Assignment 3: `tsh` – the tiny shell

Kudos to Randal Bryant and Dave O'Hallaron of Carnegie Mellon for this awesome assignment (and most of this handout.)

You've all been using shells to drive your conversations with UNIX system since the first day you logged into a `myth` (and maybe even before that). It's high time we uncover the shell's magic by leveraging what we've built together in lecture and extending it to support process control, job lists, signals, and I/O redirection—all while managing the interprocess concurrency problems that make the shell's implementation a genuinely advanced systems programming project. There's lots of neat code to write, and with your smarts and my love to guide you, I'm confident you can pull it off.

Due Date: Monday, October 27, 2014 at 11:59 p.m.

Getting started

All coding should be done on a `myth` cluster machine, as that's where we'll be testing all `assign3` submissions. You should clone the master mercurial repository we've set up for you by typing:

```
hg clone /usr/class/cs110/repos/assign3/$USER assign3
```

Doing so will create an `assign3` directory within your own file space, and you can descend into your local `assign3` directory and code there.

Look at the `tsh.c` file. You'll see it contains a functional skeleton of a simple Unix shell. To help you get started, we have already implemented the less interesting functions. For the most part, your assignment is to provide full implementations for the functions listed below.

- `eval`: Main routine that parses and interprets the command line.
- `handleBuiltin`: Recognizes and interprets the built-in commands: `quit`, `fg`, `bg`, and `jobs`.
- `handleBackgroundForegroundBuiltin`: Implements the `bg` and `fg` built-in commands.
- `waitfg`: Waits for a foreground job to complete.
- `handleSIGCHLD`: Catches `SIGCHILD` signals.
- `handleSIGINT`: Catches `SIGINT` (ctrl-c) signals.
- `handleSIGTSTP`: Catches `SIGTSTP` (ctrl-z) signals.

You'll notice other header and implementation files: `tsh-constants.h`, `tsh-jobs.[hc]`, `tsh-parse.[hc]`, `tsh-signal.[hc]`, `tsh-state.[hc]`. You will probably only need to modify `tsh.c` and `tsh-jobs.[hc]`. However, you certainly need to understand the code in the other modules.

Of course, each time you modify any of your files, type `make` to recompile. To run your shell, type `tsh` to the command line:

```
myth22> ./tsh
tsh> [type commands to your shell here]
```

General Overview of Unix Shells

A shell is an interactive command-line interpreter that runs programs on a user's behalf. It repeatedly posts a prompt, waits for a command to be typed and published through `stdin`, and then executes that command (provided it's legit).

The command line is a sequence of ASCII text tokens delimited by whitespace. The first token is either the name of a built-in (e.g. `jobs`, `fg`, `bg`, or `quit`) or the pathname of an executable. The remaining tokens are command-line arguments. If the first token is a built-in, the shell immediately executes it inline, within its own process space. Otherwise, the token is assumed to be the pathname of an executable. In that case, the shell `fork`s a child process and then loads and runs the program in the context of that child. The child process created as a result of interpreting a single command line are collectively known as a **job**.

If the command line ends with an ampersand (that is, this: `&`), then the job runs in the background. That means that the shell does not wait for the job to terminate before printing the prompt and awaiting the next command. Otherwise, the job runs in the foreground, which means that the shell waits for the job to exit before accepting any more commands. At any one point, at most one job can be running in the foreground. However, an arbitrary number of jobs can be running in the background (although our implementation artificially limits the total number of jobs to 16).

For example, typing the command line

```
tsh> jobs
```

prompts the shell to execute the built-in `jobs` command. Typing:

```
tsh> ls -lta
```

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

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

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

- `argc == 2`,

- `argv[0] == "ls"`,
- `argv[1]== "-lta"`,
- `argv[2]== NULL`.

Alternatively, typing the command line

```
tsh> ls -lta &
```

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 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. 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. The default action for `SIGTSTP` is to place a process in the stopped state, where it remains until it is awakened by receipt of a `SIGCONT` signal.

