csce215 — UNIX/Linux Fundamentals Spring 2022 — Assignment 5

This assignment is intended to provide some practice and additional content for the material covered in lecture on Monday, February 28. You'll practice using make, first by seeing and using an example makefile and then by creating your own makefile. The assignment is meant to be started in the lab sessions on Wednesday, March 2 and Thursday, March 3. It must be submitted by 11:59pm on Sunday, March 6. A total of 92 points are available.

1 Get started

First, if you are completing this assignment during your scheduled lab time, don't forget to use the QR code or the password to mark yourself present.



Then, just as in previous weeks, start a recbash recording, create a directory for this assignment, and navigate to that new assignment5 directory. Also like previous weeks, there is a collection of files for you to copy into your new assignment5 directory. Make your assignment5 directory the current directory, and then use this command to copy the files there:

```
cp -rv /class/215/assignment5/* .
```



Use recbash to record your terminal session. Create a new directory called assignment5 to use for the tasks below. Copy the files from the shared assignment5 directory to yours.

2 A grade tracker

For the first part of the assignment, navigate to the grades directory. This directory contains some files designed to use make to keep track of grades for an imaginary student in a fictional class. It's meant as a small-scale but non-trivial illustration of how make can be used.

You'll see two kinds of files here:

- Five files with names like hw1.txt, hw2.txt, etc.. Each one contains some feedback from the professor about a homework assignment, along with a grade out of 100 points.
- A makefile with rules to create a file called report.txt, showing the average score and associated letter grade, in several steps.

Take a look at the files here to get an idea of what you're working with.



You'll see rules in the makefile for creating and updating several files. Try it out by giving a command that asks make to create report.txt.

Take a close look at the output from make: It shows a list of commands that are being executed to create report.txt. Specifically, make notices that report.txt has average.txt, count.txt, and letter.txt as prerequisites. So the rules for each of those three will be checked as well, and the recipes for each of them will be executed. Each of these has prerequisites as well, and so on. Make determines which of these commands need to be executed, and in what order.

If you have done this correctly, you should be able to see that several new files have been created, including those prerequisites for report.txt. Notice in the contents of report.txt that this student apparently has a D in the class so far.



Use make to create report.txt. Then use 1s to see a list of all of the files and use cat to show the contents of report.txt. 10 points Let's look more closely at what's happening here. In this makefile, the commands in the recipes are numbered from 1–10, within comments at the end of each line (i.e. after a # character). Notice that several of them use things that we've learned about already this semester (and a couple use some tricks that we have not covered directly):

- Commands 1 and 4–7 are using command substitution, but in the backticks (that is, backward single quotation marks, ``) form instead of the \$() form that we primarily used in Chapter 4. These two forms of command substitution are exactly the same, but backticks are easier to use here because make treats \$ as a special character.
- Commands 1–9 all use redirection of some form, including input redirection and both kinds of output redirection.
- Commands 1, 2, 8, and 9 all use pipes in various ways.
- Commands 5 and 10 use a small trick we haven't seen directly before: A backslash (\) at the end of a line signals that the command continues on the next line. This can be helpful for keeping things clean and readable when we have long commands.
- Command 9 uses the paste command, which we haven't seen before. It works as a sort of inverse to cut. In this case, its job is to combine several lines of input into a single output line, separating each with a given delimiter, in this case +. The effect is that all of the scores are collected on one line with pluses in between exactly the format needed for bc to add them for us! You might be curious to learn more about paste from its manual page.

So, though the recipes may appear complicated, they do not contain much beyond what we've been learning already. Our goal today, however, is to understand how this makefile works to produce the final report.txt. All ten of the recipe commands are correct, and we won't need to modify any of them.

Now let's see make in action when things are updated. Suppose, for example, that we found out that the score for Homework 4 was recorded incorrectly, and that it should have been a 65 instead of a 6. First, use vim to update hw4.txt to show that change.

