Assignment #1: Linux

Due Date 1: Friday, September 17, 2021, 5:00 pm EST
Due Date 2: Friday, September 24, 2021, 5:00 pm EST
Online Quiz: Monday, September 27, 2021, 5:00 pm EST

Questions 1 and 2 are due on Due Date 1; the remainder of the assignment is due on Due Date 2. If you joined the course within two days of Due Date 1, then the entire assignment is due on Due Date 2. You must submit the online quiz on Learn by the Quiz date.

Topics that must have been completed before starting Due Date 1:

1. Linux: The Teaching Environment
2. Linux: Interacting with the Shell
3. Linux: Directories and Files
4. Linux: Regular Expressions
5. Handouts: Getting Started
6. Handouts: Linux Commands

Topics that must have been completed before starting Due Date 2:

1. Linux: bash scripts
2. Software Testing

Note 1: We suggest creating the directory `~/cs246/f21/a1` and creating all the assignment solutions in this directory.
Note 2: If you finish A1 DD1 i.e. questions 1 and 2 early, we *strongly* recommend that you start the other questions, since debugging bash scripts can take a lot of time. Make sure that you take advantage of using the `-x` option to the shbang line (though **remove it from your submitted scripts!**) if your scripts aren't behaving as expected, so that you can see exactly what is happening.

bash and Regular Expressions

1. **10 marks.** Provide a Linux command line to accomplish each of the following tasks. Your answer in each subquestion should consist of a single command or pipeline of commands, with no separating semicolons (;). (Please verify before submitting that your solution consists of a single line. Use `wc` for this.) Before beginning this question, familiarize yourself with the commands outlined on the Linux handout. Keep in mind that some commands have options not listed on the sheet, so you may need to examine some manual pages. Note that some tasks refer to a file `myfile.txt`. No `myfile.txt` is given. You should create your own version for testing.

   (a) Print the 10th through 25th words (including the 10th and 25th words) in `/usr/share/dict/words`. You may take advantage of the fact that the words in this file are each on a separate line. Place your command pipeline in the file `a1q1a.txt`.

   (b) Print the (non-hidden) file/directory names contained by the current directory, in reverse of the normal, alphabetical order. Place your command pipeline in the file `a1q1b.txt`.

   (c) Print the number of lines in the text file `myfile.txt` that do *not* contain the string `cs246` (all in lower-case). Place your command pipeline in the file `a1q1c.txt`.

   (d) Print the first line that contains the string `cs246` (all in lower-case) from the text file `myfile.txt`. Place your command pipeline in the file `a1q1d.txt`.

(e) Print the number of lines in the text file `myfile.txt` that contain the string `linux.student.cs.uwaterloo.ca` where each letter could be in either upper-case or lower-case. (Hint: this is not as obvious as you may think–carefully re-read the special symbols and their meanings first!) Place your command pipeline in the file `a1q1e.txt`.

(f) Print the names of all (non-hidden) files/directories in any *subdirectory* of the current directory whose names end with lower-case `.c` (immediate subdirectories only, not subdirectories of subdirectories). Do not use `find`. (Hint: there's an easy way to do this using only `ls` if you're creative with globbing patterns.) You do not need to worry about what happens if there are no subdirectories or matches found. Place your command pipeline in the file `a1q1f.txt`.

(g) Out of the first 20 lines of `myfile.txt`, how many contain at least one digit i.e. characters '0' to '9'? Place the command pipeline that prints this number in the file `a1q1g.txt`.

(h) Print the names of all (non-hidden) files/directories in the current directory whose names start with the lower-case letter `a`, contain at least one lower-case letter `b`, and end with the lower-case letter `.c`. Place your command pipeline in the file `a1q1h.txt`.

(i) Print a listing, in long form, of all non-hidden entries (files, directories, etc.) in the current directory that are executable by at least one of owner, group, other (the other permission bits could be anything). You may assume that you only need to have the execution bit set to 'x' i.e. you do not need to worry about the special 's' permission value. Do not attempt to solve this problem with `find`. Place your command pipeline in the file `a1q1i.txt`.

(j) Before attempting this subquestion, do some reading (either skim the manual page or have a look on the Web) on the `awk` utility. In particular, be sure you understand the effect of the command:

```
awk '{print $1}' < myfile.txt
```

Give a Linux pipeline that gives a sorted, duplicate-free list of userids currently signed on to the (school) machine the command is running on.
Place your command pipeline in the file `a1q1j.txt`.

