HW1

Sunday, April 14, 2019 8:31 AM

hw1

HW 1: Shell

CS 162

Due: September 17, 2018

Contents


```
/* Convenience macro
#define unused __att

/* Whether the shell
bool shell_is_intera

/* File descriptor f
int shell_terminal;

/* Terminal mode set
struct termios shell

/* Process group id
pid_t shell_pgid;

int cmd_exit(struct
int cmd_help(struct

/* Built-in command
typedef int cmd_fun_

/* Built-in command
typedef struct fun_c
```

```
 to silence compiler warnings about unused function parameters. */
tribute__((unused))

 is connected to an actual terminal or not. */
active;

for the shell input */


tings for the shell */
_tmodes;

 for the shell */


tokens *tokens);
tokens *tokens);

 functions take token array (see parse.h) and return int */
t(struct tokens *tokens);

 struct and lookup table */
esc {···
```

API for OS, typically a CLI

In this homework, you'll be building a shell, similar to the Bash shell you use on your CS 162 Virtual Machine. The purpose of a shell is to allow users to run and manage programs. The operating system kernel provides well-documented interfaces for building shells. By building your own shell, you'll become more familiar with these interfaces and you'll probably learn more about other shells as well. This assignment will be due **September 17, 2018 at 11:59 pm PST**.

1 Getting started

Log in to your Vagrant Virtual Machine and run:

```
$ cd ~/code/personal/
$ git pull staff master
$ cd hw1
```

We have added starter code for your shell and a simple Makefile in the `hw1` directory. It includes a string tokenizer, which splits a string into words. In order to run the shell:

```
$ make
$ ./shell
```

In order to terminate the shell after it starts, either type `exit` or press `CTRL-D`.

2 Add support for cd and pwd

Out of the box utilities ready to go,

The skeleton code for your shell has a **dispatcher** for "built-in" commands. Every shell needs to support a number of built-in commands, which are functions in the shell itself, not external programs. For example, the `exit` command needs to be implemented as a built-in command, because it exits the shell itself. So far, the only two built-ins supported are `?`, which brings up the help menu, and `exit`, which exits the shell.

Add a new built-in `pwd` that prints the current working directory to standard output. Then, add a new built-in `cd` that takes one argument, a directory path, and changes the current working directory to that directory.

Once you're done, push your code to the autograder. In your VM:

```
$ git add shell.c
$ git commit -m "Finished adding basic functionality into the shell."
$ git push personal master
```

You should commit your code periodically and often so you can go back to a previous version of your code if you want to.

3 Program execution

If you try to type something into your shell that isn't a built-in command, you'll get a message that the shell doesn't know how to execute programs. Modify your shell so that it can execute programs when they are entered into the shell. The first word of the command is the name of the program. The rest of the words are the command-line arguments to the program.

For this step, you can assume that the first word of the command will be **the full path to the program**. So instead of running `wc`, you would have to run `/usr/bin/wc`. In the next section, you will implement support for simple program names like `wc`. But you can pass some autograder tests by only supporting full paths.

cd

You should use the functions defined in `tokenizer.c` for separating the input text into words. You do not need to support any parsing features that are not supported by `tokenizer.c`. Once you implement this step, you should be able to execute programs like this:

```
t(struct tokens *tokens);

 struct and lookup table */
lesc {···

e[] = {···

 description for the given command */
 struct tokens *tokens) {···

 */
 struct tokens *tokens) {···

t-in command, if it exists. */
]) {···

ocedures for this shell */
···

 argc, unused char *argv[]) {···
```

nds regarding the shell itself

```
nce

ct that represents a list of words. */
kens;

 string into a list of words. */
kens *tokenize(const char *line);

ny words are there? */
kens_get_length(struct tokens *tokens);

 the Nth word (zero-indexed) */
ens_get_token(struct tokens *tokens, size_t n);

he memory */
ns_destroy(struct tokens *tokens);
```

Makefile

```
1  SRCS=shell.c tokenizer.c
2  EXECUTABLES=shell
3
4  CC=gcc
5  CFLAGS=-g -Wall -std=gnu99
6  LDFLAGS=
7
8  OBJS=$(SRCS:.c=.o)
9
10 all: $(EXECUTABLES)
11
12 $(EXECUTABLES): $(OBJS)
13     $(CC) $(CFLAGS) $(OBJS) $(LDFLAGS) -o $@
14
15 .c.o:
16     $(CC) $(CFLAGS) -c $< -o $@
17
18 clean:
19     rm -rf $(EXECUTABLES) $(OBJS)
20
```

cd

You should use the functions defined in `tokenizer.c` for separating the input text into words. You do not need to support any parsing features that are not supported by `tokenizer.c`. Once you implement this step, you should be able to execute programs like this:

```
$ ./shell
0: /usr/bin/wc shell.c
      77     262    1843 shell.c
1: exit
```

When your shell needs to execute a program, it should fork a child process, which calls one of the `exec` functions to run the new program. The parent process should wait until the child process completes and then continue listening for more commands.

