A Big Step

Shell Scripts, I/O Redirection, Ownership and Permission Concepts, and Binary Numbers

What a shell really does

- Here is the scoop on shells.
- A shell is a program that displays a prompt (like '**$**') and then waits for you to type a command. When you type the command, the shell runs ***the file whose name is that command***. When the shell finishes running that file, it displays the prompt and starts this all over again.
- Of course it is a bit more complicated to do this than it sounds, but this is the gist of it.
- The key point right now is that ***commands are just files*** (well, not always.)

About commands

- A command is (usually) just a file that is executed by the shell when you type its name. For example, **`ls`** is really the file **`/bin/ls`**.
- You can find the location of a command with the **`whereis`** (a

```
$ whereis ls
/bin/ls
$
```

- Not all commands are files; some are built into the shell itself. For example **`cd`** is not a file. It is part of **`bash`**. When you type **`cd`**, **`bash`** itself changes your working directory.

Executable files

- If a command is just a file, then how can it be run?
- First of all, the file has to contain a ***program*** – a sequence of instructions to be executed. It cannot contain your best friend's favorite song or a photo.
- But that is not enough. UNIX protects you and itself by requiring that if a file is supposed to be executable, then its "***execute-bit***" must be turned on for every set of users that should be able to execute it. Remember from an earlier lesson that the execute-bit is part of the file mode, and that there are three such sets of users: ***user***, ***group***, and ***others***.
- So how do you tell UNIX to turn on that bit?

Making files executable

- To ***change the mode*** of the file, you use the **`chmod`** command. **`chmod`** is a mnemonic:

  `chmod`

  `ch` for ***change***, **`mod`** for ***mode***

- To make a file named ***`filename`*** executable for everyone (user, group, and others), type

```
chmod +x filename
```

- To make it executable for the user and turn it off for group,

```
chmod u+x,g-x filename
```

- We will learn more about **`chmod`** later.

Digression: Output redirection

- All commands in UNIX display their output, i.e., their results, on your terminal. You do not have to do anything special for this to happen. For example, when you type **`ls`**, it displays the list of files in your terminal window.
- In an earlier lesson, you saw that UNIX provides a way to ***redirect that output to a file*** instead using the ***output redirection operator***, **`>`**, as in

```
$ ls > currentdir_contents
```

  which puts the output of **`ls`** into a file named **`currentdir_contents`** in the current working directory.

Creating files using **>**

- It should be obvious that this is a convenient way to create files, especially with the **echo** command. Assuming that the file **file1** does not exist in the working directory,

```
$ echo “Some days you're a bug.” > file1
```

puts the text “ **Some days you're a bug.** “ in **file1** (without the quotes):

```
$ cat file1
Some days you're a bug.
$
```

Appending to files using **>>**

- If a file already exists and you want the output of a command to be *appended* (added) to the end of the file, you can use the output redirection append operator, **`>>`**. This puts the output after the last line of the file, as in

```
$ echo “Other days a windshield.” >> file1
```

adds the text “"**`Other days a windshield.`**” to the end of **`file1`** :

```
$ cat file1
Some days you're a bug.
Other days a windshield.
$
```

About quotes ' “ in commands

- It is usually safest to enclose the argument to **`echo`** in single quotes: `' '` . The reason is partly explained below. But if the argument itself contains a single quote, then you must enclose it in double quotes: **`“it's”`**.
- The shell treats certain characters as special, but the single quotes hide their specialness from the shell. Double quotes only hide certain characters and not others. Which ones are hidden by single quotes but not double quotes is a topic for a future lesson.

Creating a simple script

- We can use these operators to create files without using a text editor. We will create our first shell script this way.
- Try typing these three commands:

```
$ echo '#!/bin/bash'    > first_script
$ echo 'hello world! ' >> first_script
$ cat first_script
#!/bin/bash
echo hello world!
$
```

- The file **`first_script`** has two lines, which I now explain.

The script explained

- The first line tells the shell to run the ***interpreter*** **`/bin/bash`** using the rest of the file as its input. In effect, it says, "I am a bash script". ***You must put it at the top of every*** **`bash`** ***script***.
- In general, a script must start with the two characters **`#!`** followed by the absolute pathname of the script interpreter. A Perl script would have the first line **`#!/usr/bin/perl`** on our system because the **`perl`** command is located in **`/usr/bin`**. (Later I will show you how to do this when you do not know where the interpreter is.)
- The second line is the **`echo`** command. It simply displays the words "**`hello world!`**" on the standard output.