2. **9 marks.** For each of the following text search criteria, provide a regular expression that matches the criterion, suitable for use with `egrep`. Your answer in each case should be a text file that contains just the regular expression (i.e., you don't need to include the `egrep` command in your submitted solution), on a single line (again, use `wc` to verify this). If your pattern contains special characters, enclose it in quotes.

(a) Lines that contain both `cs246` and `cs247`, in lower-case.
Place your answer in the file `a1q2a.txt`.

(b) Lines that contain nothing but a single occurrence of laughter, where laughter is defined as a string of the form `Hahahahahahahahahahahahahaha!`, with arbitrarily many `ha`'s. The string must start with `H` and end with `!`. Place your answer in the file `a1q2b.txt`.

(c) Lines that contain nothing but a single occurrence of generalized laughter, which is like ordinary laughter, except that there can be arbitrarily many (but at least one) `a`'s between each pair of consecutive `H/h`'s. (For example: `Haahahaaaa!`) Place your answer in the file `a1q2c.txt`.

(d) Lines that contain at least one lower-case `a` and at least two lower-case `b`'s.
Place your answer in the file `a1q2d.txt`.

(e) Lines consisting of a definition of a single C variable of type `int`, without initialization, optionally preceded by `unsigned`, and optionally followed by any single line `//` comment. Example:

```
int varname; // optional comment
```

You may assume that all of the whitespace in the line consists of space characters (no tabs). You may also assume that `varname` will not be a C keyword (i.e., you do not have to try to check for this with your regular expression). If you don't remember the rules for naming a C variable, please consult `https://www.programiz.com/c-programming/c-variables-constants` Place your answer in the file `a1q2e.txt`.

Hints

- Lecture "Software Testing: Examples" gives an example of how the `produceOutputs` and `runSuite` scripts (Q3-Q5) should be used.
- Lecture "Pipes" gives an example of how to substitute the output of one bash command as the arguments to another bash command. It also explains why you shouldn't use `echo` combined with `cat` to output the contents of a file.
- Lecture "Exercise: good password" gives an example of how to redirect standard output to standard error.
- It is up to the person running the script to specify the path to the program to be tested as part of the command-line argument information. Your script should not make any assumptions about the location of the program executable.

Note that questions 3-5 all follow the same error-handling in terms of validating the command-line arguments for each of `produceOutputs` and `runSuite`. The following sample execution shows you that the error messages are being redirected to the standard error stream, and that the shell script exit code is non-zero.

```
# Invalid number of command-line arguments.
$ ./produceOutputs 1> err1.stdout 2> err1.stderr
$ echo $?
1
$ cat err1.stdout
$ cat err1.stderr
Incorrect number of arguments
$ ./produceOutputs a b c 1> err2.stdout 2> err2.stderr
$ echo $?
1
$ cat err2.stdout
$ cat err2.stderr
Incorrect number of arguments
# Test suite file that cannot be read.
$ ./produceOutputs no_read.args ./my_factorial_correct 1> err3.stdout 2> err3.stderr
$ echo $?
1
$ cat err3.stdout
$ cat err3.stderr
no_read.args is not readable
# Program that cannot be executed.
$ ./produceOutputs test_suite.txt no_execute.sh 1> err4.stdout 2> err4.stderr
$ echo $?
1
$ cat err4.stdout
$ cat err4.stderr
no_execute.sh is not executable
$ ./produceOutputs test_suite.txt ./my_factorial_correct 1> good.stdout 2> good.stderr
$ echo $?
0
$ cat good.stdout
$ cat good.stderr
```

3. **5 marks.** Write a `bash` script called `produceOutputs` that is invoked as follows:

   ```
   ./produceOutputs suite-file program
   ```

   The argument `suite-file` is the name of a file containing a list of filename stems (more details below), and the argument `program` is the name of a program to be run.

   The `produceOutputs` script runs `program` on each test in the test suite and, for each test, creates a file that contains the output produced for that test.