Unix shells also provide various built-in commands to support job control. There are the ones you need to support:

- `jobs`: List the running and stopped background jobs.
- bg <job>: Change a stopped background job to a running background job.
- fg <job>: Change a stopped or running background job to a running job in the foreground.
- `quit`: terminate the shell without waiting for any background processes to finish.

Unix shells also support the notion of I/O redirection, which allows users to redirect `stdin` and `stdout` to files on disk. 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
```

publishes the contents of `foo` to `stdout`.

The `tsh` Specification

Your `tsh` shell should support the following:

- The prompt should be the string "tsh> ". (Don't change this)
- 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 the leading token is a built-in, then tsh should execute it to completion before presenting the prompt again. Otherwise, tsh should assume the leading token identifies an executable file, which it loads and runs in the context of an initial child process.
- tsh need not support pipes (|), but it must support I/O redirection (< and >), as with:

```
tsh> ./mycat < foo > bar
```

In particular, your shell must support input and output redirection in the same command line. (Note: you should leave this until last, as it'll necessitate you change the signatures of several functions across some of the modules we've provided to accommodate additional [file descriptor] parameters).

- Typing **ctrl-c** (**ctrl-z**) should fire a SIGINT (SIGTSTP) signal at the current foreground job, as well as any of that job's descendents (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 in the tsh-jobs module).
- tsh should support the following commands as built-ins:
    - 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.
- 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 and print a message with the job's PID and a description of the offending signal.

Checking your work

Because we're nice people, we're providing you with a framework to exercise your implementation.

Reference Solution

We've supplied a reference solution within /usr/class/cs110/samples/assign3. Run this program to resolve any questions you have about how your shell should behave. Your shell should produce output that is identical to the reference solution (except for PIDs, of course—they'll change with each test 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.

Type sdriver.pl -h to learn how to use it:

```
myth22> sdriver.pl -h

Usage:  sdriver.pl [-hv] -t <trace> -s <shellprog> -a <args>
```

Options:

- h Print this message
- v Log information about execution
- t <trace> Trace file
- s <shell> Shell program to test
- a <args> Shell arguments

We've also provided 19 trace files (trace{01-19}.txt) that you can feed to the driver to test your own shell. The lower-numbered trace files do very simple tests, and the higher-numbered ones are more advanced.

For instance, run the shell driver on your own shell using trace file trace01.txt by typing this:

```
myth22> ./sdriver.pl -t traces/trace01.txt -s ./tsh -a "-p"
```

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

```
myth22> make test01
```

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

```
myth22> ln -s /usr/class/cs110/samples/assign3/tsh_soln tsh_soln

myth22> ./sdriver.pl -t traces/trace01.txt -s ./tsh_soln -a "-p"
```

`tsh_soln.out` (in `/usr/class/cs110/samples/assign3`) houses the output of the sample shell on all traces.

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 file). For example:

```
myth22> make test15
./sdriver.pl -t traces/trace15.txt -s ./tsh -a "-p"
#
# trace15.txt - Putting it all together
#
tsh> ./bogus
./bogus: Command not found.
tsh> ./myspin 10
Job (9721) terminated by signal 2
tsh> ./myspin 3 &
[1] (9723) ./myspin 3 &
tsh> ./myspin 4 &
[2] (9725) ./myspin 4 &
tsh> jobs
[1] (9723) Running ./myspin 3 &
[2] (9725) Running ./myspin 4 &
tsh> fg %1
Job [1] (9723) stopped by signal 20
tsh> jobs
[1] (9723) Stopped ./myspin 3 &
[2] (9725) Running ./myspin 4 &
tsh> bg %3
%3: No such job
tsh> bg %1
[1] (9723) ./myspin 3 &
tsh> jobs
[1] (9723) Running ./myspin 3 &
```

