to emphasize the exact differences between the two results.

`sdriver` has several more options; see its help output (`./sdriver -h`) for details.

8 Hints

- 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!
- Read the manual pages for all system calls that you make. Be sure to understand what their arguments and return/error values are.
- **Signal Blocking and Unblocking.** Child processes inherit the `blocked` vectors and handlers of their parents, so the child must be sure to then unblock any signals before it `exec`s the new program, and also restore the default handlers for the signals that are ignored by the shell.
- **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.** 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. The `WUNTRACED` and `WNOHANG` options to `waitpid` will also be useful. Use `man` and your textbook to learn more about each of these functions.
- **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.
- **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.)

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

9 Evaluation

Your score will be computed out of a maximum of 103 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, or there are race conditions), we will deduct correctness points (up to 20 percent!) 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.

4 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 system calls and library functions, and handle any error conditions appropriately (see Appendix B for exemptions).

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 comment every single line of code.

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.

Your solution shell will be tested for correctness on a 64-bit shark 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.

10 Hand In Instructions

To receive a score, you will need to upload your submission to Autolab. The Autolab servers will run the same driver program that is provided to you. There are two ways you can submit your code to Autolab.

1. Running the `make` command will generate a tar file, `tshlab-handin.tar`. You can upload this file to the Autolab website.
2. If you are running on the Shark machines, you can submit from the command line by typing:

   ```
   $ make submit
   ```

Keep in mind the following:

- You may handin as often as you like until the due date. However, you will only 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.
- Do not assume your submission will succeed! You should ALWAYS check that you received the expected score on Autolab. You can also check if there were any problems in the autograder output, which you can see by clicking on your autograded score in blue.
- 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 A: 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	Child sends SIGINT to itself.
trace10.txt	Child sends SIGTSTP to itself. (known to be buggy, removed)
trace11.txt	Run a background job that kills itself
trace12.txt	Send fatal SIGINT to foreground job.
trace13.txt	Send SIGTSTP to foreground job.
trace14.txt	Send fatal SIGTERM (15) to a background job.
trace15.txt	Forward SIGINT to foreground job only.
trace16.txt	Forward SIGTSTP to foreground job only.
trace17.txt	Forward SIGINT to every process in foreground process group.
trace18.txt	Forward SIGTSTP to every process in foreground process group.
trace19.txt	Exit the child in the middle of sigint/sigtstp handler
trace20.txt	Signal a job right after it has been reaped.
trace21.txt	Forward signal to process with surprising signal handlers.
trace22.txt	Process bg built-in command (one job).
trace23.txt	Process bg built-in command (two jobs).
trace24.txt	Check that the fg command waits for the program to finish.
trace25.txt	Process fg builtin command (many jobs, with PID and JID, test error message)
trace26.txt	Signal and end a background job in the middle of a fg command
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

Appendix B: CS:APP library and Error handling

B.1 CS:APP library

The `csapp.c` provides the SIO series of functions, which are async-signal-safe functions you can use to print output. This code will be linked with your code, and so you can make use of any of these functions. 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.

B.2 Error handling

Using wrapper functions to handle errors can be useful. However, in systems programming, *abruptly exiting the program is rarely the right way to handle errors*. For example, even if your shell is unable to start new processes, it should still continue to run so that the user's existing background jobs can be managed.

For this reason, we have **removed** all of the "Stevens-style wrapper functions" used in the CS:APP textbook. While you are welcome to write your own, we strongly discourage doing so, as opposed to thinking carefully about how to handle each error on an individual basis.

We expect you to check for and appropriately handle errors for any system calls or library functions that you invoke. However, you do not need to check for error for the following calls (you can assume they always succeed):

```
getpgid, getpid, getppid, sigaddset, sigdelset, sigemptyset, sigfillset,
sigismember, sigprocmask, setpgid, sigsuspend
```

B.3 errno

System calls and library functions generally indicate the *presence* of an error by their return value. For example, `fork()` returns `-1` on error, and `malloc()` returns `NULL` on error.

However, many of these functions can also return information about the *type of error* that was encountered through the "global variable" `errno` (see `man errno` for more information). The types of errors that a function can return are documented in its man page. For instance, `man fork` shows that `ENOMEM` is one of the errors that can be returned by `fork()`.

When handling errors, you should use the `perror` or `strerror` functions, which provide user-readable strings for `errno` values.

Appendix C: Unix I/O Redirection

The conventional Unix shell accepts inputs provided from a keyboard and displays outputs to the terminal window. In particular, we refer to the keyboard input as `stdin` and the output to the terminal window as `stdout`. In many cases we may wish to alter the source of the input or output of our commands. This can be done through I/O redirection.

Standard Output

By default, a Unix shell will display output content to the terminal as defined by `stdout`. In the event that we want to change the output destination, we can redirect `stdout` to another location such as a file using the ">" character. For example:

```
tsh> ls > dir.txt
```

should write the output of the `ls` command to the file `dir.txt`.

Standard Input

Similarly, a Unix shell will read input content from the keyboard as defined by `stdin`. In the event that we want to change the input source, we can redirect `stdin` from another location such as a file using the "<" character. For example:

```
tsh> grep < foo.txt -i bar
```

should read the input of the file `foo.txt` into the `grep` command.