Scripts must be executable

- If you try running this script this without making the file executable, all you will get is an error message like this:

```
-bash: ./first_script: Permission denied
```

- We must make it executable:

```
    chmod  +x first_script
```

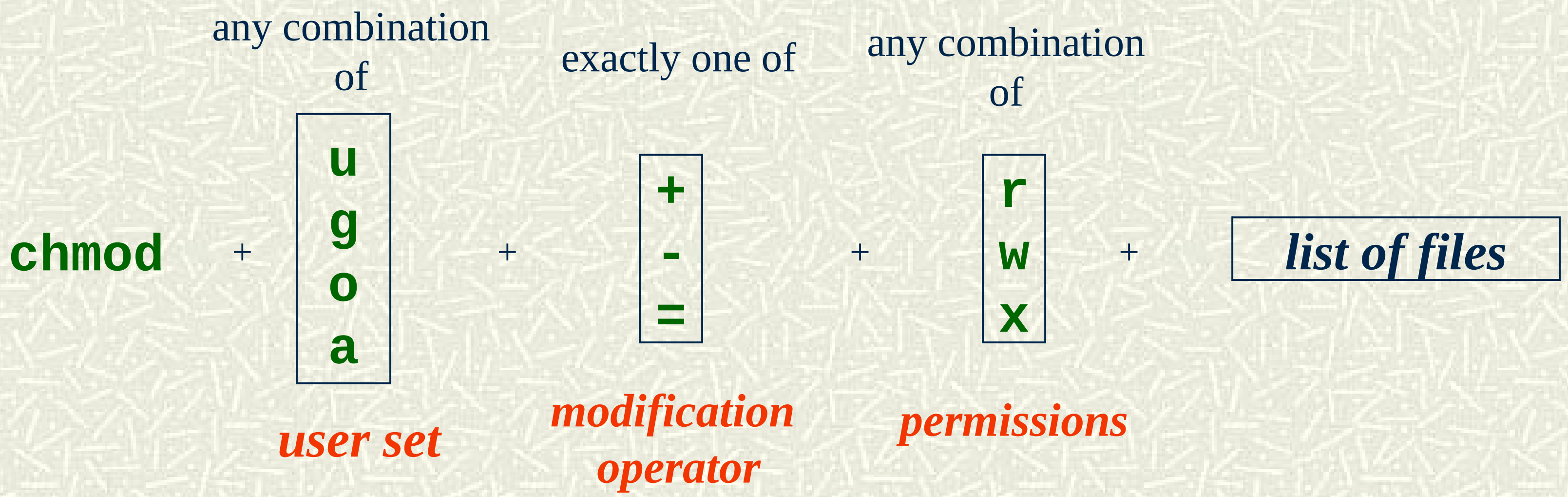
Then when we type `./first_script` we will see

```
    hello world!
```

chmod revisited

- **chmod** is given a group of ***flags*** and a list of files and it gives those files the mode defined by the flags.

  The flags consist of 3 parts, the ***user set,*** the ***modification operator,*** and the ***permissions*** to modify.



Some **chmod** details

- The **u**,**g**,**o**, and **a** stand for ***user***, ***group***, ***others***, and ***all*** respectively. (I remember **ugo** by remembering Victor Hugo -- Hugo sounds like ugo, user-group-other. Pretty lame but it works.)
- **+** adds a permission, **-** removes it, and **=** sets the mode to exactly the ones specified.
- **r**,**w**, and **x** are ***read***, ***write***, and ***execute*** respectively.
- ***You cannot put space anywhere between the parts***.

`chmod` examples

```
chmod u+rw  foo
```

adds ***read*** and ***write*** permission for ***owner*** of `foo`

```
chmod g+rx foo
```

adds ***read*** and ***execute*** permission for ***group*** of `foo`

```
chmod go-w foo
```

removes ***write*** permission for everyone in ***group*** and ***others*** for `foo`. (So only owner can modify the file.)

```
chmod u=rwx  foo
```

sets the mode to `rwx------` for `foo`

More about the **chmod** command

- If the set of users flag is omitted (i.e., ugo), the permission flags are applied to all three sets, user, group, and others.

```
chmod +x  foo
```

adds ***execute*** permission for ***everyone***

```
chmod -w foo
```

removes ***write*** permission for everyone, including the user (owner), who will not be able to modify **foo** after this without changing the mode.

Advice about **chmod**

- In short, you can create any possible combination of permissions on any file using the **chmod** command. (How many possible combinations are there?)
- Usually the owner has the greatest permission, then the group, then everyone else. Usually write permission is restricted to the owner alone, or to the group if it is a file being cooperatively written. If it is executable, usually everyone has that permission set. Lastly, read permission is given to everyone only when it is a document intended for global consumption.

