

Assignment 2

Creating Your Own Command Shell – `mysh`

Objectives

- Practice how processes are created and managed (Chapters 4 and 5 in book).
- Learn how a command shell works.
- Practice development in C and the Linux environment.

Overview

In this assignment you will create your own command shell (kind of like a stripped down version of *bash*). In very simple terms, a command shell is a program that tells the OS to launch other programs. A shell like *bash* also has additional built-in commands (more on that later), and other advanced features like keeping history of commands executed, redirecting the output of one command to the input of another (i.e., chaining commands), etc. There are many different Unix/Linux shells: *bash*, *ksh*, *tcsh*, *zsh*, etc. The most common one is *bash* (Borne Again SHell). You can determine which one is the default on your Linux system by typing the command:

```
echo $SHELL.
```

In GUI-rich operating systems such as Windows and macOS users typically start programs by double-clicking some icon or clicking some menu. They do not start a program by typing a command (although they can if they want to but it is not the common way of doing things). In other systems such as the Linux environment, the typical way of interacting with the OS is via commands. The user opens a terminal window, types a command, and presses the Enter key to execute the command. For example:

```
shell-prompt > gcc prog.c -o prog.o
shell-prompt > mkdir module5
shell-prompt > ./my_program
shell-prompt > cd ../myfolder
```

In the first three examples the user is asking the shell to run `gcc`, `mkdir`, and `my_program` (assume `my_program` is some program you wrote). All three are external programs. The shell simply asks the OS to run them.

Shells also have built-in commands. For example, the widely-used command `cd` (change directory) is a built-in command (and not an external program like `gcc` or `my_program`). By built-in, we mean that the shell itself provides the functionality of `cd`. If curious, the built-in commands of *bash* are listed [here](#)

– notice that `cd` is one of them.

The end user usually does not know (or care) about whether the command is external or built-in. For the end user all commands look alike: type something at the prompt and press the Enter key. As a shell developer, however, you need to distinguish between the two. At a high level, your shell should have, as part of its logic, code that resembles the below:

```
if ( command is an external program )
    Ask OS to run it // We learned how to use fork, exec, wait to do that
else
    Provide implementation
```

The shell does not know how external programs (like `gcc`, `ls`, `my_program`) work and, it doesn't need to. It only knows about built-in commands since it is providing their functionalities. Therefore, for non-built-in commands, it is not the shell's job to validate that the program was called in a correct way. It is unrealistic to expect the shell to know such details – it will have to learn the internals of all programs in the entire world. It rests on the program that was launched to check that it has all the needed information to proceed.

Specifications

Create a shell and call it **mysh**. Assuming you put all your code in a single `mysh.c` file, to build it you need to do something like:

```
gcc mysh.c -o mysh -Wall.
```

And to run your shell, someone types:

```
./mysh
```

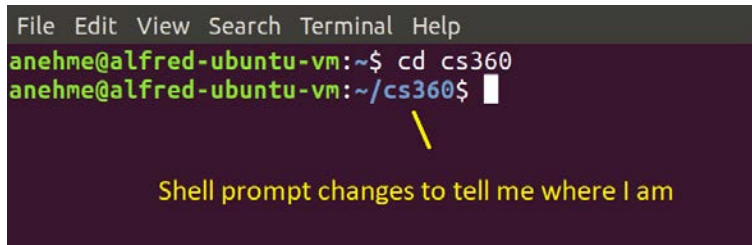
Your `mysh` shell should provide the following functionalities:

- Display a command prompt, wait for the user to type a command and press Enter, execute the command and wait for it to finish, then display another prompt again.
- Your shell should continue doing the above until the user types **exit** (more on exit in the Implementation Hints section). At this point you should exit the shell.
- When someone starts the shell from his/her home directory, the command prompt displayed should look like this: `~$mysh>` (with a single empty space after the `>` sign) or like this: `/home/jsmith$mysh>` (assuming that the user's home directory is `/home/jsmith`). Note that the `~` is a special character that evaluates to your home directory.

When someone starts your shell from a directory other than his/her home directory, the

command prompt displayed should look like: `/path/to/currentdirectory$mysh>` to show the current directory.

When the user uses `cd` to change to another directory, you should adjust the command prompt text to reflect where the user is. This is a useful feature that the shell provides to allow users to know where they are in the directory structure (without the need to keep executing `pwd`, which gives you the current directory). Most shells, including `bash`, already do this for you. Experiment with the `bash` shell on your system to see how it works. For example, in the picture below, I was initially in my home directory (`~`). When I `CD`ed to the `cs360` folder, the shell prompt changed to `~/cs360`:

A terminal window with a dark background and light green text. The menu bar at the top shows 'File Edit View Search Terminal Help'. The prompt is 'anehme@alfred-ubuntu-vm:~\$'. The user enters 'cd cs360'. The prompt changes to 'anehme@alfred-ubuntu-vm:~/cs360\$'. A yellow arrow points from the text 'Shell prompt changes to tell me where I am' below to the new prompt.

```
File Edit View Search Terminal Help
anehme@alfred-ubuntu-vm:~$ cd cs360
anehme@alfred-ubuntu-vm:~/cs360$
```

Shell prompt changes to tell me where I am

It is kind of useful because it gives me a visual hint where I am. I can always get where I am by executing `pwd` (print working directory). But it is nice not to have to keep doing it. Experiment with command `pwd`: it return to you the path of where you are in the directory structure.