The file `suite-file` contains a list of stems, from which we construct the names of files containing the command-line arguments used by each test. Stems will not contain spaces. For example, suppose our suite file is called `suite.txt` and contains the following entries:

```
test1 test2
reallyBigTest
```

Then our test suite consists of three tests. The first one (`test1`) will use the file `test1.args`. The second one (`test2`) will use the file `test2.args`. The last one (`reallyBigTest`) will use the file `reallyBigTest.args`.

A sample run of `produceOutputs` would be as follows:

```
./produceOutputs suite.txt ./myprogram
```

The script will then run `./myprogram` three times, once for each test specified in `suite.txt`:

- The first time, it will run `./myprogram` with command-line arguments provided to the program from `test1.args`. The results, captured from standard output, will be stored in `test1.out`.
- The second time, it will run `./myprogram` with command-line arguments provided to the program from `test2.args`. The results, captured from standard output, will be stored in `test2.out`.
- The third time, it will run `./myprogram` with command-line arguments provided to the program from `reallyBigTest.args`. The results, captured from standard output, will be stored in `reallyBigTest.out`.

Note that if the test suite contains a stem but a corresponding `.args` file is not present, the program is run without providing any command-line arguments.

Your script must also check for incorrect number of command-line arguments to `produceOutputs` and invalid command-line arguments as described in the preceding "Hints" section. If such an error condition arises, print an informative error message to standard error (the exact message is up to you) and abort the script with a **non-zero** exit status (exact value is up to you). For example, running `produceOutputs` with `.args` or `.out` files with invalid permisisons would produce output and exit codes similar to the following:

```
$ ls -l | egrep "test|no_"
-rw-r----- 1 cs246 cs246     0 May 31 12:41 no_execute.sh
--w-r----- 1 cs246 cs246     0 Aug 11 14:45 no_read.args
-r--r----- 1 cs246 cs246     0 May  4 10:08 no_write.out
-rw-r----- 1 cs246 cs246     0 May  4 10:08 test0.args
-rw-r----- 1 cs246 cs246     2 May  4 10:08 test1.args
-rw-r----- 1 cs246 cs246     3 May  4 10:08 test2.args
-rw-r----- 1 cs246 cs246     2 May  4 10:08 test3.args
--w-r----- 1 cs246 cs246     0 Aug 11 14:45 test4.args
-rw-r----- 1 cs246 cs246    24 May  4 10:14 test_suite.txt
-rw-r----- 1 cs246 cs246    47 Jun  1 15:30 test_suite_V2.txt
# Example of test run without any errors.
$ cat test_suite.txt
test0
test1
test2
test3
$ ./produceOutputs test_suite.txt ./my_factorial_correct 1> good.stdout 2> good.stderr
$ echo $?
0
$ cat good.stdout
$ cat good.stderr
$ ls -l | egrep "test|no_"
-rw-r----- 1 cs246 cs246     0 May 31 12:41 no_execute.sh
--w-r----- 1 cs246 cs246     0 Aug 11 14:45 no_read.args
-r--r----- 1 cs246 cs246     0 May  4 10:08 no_write.out
-rw-r----- 1 cs246 cs246     0 May  4 10:08 test0.args
```

```
-rw-r----- 1 cs246 cs246     0 Aug 11 15:00 test0.out
-rw-r----- 1 cs246 cs246     2 May  4 10:08 test1.args
-rw-r----- 1 cs246 cs246     9 Aug 11 15:00 test1.out
-rw-r----- 1 cs246 cs246     3 May  4 10:08 test2.args
-rw-r----- 1 cs246 cs246    14 Aug 11 15:00 test2.out
-rw-r----- 1 cs246 cs246     2 May  4 10:08 test3.args
-rw-r----- 1 cs246 cs246     7 Aug 11 15:00 test3.out
--w-r----- 1 cs246 cs246     0 Aug 11 14:45 test4.args
-rw-r----- 1 cs246 cs246    24 May  4 10:14 test_suite.txt
-rw-r----- 1 cs246 cs246    47 Jun  1 15:30 test_suite_V2.txt
$ rm *.stdout *.stderr test*.out
$ ls -l | egrep "test|no_"
-rw-r----- 1 cs246 cs246     0 May 31 12:41 no_execute.sh
--w-r----- 1 cs246 cs246     0 Aug 11 14:45 no_read.args
-r--r----- 1 cs246 cs246     0 May  4 10:08 no_write.out
-rw-r----- 1 cs246 cs246     0 May  4 10:08 test0.args
-rw-r----- 1 cs246 cs246     2 May  4 10:08 test1.args
-rw-r----- 1 cs246 cs246     3 May  4 10:08 test2.args
-rw-r----- 1 cs246 cs246     2 May  4 10:08 test3.args
--w-r----- 1 cs246 cs246     0 Aug 11 14:45 test4.args
-rw-r----- 1 cs246 cs246    24 May  4 10:14 test_suite.txt
-rw-r----- 1 cs246 cs246    47 Jun  1 15:30 test_suite_V2.txt
# Example of test run where cannot read from or write to some files.
$ cat test_suite_V2.txt
test0
test1
no_read
no_write
test2
test3
test4
$ ./produceOutputs test_suite_V2.txt ./my_factorial_correct 1> err.stdout 2> err.stderr
$ echo $?
1
$ cat err.stdout
$ cat err.stderr
no_read.args is not readable
no_write.out is not writable
test4.args is not readable
$ ls -l | egrep "test|no_"
-rw-r----- 1 cs246 cs246     0 May 31 12:41 no_execute.sh
--w-r----- 1 cs246 cs246     0 Aug 11 14:45 no_read.args
-r--r----- 1 cs246 cs246     0 May  4 10:08 no_write.out
-rw-r----- 1 cs246 cs246     0 May  4 10:08 test0.args
-rw-r----- 1 cs246 cs246     0 Aug 11 15:00 test0.out
-rw-r----- 1 cs246 cs246     2 May  4 10:08 test1.args
-rw-r----- 1 cs246 cs246     9 Aug 11 15:00 test1.out
-rw-r----- 1 cs246 cs246     3 May  4 10:08 test2.args
-rw-r----- 1 cs246 cs246    14 Aug 11 15:00 test2.out
-rw-r----- 1 cs246 cs246     2 May  4 10:08 test3.args
-rw-r----- 1 cs246 cs246     7 Aug 11 15:00 test3.out
--w-r----- 1 cs246 cs246     0 Aug 11 14:45 test4.args
-rw-r----- 1 cs246 cs246    24 May  4 10:14 test_suite.txt
-rw-r----- 1 cs246 cs246    47 Jun  1 15:30 test_suite_V2.txt
```