```
[2] (9725) Running ./myspin 4 &
tsh> fg %1
tsh> quit
myth22>
```

Your process ID's will be different, but otherwise your output should be exactly the same.

Helpful Hints

- Read every word of Chapters 8 and 10 in your B&O reader.
- Use the trace files to guide development. Start with `trace01.txt` and make sure your shell produces the same output as the sample solution. Then move on to trace file `trace02.txt`, and so on.
- The `waitpid`, `kill`, `fork`, `execvp`, `setpgid`, and `sigprocmask` functions are your friends. The `WUNTRACED` and `WNOHANG` options to `waitpid` are also relevant, so made sure to read up on them you understand what they do for you.
- As you implement your handlers, be sure to forward `SIGINT` and `SIGTSTP` signals to the foreground process group, using `-pid` instead of `pid` in the argument to the `kill` function. You might think that `pid` would work, and very often it might, but it won't kill off any auxiliary processes the foreground process itself forked off.
- One of the tricky parts of the assignment is deciding how to split code between the `waitfg` and `handleSIGCHLD` functions. We recommend the following:
  - In `waitfg`, busy loop around the `sleep` function. If you're really feeling ambitious, you can read up on the `sigsuspend` function and use that instead. Busy waiting in a loop is normally a big no-no, but you're early enough in your multiprocessing careers that I'll let it go this first time.
  - In `handleSIGCHLD`, make exactly one call to `waitpid` in a `while` loop.

While there are other solutions, such as calling `waitpid` in both `waitfg` and `handleSIGCHLD`, doing so 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 `fork`s the child, and then unblock these signals, again using `sigprocmask` after it adds the child to the job list. Since children inherit the blocked vectors from their parents, the child must unblock `SIGCHLD` signals before it `execvp`s 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 `handleSIGCHLD` (and thus removed from the job list) before the parent gets to add it to the job list. You've seen this very thing simulated in a lecture example.

- Programs such as `more`, `less`, `vi`, and `emacs` do strange things with the terminal settings. Don't run these programs from your shell (or at least don't expect them to work). Stick with simple programs such as `ls`, `ps`, and `echo`.
- When you run your `tsh` shell from the standard Unix shell, your own shell is running in the foreground. If your shell then creates a child process, by default that child will also be a member of the foreground process group as well, and you don't want that. 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 solution (pretty much as documented in the text): After the `fork`, but before the `execvp`, 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 `tsh` shell—in the foreground process group. When you type **ctrl-c**, the shell should catch the `SIGINT` and then forward it to the appropriate foreground job (or more precisely, the process group that contains the foreground job).
- Investigate the `dprintf` function, which is like `fprintf`, except that you print to a file descriptor instead of a `FILE *`. Type `man dprintf` at the command prompt to get more information.

Evaluation

Your solution shell will be tested for correctness on a `myth` machine, using the same shell driver and trace files that were included in your `assign3` directory. Your shell should produce identical output on these traces as the reference shell, with two exceptions:

- The PIDs will be different. There's no way to change that.
- The output of the `/bin/ps` command in `trace11.txt`, `trace12.txt`, and `trace13.txt` will be different—possibly very different—from run to run. However, the running states of any `mysplit` processes in the output should be identical.

In addition to the traces we supply, we'll also test your shell against many other inputs. In particular, we'll also be testing your submission's ability to handle redirection and to handle single-quote-delimited arguments (which you should be able to infer from reading through the code we've supplied for `parseLine` in `tsh-parse.[hc]`).

Sanity Checking

I've enabled the sanity check tool used in CS107 to also work on this assignment. Rather than requiring you to type in make tests and then visually diffing against `tsh_soln.out`, you can run `/usr/class/cs110/tools/sanitycheck` for it to run most of these tests for you. (The `sanitycheck` tool needs to omit those traces involving `/bin/ps`, because it's nearly impossible to reliably expect the same output every time on shared machines. You can manually manage those tests on your own).

Submitting your work

Once you're done, you should `hg commit` all of your work as you normally would and then run the famous submissions script by typing in `/usr/class/cs110/tools/submit`.