How to `diff`

CS 15

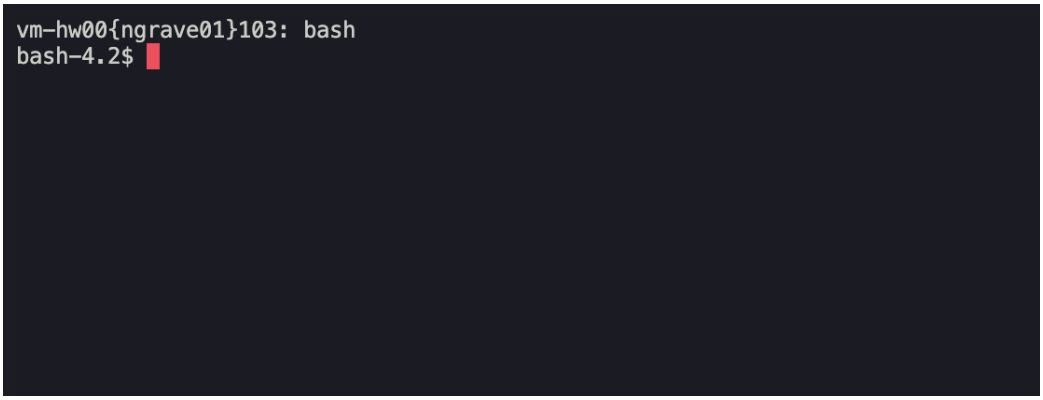
December 2021

For many assignments in this course you will receive a *reference* that exhibits the correct behavior of an assignment. In other words, your implementation's behavior must match the reference's behavior, if it does not, then there is something wrong with your implementation. The `diff` command is a handy tool to compare the output between the two implementations.

`diff` is a command-line tool for Unix systems that allows you to compare the contents of two files or directories and see the differences between them. This tutorial outlines how to utilize this command to compare the output of your program with the output of the reference program. This is a good testing strategy to confirm that your program is mirroring the behavior of the reference program. The following are steps on how to do this:

1 Get a BASH Shell

To make this process a bit more straightforward we will need to get a `bash` shell for some of the commands to work. To do this all we need to do is type `bash` into the terminal and press `return`:



The default shell on the Halligan Server is `tcsh`. Although, `tcsh` does support redirection, it is a bit more complicated to perform in a `tcsh` shell than in a `bash` shell. [1]

2 Store Our Output to a File

The first step is to store the output our implementation generates given a set of commands into a file. To do this, we will create a file containing a set of commands for our program and *redirect* those commands to `stdin` using `<` redirection operator.

```
./MetroSim stations.txt my_logfile.log < commands.txt
```

To redirect **all output** of our program to a file with use the `&>` redirection operator.

```
./MetroSim stations.txt my_logfile.log < commands.txt &> my_output.out
```

This will redirect both `stdout` and `stderr` generated by our program to a file named `my_output.out`.

Sometimes it is helpful to split `stdout` and `stderr` to separate files. We do this by using the `>` and `2>` operators, respectively.

```
./MetroSim stations.txt my_logfile.log < commands.txt > my_output.stdout 2>
    my_error.stderr
```

[1] *Note*: To close the `bash` shell and return to the default shell on Halligan you can type `exit` and press `return`

The above line will redirect `stdout` to the file named `my_output.stdout` and `stderr` to the file named `my_error.stderr`. [2]

3 Store Reference Output to a File

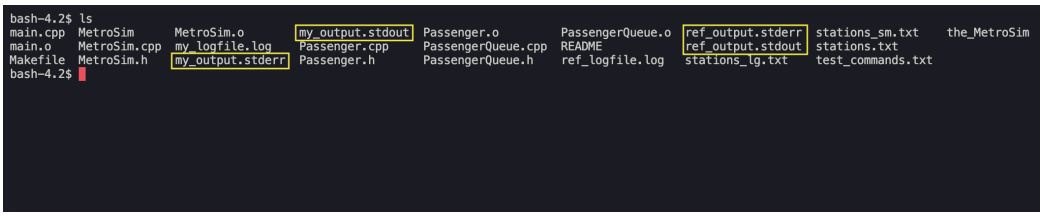
After we have successfully stored our output to files we follow the same steps using the same commands with reference implemenation to store its output to separate files.

```
./the_MetroSim stations.txt ref_logfile.log < commands.txt > ref_output.stdout
    2> ref_error.stderr
```

This line redirects the reference program's `stdout` to `ref_output.stdout` and `stderr` to `ref_error.stderr`.

4 `diff` the Files

At this point, we have successfully stored the `stdout` and `stderr` of our program and the reference program to separate files:



The command, `diff`, takes two command-line arguments, the filenames of the two files we want to compare:

```
diff [file1] [file2]
```

For example, to compare the output we captured from `stderr` we use the following line:

```
./diff my_output.stdout ref_output.stdout
```

[2] *Note*: the `1>` redirection operator also redirects `stdout` that same way `>` would.

Matching Files

If our output matches the reference output (i.e. the files are identical), then no output will be returned:

```
bash-4.2$ diff my_output.stderr ref_output.stderr
bash-4.2$
```

This is good, it means that with the given commands, our implementation produced output identical to the reference implementation.

Unmatching Files

If there is a difference between the two files the output will look similar to this:

Example 1

```
bash-4.2$ diff my_output.stderr ref_output.stderr
1c1
< Thanks for playing MetroSim. Have a nice day
---
> Thanks for playing MetroSim. Have a nice day!
bash-4.2$
```

The '<' refers to `file1` and '>' refers to `file2`. The `1c1` tell us what lines were affected and which action was performed. The action performed is designated by the letter in between the two numbers. The letter `c` stands for *change*, `d` stands for *delete*, and `a` stands for *add*. The number on the left of the character refers to the line number in `file1`. The number on the right of the character refers to the line number in `file2`. So, `1c1` tells us that in order for the two files to match, the first line in `my_output.stderr` should be changed to the first line of `ref_output.stderr`. It then tells us what those lines are, as mentioned about lines preceded by `<` are lines from the first file, and lines preceded by `>` are lines from the second file. The three dashes ("—") simply separate the lines from the two files.

Example 2

```
bash-4.2$ diff my_output.stdout ref_output.stdout
0a1
> Passengers on the train: {}
27c28
< Command?
---
> Command? Passengers on the train: {}
bash-4.2$
```

In this example, `0a1` tells us that the line, "`Passengers on the train: {}`", must be added to the line *after* the 0*th* line (i.e. the first line) of `my_output.stdout` to match `ref_output.stdout`. Similar to the first example, `27c28` tells us that the line 27 of `my_output.stdout` must be changed to match line 28 of `ref_output.stdout` in order for the files to match.

Example 3

```
bash-4.2$ diff my_output.stdout ref_output.stdout
29d28
< Command?
bash-4.2$
```

In this last example, `29d28` tells us that line 29 in `my_output.stdout` should be deleted in order for the two files to match at line 28.

5 Conclusion

Using the `diff` command to compare your program's output with the reference program's output is a very good method for testing if your program is handling commands correctly. If there is any difference between your output and the reference's output then it means that **you have implemented something incorrectly** and should change your implementation so that your ouput matches the reference's. It is important to note that this method is only as strong as your testing, you are responsible for inputting a set of commands that tests your implementation thoroughly and completely.