Lecture Notes

Beginning at changing access permissions and discusses more with the shell steps and that process.

-rw-----

each of these can be associated with an octal number which goes 1-7

read is worth 4

write is worth 2

execute is worth 1

example:

chmod 660 fruits

660

(4+2) (4+2)

so we have read+write read+write zero

1 1 0

which is 6

444 is read-read-read

1 0 0 -> 4

octal	read	write	execute
7	1	1	1
6	1	1	0
5	1	0	1
4	1	0	0
3	0	1	1
2	0	1	0
1	0	0	1
0	0	0	0

syntax:

u - owner

g - owning group

o - all others

then an operator +, -, =

then r, w, or x

- **User classes**
  - **User** - owner
  - **Group** - users who are members of the same group
  - **Others** - users are neither the owner nor members of the owner's group
- **Modes:**
  - "r" - read a fil
  - "w" - write, modify, or delete a file
  - "x" - execute a file

Examples:

```
fox01:~$ chmod u+x,go= whoson.bash
#changes execute for the owner, stripping permissions
fox01:~$ chmod a-x whoson.bash #removes permissions for all
                               #a-x same as ugo
```

make sure you're prefixing with a leading 0

```
fox01:~$ chmod 600 whoson.bash
#if you just gave it 600 it interprets it as a decimal number in
languages like C
fox01:~$ chmod 0600 whoson.bash
fox01:~$ chmod u+x whoson.bash
-rwx------ 1 namehere faculty 27 Jan 27 13.38
fox01:~$ chmod a+rx whoson.bash
#execute for everybody
-rwxr-xr-x 1 namehere faculty 27 Jan 27 13.38
fox01:~$ ls -d ~
drwx---- 16 namehere faculty 560k Jan 29 09:51 /home/namehere
fox01:~$ chmod g+rw name
$ ls -l name
-rw-rw-- 1 namehere faculty 27 Apr 23 1:33 names
#this is for if we want read-write permissions for the group
```

Aliases

An alias is a short name for a command which may include parameters.

- **Bash:**
  - alias aliasName='value'
  - alias aliasName="value"
- **Tcsh:**
  - alias aliasName 'value'
  - alias aliasName "value"
  -

Note: that surrounding the value with double quotation marks causes any variable references to be substituted when the alias is created. With single quotation marks, any embedded variables would be substituted when the alias is referenced. If you have space around this, the shell's going to think you're trying to run a command named greet with two parameters 'equals' and 'hello'

```
fox01:~$ greet = hello
No command 'greet' found, did you mean:
fox01:~$ greet=hello
fox01:~$ echo $greet
hello
fox01:~$ alias eek="echo $greet"
#Remembering our 7 steps, the shell won't even attempt to resolve
whether or not 'alias' even exists until the parsing process finishes.
```

```
It first sees $greet within the double quotes, and take that current
value of $greet, put as a string literal in the definition of alias
fox01:~$ alias eek
alias eek='echo hello'
fox01:~$ eek
hello
fox01:~$ greet=xyz
fox01:~$ eek
hello
#doesn't matter if you change the value of greet
fox01:~$ alias eek='echo $greet'
#single quotes specify a string literal
fox01:~$ alias eek
alias eek='echo $greet'
#variable reference is now embedded in it

fox01:~$ xyz
#if you press enter, what will the shell do?
do you think it's going to check the current directory to see if
there's a program called xyz?
What if there's an xyz system utility?
What if there's 100 executables named xyz spread across the file
system? which of the 100 will it run?
that's what the Shell steps aim to clarify
Output:
No command 'xyz' found, did you mean:
. . .

fox01:~$ /bin/echo hello world
#the shell knows I want to run hello world in /bin/echo
fox01:~$ ../echo hello world
#this also works because we're telling it to look in the parent
directory

fox01:~$ alias hw="/bin/echo hello world"
fox01:~$ hw
#replaces the entire command line with "/bin/echo hello world"
#if it's an alias expanded, you start over again with the steps
hello world

If you want to see if something is a built-in function you can use the
'type' command
fox01:~$ type echo
echo is a shell builtin
fox01:~$ type source
source is a shell builtin
fox01:~$ type ls
ls is an aliased to 'ls -larth --color --group-directories-first'
fox01:~$ type cat
cat is hashed (/bin/cat)
```

```
fox01:~$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:..etc
Goes through everything to see if there's an executable named xyz
You need to include the current directory at the very end
"which" locates the paths of executable files
fox01:~$ which cat
/bin/cat
fox01:~$ which ls
/bin/ls
fox01:~$ which echo
/bin/echo

In this example we have a script named cat as well
fox01:~$ vi cat
fox01:~$ alias tiger=cat
fox01:~$ alias myCat=./cat
fox01:~$ alias myCat tiger
alias myCat='./cat'
alias tiger='cat'
fox01:~$ cat cat
echo "meow"
cat
fox01:~$ cat < numbers.txt
one hundred one
one hundred two
one hundred three
. . .
fox01:~$ which cat
/bin/cat
fox01:~$ ./cat < numbers.txt
#this stops at step 1 because of the slash
meow
one hundred one
one hundred two
. . .
#Runs both versions of cat
fox01:~$ ./cat
#This runs in the current directory, but will block and wait for input
meow
line one
line two
. . .

In the script if we did
echo "meow"
./cat
It'll create a recursive loop (bad)
```