4 Path resolution

You probably found that it was a pain to test your shell in the previous part because you had to type the full path of every program. Luckily, every program (including your `shell` program) has access to a set of "environment variables", which is structured as a hashtable of string keys to string values. One of these environment variables is the `PATH` variable. You can print the `PATH` variable of your current environment on your Vagrant VM: (use `bash` for this, not your homemade shell!)

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:...
```

When `bash` or any other shell executes a program like `wc`, it looks for a program called "wc" in each directory on the `PATH` environment variable and runs the first one that it finds. The directories on the path are separated with a colon.

Modify your shell so that it uses the `PATH` variable from the environment to resolve program names. Typing in the full pathname of the executable should still be supported. **Do not use "execvp". The autograder looks for "execvp", and you won't receive a grade if that word is found.** Use `execv` instead and implement your own `PATH` resolution.

5 Input/Output Redirection

When running programs, it is sometimes useful to provide input from a file or to direct output to a file. The syntax "`[process] > [file]`" tells your shell to redirect the process's standard output to a file. Similarly, the syntax "`[process] < [file]`" tells your shell to feed the contents of a file to the process's standard input.

Modfiy your shell so that it supports redirecting stdin and stdout to files. You do not need to support redirection for shell built-in commands. You do not need to support stderr redirection or appending to files (e.g. "`[process] >> [file]`"). You can assume that there will always be spaces around special characters `<` and `>`. Be aware that the "`< [file]`" or "`> [file]`" are NOT passed as arguments to the program.

6 Signal Handling and Terminal Control

Most shells let you stop or pause processes with special key strokes. These special keystrokes, such as `Ctrl-C` or `Ctrl-Z`, work by sending signals to the shell's subprocesses. For example, pressing `CTRL-C` sends the `SIGINT` signal which usually stops the current program, and pressing `CTRL-Z` sends the `SIGTSTP` signal which usually sends the current program to the background. If you try these keystrokes in your

shell, the signals are sent directly to the shell process itself. This is not what we want since, for example, attempting to `CTRL-Z` a subprocess of your shell will also stop the shell itself. We want to have the signals affect only the subprocesses that our shell creates.

Before we explain how you can achieve this effect, let's discuss some more operating system concepts.

6.1 Process groups

We have already established that every process has a unique process ID (pid). Every process also has a (possibly non-unique) process group ID (pgid) which, by default, is the same as the pgid of its parent process. Processes can get and set their process group ID with `getpgid()`, `setpgid()`, `getpgrp()`, or `setpgrp()`.

Keep in mind that, when your shell starts a new program, that program might require multiple processes to function correctly. All of these processes will inherit the same process group ID of the original process. So, it may be a good idea to put each shell subprocess in its own process group, to simplify your bookkeeping. When you move each subprocess into its own process group, the pgid should be equal to the pid.

6.2 Foreground terminal

Every terminal has an associated "foreground" process group ID. When you type `CTRL-C`, your terminal sends a signal to every process inside the foreground process group. You can change which process group is in the foreground of a terminal with "`tcsetpgrp(int fd, pid_t pgrp)`". The `fd` should be `0` for "standard input".

6.3 Overview of signals

Signals are asynchronous messages that are delivered to processes. They are identified by their signal number, but they also have somewhat human-friendly names that all start with `SIG`. Some common ones include:

- **SIGINT** - Delivered when you type `CTRL-C`. By default, this stops the program.
- **SIGQUIT** - Delivered when you type `CTRL-\`. By default, this also stops the program, but programs treat this signal more seriously than SIGINT. This signal also attempts to produce a core dump of the program before exiting.
- **SIGKILL** - There is no keyboard shortcut for this. This signal stops the program forcibly and cannot be overridden by the program. (Most other signals can be ignored by the program.)
- **SIGTERM** - There is no keyboard shortcut for this either. It behaves the same way as SIGQUIT.
- **SIGTSTP** - Delivered when you type `CTRL-Z`. By default, this pauses the program. In bash, if you type `CTRL-Z`, the current program will be paused and bash (which can detect that you paused the current program) will start accepting more commands.
- **SIGCONT** - Delivered when you run `fg` or `fg %NUMBER` in bash. This signal resumes a paused program.