- You should structure your code so that you create a new process for each external command – using `fork/exec/wait`. For build-in commands, you provide the implementations. You need to support three build-in commands only:

<code>cd</code>	Change directory
<code>pwd</code>	Print working directory
<code>exit</code>	Exit the shell

- Below are three snapshots that show various commands and how your shell should behave in general (**the only thing the snapshots don't show is the change in the prompt text after say you do `cd`**). Remember that you need to do that.

```

anehme@alfred-ubuntu-vm:~$ cd cs360/assignments/hw2
anehme@alfred-ubuntu-vm:~/cs360/assignments/hw2$ ./mysh
$mysh > pwd
/home/anehme/cs360/assignments/hw2
$mysh > mkdir newFolder
$mysh > ls
mysh mysh.c mysh_save.c newFolder
$mysh > cd newFolder
$mysh > ls
$mysh > cd ../
$mysh > ls
mysh mysh.c mysh_save.c newFolder
$mysh > ls -al
total 36
drwxr-xr-x 3 anehme anehme 4096 Feb  2 07:04 .
drwxr-xr-x 5 anehme anehme 4096 Jan 30 11:59 ..
-rwxr-xr-x 1 anehme anehme 13416 Feb  1 22:12 mysh
-rw-r--r-- 1 anehme anehme 2685 Feb  1 22:12 mysh.c
-rw-r--r-- 1 anehme anehme 3964 Feb  1 15:46 mysh_save.c
drwxr-xr-x 2 anehme anehme 4096 Feb  2 07:04 newFolder
$mysh > ps -x
  PID TTY          STAT TIME  COMMAND
 1986 ?        Ss   0:00 /lib/systemd/systemd --user
 1987 ?        S    0:00 (sd-pam)
 1998 ?        SL   0:00 /usr/lib/gnome-session/gnome-session-binary
 1999 ?        SL   0:00 /usr/lib/xorg/Xorg :10 -auth .Xauthority -config xrdp/xorg.conf -noreset -nolisten tcp -logfile .xorgxrdp.%s.log
 2014 ?        SL   0:00 /usr/sbin/xrdp-chansrv
 2060 ?        Ss   0:00 /usr/bin/dbus-daemon --session --address=systemd: --nofork --nopidfile --systemd-activation --syslog-only
 2082 ?        Ss   0:00 /usr/bin/ssh-agent /usr/bin/ln-launch x-session-manager
 2085 ?        Ssl  0:00 /usr/lib/gvfs/gvfsd

```

When you start mysh, it should display the prompt: \$mysh >

Executed command mkdir newFolder

Executed command ls and it shows that newFolder got created
cd to newFolder, then cd back to parent

Example executing ls -al

Example executing ps -x which shows processes

```

$mysh > exit
anehme@alfred-ubuntu-vm:~/cs360/assignments/hw2$

```

When the user types exit, you exit the shell

```

$mysh > hello
Failed to execute command

```

There is no command named hello. Print correct message ("Failed to execute command")

- **YOU DO NOT NEED TO IMPLEMENT** redirection, pipelining, and background execution.
- Your shell should print to stderr the following errors (if they happen):
 - If fgets fails, print **"Failed to read command"**
 - If fork fails, print **"Failed to create process"**
 - If a command fails to execute, print **"Failed to execute command"**. exec() functions return only if an error has occurred (see documentation [here](#)). Check the return value of execvp. In some cases, if a command fails and prints its own message, then that's OK too.
 - If cd fails because the directory you are changing to does not exist, then print **"No such directory"**. For all other cd-related errors, print **"Change directory failed"**.

TIP: How do I know what is the cause of failure of cd, if any? Look at the documentation [here](#). The combination of the return value of chdir and errno tell you if cd failed and what the failure was due to.

Implementation Hints

- To read what the user typed at the prompt, use the [fgets](#) function.
- When the user types something and presses the Enter key, the string you read may contain the newline character ('\n') at its end. You need to strip that out.

- If you need to parse a command, a useful function is [strtok_r](#).
- To implement the pwd built-in command, call function [getcwd](#).
- To implement the cd built-in command, call function [chdir](#).

- To implement the exit built-in command, use:

```
exit(EXIT_SUCCESS) or exit(0)
```

- To implement external commands, use fork, execvp, and wait.
- You can make the following assumption:
 - A command (with all its arguments) cannot be more than 256 characters long.
- If you like to use the Boolean type in C, you can do so by including `stdbool.h`.

```
#include <stdbool.h>
bool my_var = false;
```

- I will build your program by running **make**. Therefore, make sure you enclose a Makefile and the Makefile works. I won't grade your assignment if it doesn't compile (and I do not fix compilation errors).

Grading Rubric

- Your built-in commands (pwd, cd, exit) do not work (10% penalty for each).
- External commands do not work or you are not using fork/wait/exec to run commands (30% penalty).
- Wrong prompt text (10% penalty). That is, your prompt is something other than "\$mysh> " or "current/path\$mysh> " where current/path is the current working directory.
- No make file given (-15%).
- Program structure is confusing, code not commented, poor code organization (up to 20% penalty depending on severity).

TEST YOUR CODE SEVERAL TIMES BEFORE SUBMITTING IT. Make sure all functionalities listed in the Specifications section work.

What to Submit

- Create a folder `firstName_lastName_hw2` (e.g., `john_smith_hw2`).
- Put in above folder your `mysh.c` and `makefile`. That is, your `firstName_lastName_hw2` folder should look like this (if all your code is in `mysh.c`):

```
john_smith_hw2
  mysh.c
  Makefile
```

- Zip above folder to `john_smith_hw2.zip`.
- Upload above zip to Canvas by the due date.

Late or Improper Submission:

- **Late submission:** Up to 2 days late (**-30% penalty**). More than 2 days late: not accepted. You will lose all points allocated to the assignment.
- **Incorrect submission:** (50% penalty). See enclosed file “Assignments Submission Check List.pdf”. Do a check and verify the items I listed in it after you submit