Advanced Exercises Set 2
More Shells

EECS 201 Winter 2020

Submission Instructions

To receive credit for this assignment you will need to stop by someone's office hours, demo your running code/system, and answer some questions. You may choose the exercises that you wish to do from this set. Each exercise is denoted by its point count. **Extra credit is given for early turn-ins of advanced exercises. These details can be found on the website under the advanced homework grading policy.**

1 Pretty `PS1` (5)

Open a new terminal and try the following commands in order:

```
echo -e "\\033[44;3;70;38;5;214m"
<hit enter again>
PS1="Hello World -- "
echo -e "\\033[44;3;70;38;5;214m"
pwd
ls
pwd
echo -e "\\033[44;3;70;38;5;214m"
ls --color=none
pwd
reset
pwd
bash --norc
echo -e "\\033[44;3;70;38;5;214m"
ls
pwd
ls --color=auto
pwd
exit
PS1="\\033[44;3;70;38;5;214mHello Again -- "
ls
```

What happened to your terminal as you ran these commands? Play around with some other forms of `PS1` and other colors, see what happens. Why does calling `ls` sometimes reset things?

Create a custom `PS1` for yourself. Look into some of the options for `PS1`, you will need to explain why you added the options you did and decided against options you didn't choose.

Extend your PS1 by writing a `bash` function the changes your prompt in a way that is not built-in to `bash`. Some examples: Add an asterisk if your are in a git repository with uncommitted changes. Change the color if you are currently in a shared directory (i.e. in a Dropbox folder). Change the color if the current directory will not be saved across a reboot (i.e. if you are you are somewhere in the `/tmp` directory). Change the color if the last command failed.

A shell variable that may be helpful is `PROMPT_COMMAND`. Remember that `manpages` offer a lot of useful info! Interestingly, the Arch wiki has a lot of useful information on various utilities...

Submission checkoff:

- ☐ Explain what `PS1` does
- ☐ Explain what you did to customize `PS1` and **why** you chose the customizations that you did.
  - ☐ Explain how your custom function works.
- ☐ Explain what the `PS2` variable controls. Change `PS2` from the default and show an example.
- ☐ Type `set -x`. Then type `ls`. Explain all of the output.

2 Understanding Tab Completions (5)

Open a new terminal and try typing the following (note <tab> means press the tab key):

```
1 p<tab><tab>
2 y
3 <ctrl-c>
4 pi<tab><tab>
5 pin<tab><tab>
6 ping<tab><tab>
7 ping <tab><tab>
8 PATH=<enter>
9 p<tab><tab>
```

In addition to finding programs, tab completions can help you to use a program correctly by hinting at what arguments a program accepts, try this:

```
10 # Open a new terminal (or manually set your PATH correctly again)
11 ping <tab><tab>
12 ping -<tab><tab>
13 ping -I <tab><tab>
14 ping -Q <tab><tab>
15 ping -Q 0 <tab><tab>
```

Today, most programs include tab completion support, but this is a remarkably manual process. Check out the contents of the `/usr/share/bash-completion/completions/` directory.

Now take a look at `/usr/share/bash-completion/completions/ping`. There's a lot going on in this example, but try to see if you can understand some of how the completions are working. What do you think the result of `ping -T` <tab><tab> will be?

(Hint: `||` in bash means "if the previous thing failed, do the next thing". What's the output of `echo $OSTYPE`? Will that equality pass or fail?)

Submission checkoff:

- ☐ Why is the list of tab completions different between lines 1 and 9?
- ☐ What is the default tab completion behavior for a program if no custom completion function has been written?

3 Testing Made Easier (5)

When working on a project in EECS, you should be writing test cases to make sure your program is functioning properly. Checking your test cases can be tedious, but fortunately for us, scripting can help make it easier to run your test cases and report which ones are passing and which ones are failing. Here, we will do just that.
Go ahead and download some files from this URL, extract them, and take a look. You should see 3 files. `testPass.sh`, `testFail.sh`, and `testTimeout.sh`. These files will pass/fail according to their names. You should not need to modify their contents.

```
> wget https://www.eecs.umich.edu/courses/eecs201/files/assignments/adv2-files.tar.gz
> # e(X)tract (Z)ipped (F)ile
> tar xzf adv2-files.tar.gz
```