You can find an example of the expected output of `produceOutputs` in the file `sample_output.txt` in the directory `a1/codeForStudents/q3` in your Git repository. Using the contents of this directory, running your script as:

```
./produceOutputs test_suite.txt ./my_factorial_correct
```

should produce files named `test0.out`, `test1.out`, `test2.out`, and `test3.out` identical to those provided to you as examples in the same directory.

Note on purpose of this script: This script will be useful in situations where we provide you with a binary version of a program (but not its source code) that you must implement. By creating your own test cases (`.args` files) and then using this script to produce the intended output you will have something to compare with when you implement your own solution (see the next question for how to automate the comparisons). Only the program being run will be able to tell if the provided command-line arguments are correct or not. It is not something that `produceOutputs` can determine.

4. **10 marks.** Create a `bash` script called `runSuite` that is invoked as follows:

```
./runSuite suite-file program
```

The argument `suite-file` is the name of a file containing a list of filename stems (more details below), and the argument `program` is the name of the program to be run.

In summary, the `runSuite` script runs `program` on each test in the test suite (as specified by `suite-file`) and reports on any tests whose output does not match the expected output. Note that if all of the tests are passed, no output will be produced by the script.

The file `suite-file` contains a list of stems, from which we construct the names of files containing the command-line arguments and expected output of each test. Stems will not contain spaces. For example, suppose our suite file is called `suite.txt` and contains the following entries:

```
test1 test2
reallyBigTest
```

Then our test suite consists of three tests. The first one (`test1`) will use the file `test1.args` to hold its command-line arguments, and `test1.out` to hold its expected output. The second one (`test2`) will use the file `test2.args` to hold its command-line arguments, and `test2.out` to hold its expected output. The last one (`reallyBigTest`) will use the file `reallyBigTest.args` to hold its command-line arguments, and `reallyBigTest.out` to hold its expected output.