Pipelining
Pipelining lets you take the output of one command and use it as the input for another command. The | symbol connects them so you can chain commands together.

```
fox01:~$ ls | more
```
ls (which lists files) and more (shows output page by page)

```
fox01:~$ who | sort
#This will list who's on and sort them
anl176 pts/13 timestamp
gjh148 pts/2 timestamp
jaq088 pts/9 timestamp
jsherett pts/3 timestamp
nsd690 pts/10 timestamp
ssilvestro pts/14 timestamp
fox01:~$ who | sort -r
#This will do the above, but reverse the list
fox01:~$ who | sort | tac | \grep -v ssilvestro
#This does the above, tac also prints files in reverse as -r does, and
it removes the instance of Silvestro in the list printed.
fox01:~$ who | sort | tac \grep -v ssilvestro | wc -l
10
#This is "how many users are online besides myself"
```

File name patterns

?
Matches any single character. For example, p? which matches p1, p2, and p3, but would not match p1.h
p? specifically means you're looking for a two character name, so p* or p- would work, but this is why p1.h wouldn't.

"*"
Matches from zero to many of any characters. For example,p* would match p1, p1.h, p1main.c

[list]
Matches one character to any of the characters listed within the brackets. For convenience, range abbreviations can be used (e.g [a-f],[0-9]) We can also do [a-f,A-F,0-9], [aeiouy], [0-9xyz]. We do have to specify if we want lower or upper cases. So for both, we'd put [aeiouyAEIOUY].
If we want to include a hyphen it has to be at the beginning or end like: [-a-z] or [a-z-], brackets []a-z].

[^list]
This matches one character if it isn't listed within the brackets. [^a-z]

```
fox01:~$ ls {p1I,p10}*
This is going to look for all the files that start with p1I or p10
->It creates two effects of p1I* p10*
fox01:~$ echo x{red,blue}y
xredy xbluey
fox01:~$ echo x{red,blue}{car,truck}y
xredcary xredtrucky xbluecary xbluetrucky

Basically it distributes across both sets
fox01:~$ ls {p1I,p10}*
-rw---- 1 namehere faculty 0 Aug 1 2024 p1Out.txt
-rw---- 1 namehere faculty 0 Aug 1 2024 p1Input.txt
-rw---- 1 namehere faculty 0 Aug 1 2024 p1OutExtra.txt

fox01:~$ ls [a-z][a-0][abc]*.o
-rw---- 1 namehere faculty 0 Aug 1 2024 p1abc123.o
```

echo p1*.*
Matches anything that has p1 and a “.”
like p1test.txt.bak.final.raw.bin
Just saying there has to be a dot in the rest of the file, and will match all dots.

echo p1*.?
Looks for a filename that begins with p1, dot, and ends with one character as the filetype.
So p1abc123.o or p1abc123.c would be the output.