Write a shell script which takes all files in the current directory with "test" somewhere in the name, runs each of them, and reports whether the program passed or failed the test case by printing "PASSED", "FAILED". Your script should also stop programs from running for more than 3 seconds and print "TIMEOUT".
An example test runner file to use as a framework is below. A solution may follow the structure very closely, but remember, there are almost always multiple ways of solving the same problem, so feel free to deviate from the suggestions.
One requirement is to not hard code the list of filenames into the script, this is where shell globbing/wildcards can help (we suggest looking these up to learn more). This requirement is so that you can run your master script in any directory with files including "test" in the name and it should Just Work ™.

```
#!/bin/bash

# loop through all files with ‘test‘ in the name
# (learning more about for loops and shell globbing/wildcards will help for this)

    # for each file, execute it
    # (you may need to execute the file with another program that will handle the
    # timeout case)

    # save the return code of the previous execution into a variable

    # check the return code for timeout, print "TIMEOUT" if so

    # check the return code for failure, print "FAILURE" if so

    # check the return code for success, print "SUCCESS" if so
```

Submission checkoff:

- ☐ How does the exit command work? (Hint: How is this similar to `return` in C/C++ ?)
- ☐ How can a test case report to our test running script whether the target program passed or failed?
- ☐ How can you tell if a program is running too long?
- ☐ Can you demo your code?

4 Programming with POSIX (5)

In this week's lecture exit survey, I asked if you think you could write a shell. No matter what your answer was, I believe that you are capable of writing a basic shell :)

I briefly mentioned in lecture how proccesses are created in Unix by `fork`-ing and `exec`-ing. Let's showcase some POSIX programming and get some practice reading `man` pages with this exercise. In this exercise you'll be creating "μShell", or "`mush`" using C/C++ (your choice on either). `mush` is a simple, minimalist shell whose only job is to execute the commands presented to it.
Its specifications are:

- Presents a prompt of "`<username>:<current working directory>$ `".
  For example, for a `USER` "doe": "`doe:/home/doe$ `". Note the space at the end. Assume that the current working directory can only be represented by at most 127 characters.
- Assume that the entered input has at most 127 characters and that there will be at most 15 arguments.
- Stops and exits with a 0 if the command is `exit`
- Changes the current working directory if the command is `cd`; if no path is specified, the current working directory is set to `HOME`. `PWD` does **not have to** reflect this change. If the directory does not exist, prints "`mush: no such file or directory '<directory path>'`".
  For example: "`mush: no such file or directory '/emoh/doe'`".
- Executes the entered command and with its arguments. These commands are either in the `PATH` or specified with a path (i.e. has a forward slash in it). That means you do not have to implement any other shell built-ins besides `exit` and `cd`.
- If the entered input is empty, prompts the user again.
- If the command does not exist, prints "`mush: command '<command name>' not found`".
  For example: "`mush: command 'iamnotacommand' not found`".

Note that there are no built-in commands besides `exit` and `cd`, job control, file redirection, or signal handling specified (if you want to, you can do them for personal edification).
Some helpful functions:

- `getenv`
- `getcwd`
- `fgets`
  - `fgets` has a gotcha where it'll include the newline character. Be sure to deal with it accordingly
- `strtok` and its reentrant sibling `strtok_r`
  - `strtok(_r)` has a gotcha where additional invocations on the same string have the `str` argument be `NULL`.
- `strcmp`
  - Note that 0 is returned when strings match.
- `chdir`
  - Note that `chdir` does not change the `PWD` environment variable for you.
- `fork`
  - Take note of how the return value differs between the parent and child process.
- `execvp`
  - The `man` page for `execve` may offer some more info.
- `waitpid`

You can see documentation for these functions by using `man`.
Here are some helpful variables/macros for error checking:

- errno
- ENOENT

You can read about them in the errno man page.
Below is an example way to structure the program. Remember that there are many different solutions possible:

```
// forever loop
  // print prompt
  // receive user input
  // parse input
  // handle "exit" and "cd"
  // fork
    // child
      // execute command
      // handle errors
    // parent
      // wait for child
```

This can be implemented in surprisingly little code: it's possible to do this in 50 lines of plain old C even with the header includes!
For me the receiving and parsing input was the most difficult part of this.

Submission checkoff:

☐ Demonstrate that it presents the correct prompt.

☐ Demonstrate that it can execute commands with arguments.

☐ Demonstrate that it can print the command not found message.

☐ Demonstrate that it can handle empty input.

☐ Demonstrate that it can exit using `exit`, with an exit code of 0.

☐ Demonstrate that it can change directories using `cd`.

☐ Demonstrate that it can print the directory not found message with `cd`.

☐ Why should the parent handle `exit` and `cd` and not the child?

☐ How was this taste of systems programming? Did you enjoy it? Did you learn anything from this experience?