A sample run of `runSuite` would be as follows:

```
./runSuite suite.txt ./myprogram
```

The script will then run `./myprogram` three times, once for each test specified in `suite.txt`:

- The first time, it will run `./myprogram` with command-line arguments provided to the program from `test1.args`. The results, captured from standard output, will be compared with the contents of `test1.out`.
- The second time, it will run `./myprogram` with command-line arguments provided to the program from `test2.args`. The results, captured from standard output, will be compared with the contents of `test2.out`.
- The third time, it will run `./myprogram` with command-line arguments provided to the program from `reallyBigTest.args`. The results, captured from standard output, will be compared with the contents of `reallyBigTest.out`.

Note that if the test suite contains a stem but a corresponding `.args` file is not present, the program is run without providing any command-line arguments.

If the output of a given test case differs from the expected output, print the following to standard output (assuming test `test2` failed):

```
Test failed: test2
Args:
(contents of test2.args, if it exists)
Expected:
(contents of test2.out)
Actual:
(contents of the actual program output)
```

with the *(contents ...)* lines replaced with actual file contents, *without any changes*, as described. The literal output `Args:` must appear, even if the corresponding file does not exist. Note that there is no whitespace after the colon for each of `Args:`, `Expected:`, and `Actual:` except for the newline character.

An example of such execution would look like:

```
$ ls -l | egrep "test|no_"
--w-r----- 1 cs246 cs246     0 May  4 11:20 no_such_file.args
-rw-r----- 1 cs246 cs246     0 May  4 11:15 test0.out
-rw-r----- 1 cs246 cs246     2 May  4 11:13 test1.args
-rw-r----- 1 cs246 cs246     9 May  4 11:15 test1.out
-rw-r----- 1 cs246 cs246     3 May  4 11:13 test2.args
-rw-r----- 1 cs246 cs246    14 May  4 11:15 test2.out
-rw-r----- 1 cs246 cs246     2 May  4 11:13 test3.args
-rw-r----- 1 cs246 cs246     7 May  4 11:15 test3.out
-rw-r----- 1 cs246 cs246     4 May  4 11:13 test4.args
-rw-r----- 1 cs246 cs246     0 May  4 11:15 test4.out
--w-r----- 1 cs246 cs246     6 May  4 11:13 test5.args
-rw-r----- 1 cs246 cs246    22 May  4 11:15 test5.out
-r--r----- 1 cs246 cs246     8 May  4 11:15 test6.out
-rw-r----- 1 cs246 cs246    30 Aug 11 15:33 test_suite1.txt
-rw-r----- 1 cs246 cs246    55 Aug 11 15:27 test_suite2.txt
-rw-r----- 1 cs246 cs246    55 Aug 11 15:27 test_suite3.txt
# runSuite runs successfully
$ cat test_suite1.txt
test0
test1
test2
test3
test4
$ ./runSuite test_suite1.txt ./my_factorial_buggy 1> t1.stdout 2> t1.stderr
$ echo $?
0
$ cat t1.stdout
Test failed: test3
Args:
0
Expected:
0! = 1
Actual:
0! = 0
$ cat t1.stderr
# runSuite runs unsuccessfully since a .args file exists but cannot be read
$ cat test_suite2.txt
test0
test1
test2
test3
test4
test5
test6
no_such_file
$ ./runSuite test_suite2.txt ./my_factorial_buggy 1> t2.stdout 2> t2.stderr
$ echo $?
1
$ cat t2.stdout
Test failed: test3
Args:
0
```

```
Expected:
0! = 1
Actual:
0! = 0
Test failed: test6
Args:
Expected:
-3! = 1
Actual:
$ cat t2.stderr
test5.args is not readable
File no_such_file.out does not exist or is not readable
# runSuite exits early since a .out file cannot be read
$ cat test_suite3.txt
test0
test1
test2
no_such_file
test3
test4
test5
test6
$ ./runSuite test_suite3.txt ./my_factorial_buggy 1> t2.stdout 2> t2.stderr
$ echo $?
1
$ cat t3.stdout
$ cat t3.stderr
File no_such_file.out does not exist or is not readable
```

Follow these output specifications *very carefully*. You will lose a lot of marks if your output does not match them. If you need to create temporary files, create them in `/tmp`, and use the `mktemp` command to prevent name duplications. **Also be sure to delete any temporary files you create in `/tmp`.**

You can find an example of the expected output of `runSuite` in the file `buggy_output.txt` in the directory `a1/codeForStudents/q4` on your Git repository. Using the contents of this directory, running your script as:

```
./runSuite test_suite1.txt ./my_factorial_buggy
```

should produce an output identical to the example provided to you in the file `buggy_output.txt` in the same directory.

