Advanced 3
The Shell

EECS 201 Fall 2020

Submission Instructions

This assignment will be submitted as a repository on the UMich GitLab server. Create a Project on it with the name/path `eecs201-adv3` and add `brng` as a Reporter. The repository should have the following directory structure, starting from the repository's root:

```
/
|-- report.txt
|-- testing/
| |-- runner.sh
|
|-- shell/
    |-- Makefile
    |-- (source files)
```

Do not commit any build output (compiled executables and object code) or any file system data like `.DS_STORE` on mac OS.

Preface

This assignment can be done on both macOS and Linux.

First, we'll need to acquire the starter files. Use your preferred method of downloading files from a given URL:

`https://www.eecs.umich.edu/courses/eecs201/files/assignments/adv3.tar.gz`

Don't create your Git repo inside it: they don't have the right directory structure. Create another directory and initialize a repository there and copy over whatever files you need.

1 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.

In the file archive you downloaded, you should see 3 files under `testing/`: `test-pass.sh`, `test-fail.sh`, and `test-timeout.sh`. These files will pass/fail according to their names. You should not need to modify their contents.

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 "SUCCESS" or "FAILURE". **Note that the file's name does not necessarily need to have a filename extension or have "test" at the beginning.** Your script should also stop programs from running for more than 3 seconds and print "TIMEOUT".

Also note that these files are not necessarily Bash scripts: you should not assume that they are to be run with Bash. You should be directly executing the file instead of explicitly passing it to Bash. For example, you should be running `./test-pass.sh`. You will have to set the execute bit yourself for the provided test files. This allows for flexibility: maybe we wanted to write a test in Python? **The runner script is NOT responsible for setting execute bits. The runner script can assume that the test files have their execute bits set. However, do note that the test files you have been provided don't have them set; you'll have to do it yourself outside of the script.**

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™.

In your submission repository, create this script file under `<repo path>/testing/runner.sh`. Make this file executable: Git is able to track these file permissions. (If you using WSL and are doing this on your Windows filesystem, it's likely that everything is considered executable. That's because the Windows filesystem doesn't have these permissions/modes. Look up how to get Git to set these bits from the Repository's standpoint.)

```
#!/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)

    # perhaps save the exit status of the previous execution into a variable

    # check the exit status for timeout, print "TIMEOUT" if so

    # check the exit status for failure, print "FAILURE" if so

    # check the exit status for success, print "SUCCESS" if so
```

Things to keep in mind:

- ☐ How does the `exit` command work? (Hint: How is this similar to `return` in C/C++ ?)
- ☐ Are you directly executing each test file (e.g. `./test-example.sh`)?
- ☐ 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?

2 Programming with POSIX (10)

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 or C++ (whichever you prefer). `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>$ `" on **standard output**.

  For example, for a `USER` "doe": "`doe:/home/doe$ `". Note the space at the end. Since users type in their command after this, don't print a newline. Assume that the current working directory can only be represented by at most 255 characters.

- Assume that the entered input has at most 255 characters and that there will be at most 15 arguments.

- If the entered input is empty, prompts the user again.

- Stops, prints a newline, and exits with a 0 if the end of the file (EOF) is reached is encountered with an empty line. You can send the EOF with Ctrl-D at a terminal (try it out and see what happens with Bash or Zsh!).

- Stops and exits with a 0 if the command is `exit` with no additional argument. If there is an additional argument in the form of an integer, it exits with that integer value. **You may assume that this additional argument will always be an integer.**

- 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: cd: no such file or directory '<directory path>'`" on **standard error**.

  For example: "`mush: cd: no such file or directory '/emoh/doe'`". This string should include a newline at the end so that the prompt appears on the next line.

- 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`.

- Waits for the command run to be complete (see `waitpid`).

- If the command does not exist, prints "`mush: command '<command name>' not found`" on **standard error**. For example: "`mush: command 'iamnotacommand' not found`". This string should include a newline at the end so that the prompt appears on the next line.

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 (the string processing ones are more for C; if you're using C++ you might have other mechanisms that fulfill the same role):

- `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 (you don't have to worry about `PWD`).
- `fork`
  - Take note of how the return value differs between the parent and child process.
- `execvp`
  - The `man` pages for `execve` and `execvp` may offer some more info. Be very careful to read up on the data format of the arguments...
- `waitpid`

You can see documentation for these functions by using `man`. Sometimes, there may be multiple `man` pages for a given function. `man` pages have multiple sections: sections 2 and 3 are the programmer's manual, with 2 being the operating system API and 3 being about library functions. You may often see things like `fork(2)`: this refers to `fork` under section 2. You can specify a section to look at as an argument: `man 2 fork`.
Here are some helpful variables/macros for error checking:

- `errno`
- `ENOENT`

You can read about them in the `errno` `man` page: `man 3 errno`.
Below is an example way to structure the program. Remember that there are many different solutions possible:

```
1  // forever loop
2    // print prompt
3    // receive user input
4    // parse input
5    // handle EOF
6    // handle "exit" and "cd"
7    // fork
8      // child
9        // execute command
10       // handle errors
11     // parent
12       // wait for child
```

In the file archive you downloaded, you can find a Makefile and starter file for your preferred language in `shell/`. You may only use the C standard library, the C++ standard library, and any POSIX function. This can be implemented in surprisingly little code: it's possible to do this in less than 50 lines of plain old C!
For me the receiving and parsing input was the most difficult part of this.

In your submission repository, put the Makefile and any other source code files involved in compilation under `<repo path>/shell/`.

Things to keep in mind:

- ☐ Does it present the correct prompt **on standard output**?
- ☐ Does it execute commands with arguments?
- ☐ Does it print the command not found message?
- ☐ Does it handle empty input?
- ☐ Does it print a newline and return 0 when an EOF is encountered??
- ☐ Does it handle both cases of `exit`?
- ☐ Does it change directories using `cd`?
- ☐ Does it print the directory not found message with `cd` **on standard error**?
- ☐ Does it print the command not found message **on standard error**?
- ☐ Why should the parent handle `exit` and `cd` and not the child?

Report

Create a file `report.txt` in your repository's root: `<repo path>/report.txt`. On the first line of the file, put down an estimate of how long you took to do this assignment in minutes as an integer (e.g. "37", "84": just numbers, no letters). On the second line and onwards of the file, put down what you learned (if anything) by doing this assignment. If you already knew how to do all of this, just put "N/A".