← Project 4: A Shell

Due by midnight, Tuesday 11/20 (or late on Wednesday)

> While working on your shell, please limit your user processes by running `ulimit -u 20` after logging into thoth. You will have to do this whenever you log in. If you think you've forkbombed thoth, or if thoth is running very slowly, please EMAIL me as soon as possible so that I can clean it up.

In this project, you'll be making a simple Unix shell. A shell is what you interact with when you log into thoth - it's a command-line interface for running programs.

You already did some of this project in lab 7 - that's exactly how you'll run programs!

Isn't a shell a special kind of program?

Nope! A shell is just a user-mode process that lets you interact with the operating system. The basic operation of a shell can be summed up as:

1. Read a command from the user
2. Parse that command
3. If the command is valid, run it
4. Go back to step 1

The shell you interact with when you log into thoth is called `bash`. You can even run `bash` inside itself:

```
(13) thoth $ bash
(1)  thoth $ pstree <yourusername>
sshd---bash---bash---pstree
(2)  thoth $ exit
(14) thoth $ _
```

You'll see the command numbers change, since you're running `bash` inside of `bash`. `pstree` will also show this - a `bash` process nested inside another `bash` process!

When you write your shell, you can test it like any other program you've written.

```
(22) thoth $ ./myshell
myshell> ls
myshell    myshell.c
myshell> exit
(23) thoth $ _
```

Input tokenization

Use `fgets()` with a generously-sized input buffer, like 300 characters.

Once you have the input, you can *tokenize* it (split it into "words") with the `strtok()` function. It behaves oddly, so be sure to read up on it.

Here is a sample program that demonstrates `strtok`.. Feel free to use it as the basis for your command parsing, but remember...

> Since `strtok` operates in place, you cannot return the resulting array from a function. You have to allocate that array in the function that needs it, and pass a `char**` pointer.

In the worst case where someone types e.g. `a b c d e f g h i...`, you could have half as many tokens as the size of your character buffer - so, 150 tokens.

For `strtok()`'s "delim" parameter, you can give it this string:

```
" \t\n"
```

> Get the string tokenization working *first*. Test it out well, and try edge cases - typing nothing, typing many things, typing several spaces in a row, using tab characters...

Commands

Many of the commands you're used to running in the shell are actually *builtins* - commands that the shell understands and executes instead of having another program execute them.

Anything that **isn't** a builtin should be interpreted as a command to run a program.

Following is a list of commands you need to support.

`exit` and `exit number`

System calls needed: `exit()`

The simplest command is `exit`, as it just... exits the shell.

> ***NOTE: In all these examples,*** `myshell>` ***indicates your shell program's prompt, and*** `$` ***indicates bash's prompt.***

```
$ ./myshell
myshell> exit
$ _
```

You also need to support giving an argument to `exit`. It should be a number, and it will be returned to bash. You can check it like so:

```
myshell> exit 45
$ echo $?
45
$ _
```

The `echo $?` command *in bash* will show the exit code from the last program.

If no argument is given to `exit`, it should return 0:

```
myshell> exit
$ echo $?
0
$ _
```

Hint: there are a few functions in the C standard library you can use to parse integers from strings.

`cd dirname`

System calls needed: `chdir()`

You know how `cd` works! You don't have to do anything special for the stuff that comes after the `cd`. `chdir()` handles it all for you.

> Really, `chdir()` handles it *all* for you. You don't have to parse the path, or look for '..', or make sure paths are relative/absolute etc. `chdir()` is like `cd` in function form.**

You do not need to support `cd` without an argument. Just regular old `cd`.

You can see if it works properly using the `pwd` program, once your shell can run regular programs.

```
myshell> cd test
myshell> pwd
/afs/pitt.edu/home/x/y/xyz00/private/test
myshell> cd ..
myshell> pwd
/afs/pitt.edu/home/x/y/xyz00/private
myshell> _
```

Regular programs

System calls needed: `fork()`, `execvp()`, `exit()`, `waitpid()`, `signal()`

If something doesn't look like any built-in command, **run it as a regular program.** You should support commands with or without arguments.

You basically did this with **lab 7**! You can use that as a starting point.

Your shell should support ANY number of arguments to programs, not just zero or one.

For example, **and these are just examples:** ANY program should be able to be run like this:

```
myshell> ls
myshell.c    myshell    Makefile
myshell> pwd
/afs/pitt.edu/home/x/y/xyz00/private
myshell> echo "hello"
hello
myshell> echo 1 2 3 4 5
1 2 3 4 5
myshell> touch one two three
myshell> ls -lh .
total 9K
-rw-r--r-- 1 xyz00 UNKNOWN1 2.8K Apr  9 22:04 myshell.c
-rwxr-xr-x 1 xyz00 UNKNOWN1 4.4K Apr  9 22:04 myshell
-rw-r--r-- 1 xyz00 UNKNOWN1  319 Apr  9 18:51 Makefile
-rw-r--r-- 1 xyz00 UNKNOWN1    0 Apr  9 22:05 one
-rw-r--r-- 1 xyz00 UNKNOWN1    0 Apr  9 22:05 two
-rw-r--r-- 1 xyz00 UNKNOWN1    0 Apr  9 22:05 three
myshell> _
```

Catching Ctrl+C

Ctrl+C is a useful way to stop a running process. However by default, if you Ctrl+C while a child process is running, the parent will terminate too. So if you try to use it while running a program in your shell...

```
$ ./myshell
myshell> cat
typing stuff here...
typing stuff here...
cat just copies everything I type.
cat just copies everything I type.
<ctrl+C>
$ _
```

I tried to exit `cat` by using Ctrl+C but it exited my shell too!