Note: Do **NOT** attempt to compare outputs by storing them in shell variables, and then comparing the shell variables. This is a very bad idea, and it does not scale well to programs that produce large outputs. We reserve the right to deduct marks (on this and all assignments) for bad solutions like this would be.

You can get most of the marks for this question by fulfilling the above requirements. For full marks, your script must also check for the following error conditions:

- incorrect number of command-line arguments to `runSuite` as described in the preceding "Hints" section
- incorrect permissions on the command-line arguments to `runSuite` as described in the preceding "Hints" section
- if a `.args` file exists but isn't readable,
  (a) produce an error message,
  (b) do not run the test,
  (c) do not compare the outputs i.e. just move on to the next test, and
  (d) ensure that the script will eventually terminate with a non-zero exit status
- missing or unreadable `.out` file (for example, the suite file contains an entry `xxx`, but `xxx.out` doesn't exist or is unreadable), in which case print an informative error message to standard error and terminate the script with a **non-zero** exit status.

5. **15 marks.** In this question, you will generalize the `produceOutputs` and `runSuite` scripts that you created in questions 3 and 4. As they are currently written, these scripts cannot be used with programs that take input from standard input. For this problem, you will enhance `produceOutputs` and `runSuite` so that, in addition to (optionally) passing command-line arguments to the program being executed, the program can also be (optionally) provided input from standard input. The interface to the scripts remains the same:

```
./produceOutputs suite.txt ./myprogram
./runSuite suite.txt ./myprogram
```

The format of the suite file remains the same. But now, for each `testname` in the suite file, there might be an optional `testname.in`. If the file `testname.in` is present, then the script (`produceOutputs` or `runSuite`) will run `myprogram` with the contents of `testname.args` passed on the command-line as before and the contents of `testname.in` used for input on `stdin`. If `testname.in` is not present, then the behaviour is almost identical to question 3/4 (see below for a difference in the output).

The output of `runSuite` is changed to now also show the input provided to a test if the test failed. Assuming test `test2` from Q4 failed, the output generated by the updated `runSuite` is as follows:

```
Test failed: test2
Args:
(contents of test2.args, if it exists)
Input:
(contents of test2.in, if it exists)
Expected:
(contents of test2.out)
Actual:
(contents of the actual program output)
```

with the *(contents ...)* lines replaced with actual file contents, as described. The literal output `Args:` and `Input:` must appear, even if the corresponding files do not exist. Note that there is no whitespace after the colon for each of `Args:`, `Input:`, `Expected:`, and `Actual:` except for the newline character.

You can find an example of the expected output of the updated `produceOutputs` and `runSuite` in the directory `a1/codeForStudents/q5/sample_output.txt` on your Git repository. Using the contents of this directory, running your script as:

```
./produceOutputs test_suite1.txt ./my_factorial_correct
```

should produce files named `test0.out`, `test2.out`, `test3.out`, and `test4.out` identical to those provided to you as examples in the same directory.

And using the contents of this directory, running your script as:

```
./runSuite test_suite1.txt ./my_factorial_buggy
```

should produce an output identical to the example provided to you in the file `sample_output.txt` in the same directory.

All error-checking that was required in questions 3 and 4 is required here as well.

(a) Modify `produceOutputs` to handle input from standard input.

(b) Modify `runSuite` to handle input from standard input.

Note: To get this working should require only very small changes to your solution to questions 3 and 4.

Submission

The following files are due at Due Date 1: `a1q1a.txt`, `a1q1b.txt`, `a1q1c.txt`, `a1q1d.txt`, `a1q1e.txt`, `a1q1f.txt`, `a1q1g.txt`, `a1q1h.txt`, `a1q1i.txt`, `a1q1j.txt`, `a1q2a.txt`, `a1q2b.txt`, `a1q2c.txt`, `a1q2d.txt`, `a1q2e.txt`.
The following files are due at Due Date 2: `produceOutputs`, `runSuite`, `produceOutputs`, `runSuite`.