Week 03

ssh, grep, regexes, if, and bash scripts

February 3, 2021

Tonight we'll look at grep a bit more, we'll learn what a regex is, youll see how if statements work in bash, we'll see a tidy way to write bash commands, and you'll use a cool tool called ssh.

1 Review of the commands

By this point we've learned a whole bunch of basic command line programs you can use on Linux. Spend a few minutes reviewing them in class.

2 Vim Recap

There is alot to know about Vim, it's a great editor.

The main thing to know is

```
root@machine$ vim filename # this opens a file called "filename"
```

When you are in vim, you need to know two things to get started:

- When you first launch vim you are in NORMAL mode. You cannot type into your file yet.
- Hit "i" on the keyboard to enter INSERT mode. In INSERT mode you can type in your file.
- When you are done working on your file, hit ESC. This puts you back in NORMAL mode
- In NORMAL mode you can save and quit, save, or just quit.
- To save and quit type ":wq".
- To quit without saving type ":q!"
- To save without quitting type ":w". You are still in NORMAL mode. Remember if you want to go back to editing the file, type "i". This will put you back in INSERT mode.

3 Review SSH

We won't go too in depth. If you want to go deep, check out Open SSH Mastery, an awesome book by Michael Lukas.

Here's what I'll tell you though:

- Open a terminal and type "ssh-keygen" to generate a key pair.

- The keypair will be stored in the .ssh folder in your home directory.
- Guard the id_rsa with your life. If anyone gets that file, they can hack your servers.
- If you want to connect to a server with ssh (without using a password), copy the contents of the id_rsa.pub file to a line in the .ssh/authorized_keys file.
- With respect to the last point - I typically put 3 keys on my servers. One for my mac, one for my windows pc, and one for my linux laptop. That way I can connect to my server from any pc in my house. If you use multiple computers, you might want to generate a key pair on each one, and put all the public keys on your server.
- To connect with ssh, do this: *ssh usernamemachineIPaddress*.
- So far we are always connecting as a user named "root". This will change in the coming weeks.

Class Example
Have everyone in class send me their pub keys. Put the pub keys on my server. Show they can all connect. This might be chaos with too many students. In the case of many students, just do this with a handful

4 How To Download Files off the Internet

Two good choices: **wget** and **curl**.

4.1 wget

If you want to download this file:

```
https://www.cc.gatech.edu/~spencer/poems/woods.txt
```

Then you use this command:

```
root@machine$ wget https://www.cc.gatech.edu/~spencer/poems/woods.txt
root@machine$ ls
# you will see a file called woods.txt
```

If you would like to rename the file from the name it has on the webserver you downloaded from, use the **O** flag.

```
root@machine$ wget -O robertFrost.poem
   https://www.cc.gatech.edu/~spencer/poems/woods.txt
root@machine$ ls
# you will see a file called robertFrost.poem
```

Class Example
Go to the url above in a browser and show that there is a text file on the internet. Run wget and show it downloads to the computer.

4.2 cURL

You can also use a tool called curl. cURL is a very powerful tool that we will learn in depth later in the semester. I'll just tell you that it exists now to plant a seed in your memory.

Here's how you download a file with curl:

```
root@machine$ curl
   https://www.cc.gatech.edu/~spencer/poems/woods.txt --output
frost.poem
root@machine$ ls
frost.poem
```

Now that's enough about downloading files, let's talk about grep.

5 grep

Is for searching for strings in files. Do you want to know if the string "woods" is in a file or set of files? **Use grep**. Do you want to know how many times a particular number appears in a file? **Use grep**. Do you want to do something more crazy like find if the word

```
goal
OR
gooaal
OR
gooooaaaal
OR
goooooooooooooooaaaaaaaaaaaaaaaaal
```

(or any other drawn out spelling) exists in a file? **Use grep**.

Grep is commonly used with pipes. Like:

```
root@machine$ history | grep wget
# this will list all the wget commands you ran.
```

It can do alot more than this simple trick though.

5.1 Commonly used grep flags

- **-i** Means "case insensitive". We can match the search pattern in upper case or lowercase.
- **-w** Means match only the whole word.
- **-v** Inverse grep. Instead of matching, search for lines that DONT match.
- **-o** Usually grep returns the whole matching line. this returns just the matching THING from the line.
- **-n** Display not only the matching line, but also the line number.
- **-r** Recursive. Search in subdirectories.