- **SIGTTIN** - Delivered to a background process that is trying to read input from the keyboard. By default, this pauses the program, since background processes cannot read input from the keyboard. When you resume the background process with SIGCONT and put it in the foreground, it can try to read input from the keyboard again.

- **SIGTTOU** - Delivered to a background process that is trying to write output to the terminal console, but there is another foreground process that is using the terminal. Behaves the same as SIGTTIN by default.

In your shell, you can use `kill -XXX PID`, where `XXX` is the human-friendly suffix of the desired signal, to send any signal to the process with process id `PID`. For example, `kill -TERM PID` sends a `SIGTERM` to the process with process id `PID`.

In C, you can use the `signal` function to change how signals are handled by the current process. The shell should basically ignore most of these signals, whereas the shell's subprocesses should respond with

Dual Mode Of OPERATION

DESTROYYYYYYYYYYYYYYYYYYY by PGID?!

:). Nice bucket ot keep around

ALERT

Alias For signal Integers

As identifeirs They are Unqiue!

GIVE ME A CHANCE TO
BEFORE YOU TAKE OVE

JUST GIVE ME A CHANC

What's in a pid?
Maybe more than me
Unique integer.

In addition to a pid, yo
Another identifer, PGI

Not unique, but it lets

We want some more r

Convention, THIS STOPS THE PROGRAm
PRESSING ISSUE

STOP EVERYTHING
BURN TI TO THE GROUND
LETS GET OUT OF HERE

STRONGER SIGQUIT

Alias for SIGQUIT

PAUSE

LETS GOOOOOOO

???

OS provides servic

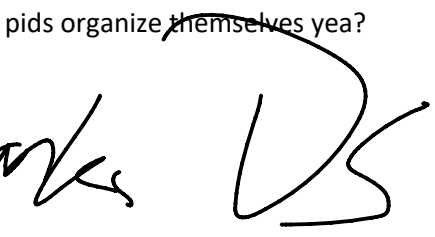
HANDLE IT
R

CE

ets the eye of some

ou can get
D.

pids organize themselves yea?



metadata please!

, WE HAVE A

Not synchronized with the process
Can come in at ANY time.

Asnychrnous vs interrupt? Sounds very similar
Interrupts must be a special type of
Asynch SIGNAL/MESSAGE.
High priority message I guess :0
Not in queue, straight 2 ya face

ces to apps

to send any signal to the process with process id `PID`. For example, `kill -TERM PID` sends a `SIGTERM` to the process with process id `PID`.

In C, you can use the `signal` function to change how signals are handled by the current process. The shell should basically ignore most of these signals, whereas the shell's subprocesses should respond with the default action. For example, the shell should ignore SIGTTOU, but the subprocesses should not. Beware: forked processes will inherit the signal handlers of the original process. Reading `man 2 signal` and `man 7 signal` will provide more information. Be sure to check out the `SIG_DFL` and `SIG_IGN` constants. For more information about how signals work, please work through the tutorial here.

Your task is to ensure that each program you start is in its own process group. When you start a process, its process group should be placed in the foreground. Stopping signals should only affect the foregrounded program, not the backgrounded shell.

7 Background processing

So far, your shell waits for each program to finish before starting the next one. Many shells allow you run a command in the background by putting an "&" at the end of the command line. After the background program is started, the shell allows you to start more processes without waiting for background process to finish.

Modify your shell so that it runs commands that end in an "&" in the background. You only need to support background processing for programs, not built-in commands. Once you've implemented this feature, you should be able to run programs in the background with a command such as "`/bin/ls &`".

You should also add a new built-in command `wait`, which waits until *all* background jobs have terminated before returning to the prompt.

You can assume that there will always be spaces around the `&` character. You can assume that, if there is a `&` character, it will be the last token on that line.

8 Optional: Foreground/Background Switching

Most shells allow for running processes to be toggled between running in the foreground versus in background. You can optionally add two built-in commands to support this:

- "`fg [pid]`" – Move the process with id `pid` to the foreground. The process should resume if it was paused. If `pid` is not specified, then move the most recently launched process to the foreground.
- "`bg [pid]`" – Resume a paused background process. If `pid` is not specified, then resume the most recently launched process.

You should keep a list of all programs you've started, whether they are in the foreground or background. Inside this list, you should also keep a "`struct termios`" to store the terminal settings of each program.

9 Autograder and Submission

To submit and push to autograder, first commit your changes, then do:

```
$ git push personal master
```


OS provides servic

Some of it provide
About it's OWN PC

q

ces to apps

es it some meta data
CB.

Within 30 minutes you should receive an email from the autograder. (If you haven't received an email within half an hour, please notify the instructors via a private post on Piazza.)

Please ensure that your solution does not print any extraneous output when stdin is not a terminal. That is, any time a built-in or a process in run with your shell, *only the output of the built-in or process should be printed*. Please do not print anything extra for debugging, as this can mess up your autograder results.