Now our grade report, report.txt, is out of date, because the numbers and letter grade it shows are based on the old score for Homework 4. What will happen if we use make report.txt to update it? See if you can determine, on your own, which commands will run in that case. Then run make report.txt to see if you were correct and take a look at report.txt to see how the total grade improved. If you've done this correctly, you should have a version of report.txt that shows a number grade of 81.40 and a letter grade of C. Use vim to edit hw4.txt, changing the grade from 6 to 65. Use make to update report.txt to reflect the change in hw4.txt. Use cat to show the updated report.txt. 10 points

Now suppose that the instructor for the course decided to decrease the grade cutoffs. This directory has a small Python program called letter_grade.py, whose job is to figure out what letter grade corresponds to a given percentage grade. The program shows that a 93 is needed for an A, 83 for a B, and so on. Change these to the more typical 90, 80, 70, 60 cutoffs.

Notice that letter_grade.py is listed as a prerequisite for letter.txt. Can you tell what commands will be executed if you do a make after changing letter_grade.py? After you think about it, use make to update report.txt again. Look at the commands it runs to see if you were correct.

Modify letter_grade.py to use cutoffs of 90, 80, 70, and 60. Use make to update report.txt to reflect the change in letter_grade.py. Use cat to show the updated report.txt. 10 points

One last change: Now suppose this student completed another homework assignment, Homework 6, earning 100/100 points. Create a new file called hw6.txt in a format that will work in this system. (If you are not sure what to put in hw6.txt, take a look at Command 8 in the makefile. What format does that command expect?)

Then modify the makefile to account for this new assignment. You can do this by adding hw6.txt directly in the appropriate places, or (perhaps more cleanly) using prerequisites with wildcards to refer to 'all files whose name starts with hw and ends with .txt'. You'll know you've done this correctly if doing make after changing hw6.txt forces report.txt to be updated. The final grade shown in report.txt, after all of these changes, should be 84.50, a B.

When you've finished, show the contents of your hw6.txt, the modified Makefile and the final up-to-date report.txt in your recording using cat.



Create a new file called hw6.txt reflecting a 100/100 score. Change the Makefile to work correctly with this new assignment. Use make to update report.txt to reflect this change, including a dependency in the appropriate place. Use cat to show the hw6.txt, Makefile, and report.txt. 10 points

One last thing before we move on. Notice that this makefile has a 'fake' target called clean. This is a very common pattern for including commands that 'clean up' a directory by removing files that where created using make. Let's clean up, using the command make clean before we leave this directory.



Here's an interesting challenge, but not part of this assignment: Can you modify this system to keep track of your own actual grades in this course?

3 There's no crying in makefiles!

Now it's your turn to create a makefile for yourself. You should skim this section, at least down to the green box on page 8, before starting to work on this part.

Navigate to the wheel directory and take a look. This directory has some files that can produce a research paper called paper.pdf. You will not need to modify any of these given files. However, that document is supposed to show several different types of images and plots, so a few steps are needed to produce it. Here are the details.

• We need a data file called freq.dat, which contains counts of how often each letter appears in a certain input text file. To create it, a Python program called count.py must be executed, reading standard input from ulysses.txt and writing standard output to freq.dat, using a command like this:

```
python3 count.py < ulysses.txt > freq.dat
```

So freq.dat depends on both count.py and ulysses.txt; if either of those files are updated, we should run this command again to update freq.dat.

• We need a plot image called plot.pdf. To create it, a Python program called plot.py must be executed, using a command like this:

python plot.py

Because this program reads from freq.dat, we know that plot.pdf depends on plot.py and freq.dat.

• We need an image file called duck.pdf. To create it, we need to convert the old-school Postscript image format duck.eps to PDF format using the command-line interface of the drawing tool Inkscape, like this:

```
inkscape --export-pdf=duck.pdf duck.eps
```

Thus, duck.pdf depends only on duck.eps.

• We need an image file called stick.pdf. To create it, we need to convert the file stick.fig, which is in the file format used by the classic drawing tool xfig, using a command like this:

fig2dev -L pdf stick.fig stick.pdf

So stick.pdf depends on just one file, stick.fig.