- **-A** Include lines AFTER the matching line in addition to the matching line.
- **-B** Include lines BEFORE the matching line in addition to the matching line.
- **-C** Include lines both BEFORE and AFTER the matching line.

```
root@machine$ cat example_data.txt
hello world of grep
my favorite numbers are 1 2 3 4 5
HelLo WoRld of Greppp
root@machine$ grep "hello" example_data.txt
hello world of grep
root@machine$ grep -i "hello" example_data.txt
hello world of grep
HelLo WoRld of Greppp
root@machine$ # the next command wont match the last line because
root@machine$ # the word grep != Greppp
root@machine$ grep -iw "grep" example_data.txt
hello world of grep
root@machine$ grep -v "of" example_data.txt
my favorite numbers are 1 2 3 4 5
root@machine$ # the next line only matches the 2, not the whole line
root@machine$ grep -o "2" example_data.txt
2
root@machine$ grep -nw hello example_data.txt
1:hello
```

The **r**,**A**,**B**, and **C** flags are important, lets make sure too look back at them another time.

5.2 Regular expressions and grep

Some people, when confronted with a problem, think "I know, I'll use regular expressions". Now they have two problems.

Unknown source. // Programming wisdom and humor.

Grep stands for
Global **R**egular **E**xpression **P**rint.
Keep an open mind as we look at regular expressions in grep. I don't have all of this committed to memory - I just know generally what they are for and know what I need to look up when I'm stuck.
Let's consider this file:

```
root@machine$ cat demo_file.txt
1
11
123
```

```
18888.111
hlo
hello
helllo
he1llo
```

Let's look at regular expressions to see if we can extract certain bits of information.

```
$ grep "[0-9]" demo_file.txt # match any line with a digit from 0-9
1
11
123
18888.111
he1llo
$ grep "[0-9]\+" demo_file.txt # match any line with a series of
    digits
1
11
123
18888.111
he1lllo
$ grep -n "[0-9]\+" demo_file.txt # match just the series of digits
    (-o flag)
1
11
123
18888
111
1
$#this next one is tricky.
$#match "h"
$#followed by an optional e ( e\? )
$#followed by any number of 1s and ls ([l1]\+)
$#followed by an obligatory o
$grep "he\?[l1]\+o" demo_file.txt
hlo
hello
helllo
he1llo
$grep "he...o" demo_file.txt # find he followed by 3 anythings
    followed by o
helllo
he1llo
$ grep "^h" demo_file.txt # any line starting with h
hlo
hello
helllo
he1llo
$ grep "[0-9]$" demo_file.txt # any line ending in a digit.
1
11
123
18888.111
```

```
$ grep "1\{2,3\}" demo_file.txt # any line with between 2 and 3 ones.
11
123
18888.111
$ # this next one is tricky because it uses the caret with a
    different meaning.
$ grep "[^0-9]" demo_file.txt # any line with no digit in it
hlo
hello
helllo
```

These are the important characters you need to know:

- '.'
- '\?'
- '\+'
- '\{'
- '\}'
- '\|'

What to know:

1. The period (.) matches any single character.
2. \? means that the preceding item is optional, and if found, will be matched at the most, once.
3. * means that the preceding item will be matched zero or more times.
4. + means the preceding item will be matched one or more times.
5. n means the preceding item is matched exactly n times, while n, means the item is matched n or more times. n,m means that the preceding item is matched at least n times, but not more than m times. ,m means that the preceding item is matched, at the most, m times.

Some more syntax:

1. ^(Caret) = match expression at the start of a line, as in ^A will match an A at the beginning of a line.
2. $(Dollarsign) = matchexpressionattheendofaline, asinA.$
3. \(Back Slash) = turn off te special meaning of the next character, as in $\hat{}$

   (Brackets) = match any one of the enclosed characters, as in [aeiou]. Use Hyphen "-" for a range, as in [0-9].
4. . (Period) = match a single character of any value, except end of line.
5. * (Asterisk) = match zero or more of the preceding character or expression.

6. {x,y} = match x to y occurrences of the preceding.
7. {x} = match exactly x occurrences of the preceding.
8. {x,} = match x or more occurrences of the preceding.
9. `[\^ ]    =   match any one character except those enclosed in [ ], as in [\^ 0-9].`

That's it! So, given the file a.txt (see this directory)
We can do the following

```
$ grep "a" a.txt # Find lines with an a
$ grep "a\?" a.txt # find lines with an optional a.
$ grep "a?" a.txt # find lines containing a?
$ grep "a\+" a.txt # find lines with 1 or more "a"s
$ grep "a+" a.txt # find lines containing "a+"
$ grep "a$" a.txt # find lines that end with a
$ grep "[0-9]$" a.txt # find lines that end with a number
$ grep "^[a-zA-Z]$" a.txt # find lines with one letter.
$ grep "a\{2,\}" a.txt # find lines with 2 or more "a"s
```

Regular expressions are usually called **regexes** (rej - exes). They are useful for searching through text for patterns. Regular expressions are so useful they are in nd nd nd virtually any other language you might become interested in.

In Linux land you will use regular expressions with tools like:

1. awk
2. sed
3. grep
4. perl

Regular expressions are a fundamental part of what grep is for.