Making this work right is pretty easy.

- At the beginning of `main`, set it to *ignore* SIGINT.
- In the child process (after `fork`, before `exec`), set its `SIGINT` behavior to the *default*.

Once that's done, you can use Ctrl+C with abandon:

```
$ ./myshell
myshell> cat
blah
blah
blahhhhh
blahhhhh
<ctrl+C>
myshell> exit
$ _
```

The Parent Process

After using `fork()`, the parent process should wait for its child to complete. Things to make sure to implement:

- Make sure to check the return value from `waitpid` to see if it failed. (This is just like in lab7.)
- If the child did *not* exit normally (`WIFEXITED` gives false):
  - if the child terminated due to a signal, print out which signal killed it. (Again, lab7!)
  - otherwise, just say it terminated abnormally.

The Child Process

After using `fork()`, the child process is responsible for running the program. Things to make sure to implement:

- Set the SIGINT behavior to the default (as explained above in the Ctrl+C section).
- Use `execvp` to run the program.
- Print an error if `execvp` failed.

> AND THEN.... exit() after you print the error. DON'T FORGET TO EXIT HERE. This is how you forkbomb. *If you forkbomb thoth multiple times, even if by accident, you may have your login privileges revoked.*

Notes on using `execvp`:

- The way `execvp` detects how many arguments you've given it is by **putting a NULL string pointer as the "last" argument.** You must put the NULL in your arguments array yourself, after parsing the user input. (The `strtok` example above does this.)

- `execvp` *only* returns if it failed. So you don't technically need to check its return value.

Input and Output redirection

Functions needed: `freopen()`

Any regular program should also support having its stdin, stdout, **or both** redirected with the `<` and `>` symbols.

> The redirections can come in either order, like `cat < input > output` or `cat > output < input`. Do not hardcode your shell to assume one will come before the other.

Your shell should support using input and output redirection on any *non-builtin command* with *any number of parameters*.

This means you should look for the redirections by looking starting at the *last* tokens. Then you can replace each redirection token (`<` and `>`) with NULL to ensure the right arguments get passed to the program.

> bash lets you write `ls>out` without spaces, but you don't have to support that. `ls > out` is fine for your shell.

```
myshell> ls > output
myshell> cat output
myshell.c    myshell    Makefile    output
myshell> less < Makefile

<then less runs and shows the makefile>

myshell> cat < Makefile > copy
myshell> ls
myshell.c    myshell    Makefile    output    copy
myshell> less copy

<then less runs and shows that 'copy' is identical to the orig

myshell> ls -lh . > output
myshell> cat output
total 31K
```

```
-rw-r--r-- 1 xyz00 UNKNOWN1 2.8K Apr  9 23:18 myshell.c
-rwxr-xr-x 1 xyz00 UNKNOWN1 4.4K Apr  9 23:18 myshell
-rw-r--r-- 1 xyz00 UNKNOWN1  319 Apr  9 18:51 Makefile
-rw-r--r-- 1 xyz00 UNKNOWN1   39 Apr  9 23:20 output
-rw-r--r-- 1 xyz00 UNKNOWN1  319 Apr  9 23:21 copy
myshell> _
```

Input and output redirection should detect and report the following errors:

- If the user tried to redirect stdin or stdout **more than once**
  - e.g. `ls > out1 > out2`
  - or `cat < file1 < file2`
- If the file to read or write could not be opened

Opening the redirection files

You should open the redirection files **in the child process after using `fork`,** but **before using `execvp()`.**

In order to redirect stdin and stdout, you have to open new files to take their place. `freopen()` is the right choice for this.

- When opening a file for redirecting `stdin`, you want to open the file as read-only.
- When opening a file for redirecting `stdout`, you want to open the file as write-only.

Grading Breakdown

Basics [20]

- **[6]** Submitted properly
- **[4]** Compiles with `gcc --std=c99 -Wall -Werror -o myshell myshell.c`
  - Note: if you get errors about "implicit declaration of function 'strsignal'" then add `#define _GNU_SOURCE` to the very top of your code, before any `#include` lines.
- **[10]** Code style. This includes:
  - Splitting code into sensible functions
  - Properly deallocating any memory you allocate
  - Naming things reasonably

Note: Error handling is not explicitly mentioned in any of the following, but you should be checking for errors in everything. You'll lose points if not.

Builtins [30]

- **[10]** `exit` exits the program
- **[10]** `exit xx` exits the program with exit code xx
- **[10]** `cd dir` changes the current directory to `dir`

Regular Programs [30]

- **[5]** SIGINT (ctrl+C) is properly handled - does nothing in your shell, but can interrupt running programs (like `cat`)
- **[10]** Can run a regular program, period
- **[10]** Can run a regular program with an arbitrary number of arguments
- **[5]** Displays a human-readable message about what signal killed a program (e.g. if you ctrl+C in `cat`)

Redirection [20]

- **[8]** Input redirection `<` works and gives an error if input is redirected more than once
- **[8]** Output redirection `>` works and gives an error if output is redirected more than once
- **[4]** Input and output redirection can be used *at the same time, in either order*
  - e.g. `cat < input > output` or `cat > output < input` should do the same thing)

Submission Instructions

1. Name your file `myshell.c`.
2. At the top of the file, put your username and full name in comments.
3. Comment your code, at least enough to explain what each function does.

Now you can **submit as usual.**