• Finally, to create the completed document paper . pdf, we need to run four commands in sequence:

```
pdflatex paper | grep Output
bibtex paper
pdflatex paper | grep Output
pdflatex paper | grep Output
```

Though it might seem strange, we do indeed need to run pdflatex three times to convert paper.tex to a completed paper.pdf.¹ These greps are simply to make the output shorter, because pdflatex can be loquacious in its output. That bibtex line in the middle is used to create the bibliography, which pulls details from refs.bib. This step pulls together all of the ingredients from the previous steps, so paper.pdf has lots of dependencies: paper.tex, plot.pdf, refs.bib, stick.pdf, and duck.pdf.

¹In case you are curious, these multiple runs are needed because pdflatex makes only one pass through the input file, and possibly leaving notes to its 'future self' about things like what reference number a certain citation should be or what number certain section will have. These can't be known on the first pass, so those extra runs of pdflatex ensure that all of these sorts of references are fully resolved.

Hmmm... a complicated, multi-step process in which many different files depend on each other in various ways? Sounds like a job for make!

Your job is to create a makefile that automates the process described above. Specifically, you should create a file called Makefile. This file does not exist yet, but you can use the command vim Makefile to create a new empty file. Your makefile should have one rule for each of the bullet points above, and should correctly declare the dependencies for each one, and include a correct recipe for creating/updating each target, as described above. The rule for paper.pdf should be the first rule in the makefile, so that make will default to building paper.pdf if we run make without arguments.

How can you be sure that your makefile is correct? A good starting point is to ensure that your makefile does indeed produce a file called paper.pdf. When it does, if you are working physically in the lab, you can use the command evince paper.pdf to see the final result. (You may see some errors from this step about a 'metadata tree', which you can safely ignore.) Beyond that simple check, remember that the touch command can update the modification time of any of these files. You can use touch to help you check whether your answer is correct. In particular, if you touch one of the prerequisites of one of the targets, then running make again to should run **only** the commands that are impacted by changes to that prerequisite.

For example, after successfully building everything, if we touch ulysses.txt and then make again, the output should look something like this:

```
$ make
make: 'paper.pdf' is up to date.
$ touch ulysses.txt
$ make
python3 count.py < ulysses.txt > freq.dat
python plot.py
pdflatex paper | grep Output
Output written on paper.pdf (1 page, 112093 bytes).
bibtex paper
This is BibTeX, Version 0.99d (TeX Live 2021)
The top-level auxiliary file: paper.aux
The style file: plain.bst
Database file #1: refs.bib
pdflatex paper | grep Output
Output written on paper.pdf (1 page, 112093 bytes).
pdflatex paper | grep Output
Output written on paper.pdf (1 page, 112093 bytes).
```

Notice that all of the steps that rely, either directly or indirectly, on ulysses.txt are re-

run. But, for example, the neither the inkscape nor fig2dev commands are re-run here, because the prerequisites for those rules have not been touched.

Your makefile should also include a clean target, which will remove all of the files generated by this process, both the ones that were created on purpose as described above, and some others that are created as 'side effects', such as the log files from pdflatex and bibtex. Specifically, if we type make clean, these files should be deleted:

That's it. Just make a correct makefile for this scenario and check to ensure that it is working correctly. We'll evaluate this section entirely based on the correctness of your final Makefile.

> Use vim in your terminal to create a Makefile as described above. If the command make is given without any arguments, it should successfully make paper.pdf. The makefile must correctly encode the dependencies listed in the bullet points that start on page 5, and include exactly the commands listed above in the appropriate recipes. It must also have a clean target whose recipe removes the generated files listed above. Use cat to show your completed Makefile in the recording. 45 points

4 It's Over!

When you've completed the tasks above, use Ctrl-D or exit to terminate the recording. Optionally, inspect its transcript it ensure that it shows all of the steps mentioned in the green boxes.