5.3 grep for things that aren't English or Numbers

English isn't the only written language in the world.

What if you are analyzing a piece of text in Brazilian Portuguese, and you want to grep for the letter "Ãğ"? How will you grep for that with your English language keyboard? The only way is to take a deep dive into Text Encodings. You'll need to learn what the hexadecimal or binary representation of the letter is, and you can use grep like that.

This also works for emojis! You can grep for emojis!

Later in the semester we will look into text encodings and how they affect the various Linux text-processing tools.

We'll revisit this later.

6 Bash Scripting

We should close tonight by recognizing that bash is a programming language and that you can write long scripts with bash. Up until this point we've only been using bash interactively on the command line. Let's take a moment to look at

YOU CAN IGNORE The NEXT SECTIONS for NOW

Just know the following:

1. Bash is a programming language
2. Bash has loops
3. Bash has if statements
4. Bash has arrays
5. Bash has variables

We will visit this at a later date when it's more important.

Just do this:

```
$cat myfirstscript.sh
#!/bin/bash

mkdir -p a/b/c
echo "this is file a in diretory a" > a/a.txt
echo "this is file b in directory a/b" > a/b/b.txt
echo "this is file c in directory a/b/c" > a/b/c/c.txt
$ chmod +x myfirstscript.sh
$ ./myfirstscript.sh
$ls a
a.txt
$ls a/b
b.txt
$ls a/b/c
c.txt
```

Ignore the following sections if you are in Spring 2021 lecture.

6.1 Introduction to if in bash

Lots of programming languages have 'ifs'. Java has them, python, C, C++, javascript, and beyond. So does bash. It is a standard thing to do in programming to check if a variable equals something.

The bash syntax for if is

```
if CONDITION
then
 command
fi
```

The tricky bit is getting the CONDITION part right, because it is different in different situations, with different data types. I'll show you a little bit about it right now. And, as always, I'll encourage you to go read more if you want the full story. A good link is somewhere down below.

One more useful piece of information is that bash generally interprets values as strings, unless they can be used as numbers, in which case it assumes they are numbers. https://www.tldp.org/LDP/abs/html/untyped.html.

Comparison operators for numbers in bash are:

- -eq
- -ne
- -gt
- -ge
- etc.

Comparison operators for strings are:

- =
- !=
- etc.

for more information see here `https://www.tldp.org/LDP/abs/html/comparison-ops.html`

```
melvyn@thinkpad$ cat myFirstScript.sh
#!/bin/bash


#x1 and x2 are integers, though they could also be interpreted as
    strings.
x1=1
x2=2
if [ $x1 -lt $x2 ]
then
    echo "$x1 < $x2"
else
    echo "$x2 <= $x1"
fi
melvyn@thinkpad$ bash myFirstScript.sh
1 < 2
```

Then this script will fail, because you are using an arithmetic comp on strings:

```
melvyn@laptop$ cat myBadScript.sh
x1=1a
x2=2a
if [ $x1 -lt $x2 ]
then
    echo "$x1 < $x2"
else
    echo "$x2 <= $x1"
```

```
fi
melvyn@laptop$ bash myBadScript.sh
# error
```

You can easily fix this with:

Listing 1: option 1

```
#!/bin/bash

x1=1a
x2=2a

if [[ "$x1" < "$x2" ]]
then
    echo "$x1 < $x2"
else
    echo "$x2 <= $x1"
fi
```

OR

Listing 2: option 2 for if with strings

```
#!/bin/bash

x1=1a
x2=2a

if [ "$x1" \< "$x2" ]
then
    echo "$x1 < $x2"
else
    echo "$x2 <= $x1"
fi
```

Tip: The old bash advice is to double quote all variables in bash to make sure they are interpreted as a single value.

7 Fluff

In case extra time, I solved a little problem today using tools you know.

I'm programming a little AVR microcontroller, and the tool you use to program it is avrdude. If you type

```
melvyn@laptop$ avrdude -c ?
```

it will list all the available hardware programmers out there. I wanted ot see if my mkii was supported by this tool so I ran

```
melvyn@laptop$ avrdude -c ? | grep mkii
```

No results. Well I didn't know if avrdude uses upper or lower case, so I just said like this:

```
melvyn@laptop$ avrdude -c ? | grep -i mkii
```

Still no results. Weird. But then I thought, how many types are there?

```
melvyn@laptop$ avrdude -c ? | wc -l
```

No results. But the command definitely outputs results. Do you know how to fix the above commands?

ANSWER: avrdude -c ? writes to stderr, not stdout (for some reason). So I just route my stderr to stdout and then use the piped commands (pipes work on stdout, not stderr:

```
melvyn@laptop$ avrdude -c ? 2>&1 | grep -i mkii
# It comes out!
```