Back to the script

- Even if `first_script` is executable, it may not run; we will get an error if it is not also readable for us, because the shell has to read this file in order to execute it.
- Therefore you must make sure that it is both readable and executable. (It will be readable for its owner unless you went out of your way to make it unreadable.)
- Files that contain shell commands and are executable are called ***shell scripts***. They are like DOS batch files.

Directory permissions

- You also use **`chmod`** to change the mode of directories, but you have to understand how UNIX uses the read, write, and execute bits for directories.

  After all, what does it mean to ***read a directory***, or to ***write or execute a directory***?

Reading a directory

- Read permission on a directory gives the user the ability to view the directory file's contents, which, you should recall, is a table containing the names of the files in that directory and their index numbers.
- If read permission is turned off, you cannot list the contents of that directory, i.e., the names of the files in that directory. **`ls`** will display an error message.

Writing a directory

- Write permission on a directory gives the user the ability to modify the directory file. Since adding a file to a directory or removing a file from a directory changes the directory's contents, you cannot add or remove files from a directory unless you have write permission on that directory.
- Be careful – you can modify files themselves even though you cannot delete them or add new ones. You can completely empty a file but you cannot remove it from the directory if you do not have write permission on the directory.

Executing a directory?

- This should sound strange to you. You cannot execute a directory, but UNIX uses the execute bit to control whether you can "**cd**" into a directory:
- If you do not have execute permission for a directory, then you cannot **cd** into it. If that directory is any part of a pathname to a file you want to open or execute for any reason, you will not be able to access that file, because you need to "get through that directory" to reach it.
- Therefore, you need to have execute permission for each directory on the path to a file you want to execute. This also applies to programs that you try to run to open that file.

More about execute permission

- A command such as

```
ls –l testdir
```

must do two things to display results. First, it must read the directory file, and for each filename it finds there, it must use the index number to find the file in the file system. When it finds the file in the file system, it has to open the part of the file where its properties are stored. This will fail if you do not have execute permission, because it requires opening the file, which cannot be done without execute permission on every directory in its absolute pathname.

An experiment

- Create a directory **temp** and a file named **foo** inside it.

```
$ mkdir temp; cd temp; touch foo; cd ..
```

  (You can put multiple commands on a line if you separate them with semi-colons.)

- Type **ls –l temp** to see the properties of **foo**.
- Now remove execute permission from **temp**, type **ls –l temp** and observe the results:

```
$ chmod -x temp; ls –l temp
ls -l temp
total 0
?----------  ? ? ? ?           ? foo
```

The time has come, the Walrus said, …

- You cannot move forward without an understanding of the binary number system. The decimal number system has ten digits, 0,1,2,3,…, 9 but the binary has two, just 0 and 1.
- It is nonetheless possible to write all possible numbers in this system. Here are the first 8 numbers in binary and decimal:

binary:	**0,**	**1,**	**10,**	**11,**	**100,**	**101,**	**110,**	**111**
decimal:	**0,**	**1,**	**2,**	**3,**	**4,**	**5,**	**6,**	**7**

You can figure out the sequence if you know how to add 1 to a number in the binary number system.

Binary numerals

- In the decimal system, there is the ones' place, the tens' place, the hundreds' place, going up by powers of ten, and so on.
- In the binary system, there is the ones' place, the twos' place, the fours' place, the eights' place and so on, going up by powers of two. Reading from ***right to left***, the number

  `11001`

  has 1 one, 0 twos, 0 fours, 1 eight, and 1 sixteen, for a total of 1+8+16=25 in base ten. So **`11001`** in binary is 25 in decimal.

Finding the value of a binary numeral

- To find the value of a binary numeral, follow the example below. Suppose the numeral is **101101**. List the powers of 2 in row 1, their values in row 2, and the binary in row 3:

2^5	2^4	2^3	2^2	2^1	2^0
32	16	8	4	2	1
1	0	1	1	0	1

- For each **1** in the bottom row, add the number above it to the value. The value of **101101** is therefore $2^5 + 2^3 + 2^2 + 2^0$ = **32 + 8 + 4 + 1 = 45**.

Binary addition

- Adding binary numbers uses the same principles as adding decimal numbers, using place values and carrying if the sum in one place is greater than the largest possible digit.
- In base 10, if the sum exceeds 9 (10 less 1), you perform a carry.
- In base 2, if the sum exceeds 1 (2 less 1), you perform a carry.
- When you learned base 10 addition in elementary school, you began with an addition table. We do the same thing now with base 2.

Binary Addition Table

The addition table:

+	0	1
0	0	1
1	1	10

Notice that **1 + 1 = 2**, which is **10** in binary. Armed with this table, the rest is easy.

To the right we add **11 + 1** and get **100** because **1 + 1 = 10**, and the **1** is carried to the next bit, as shown. Then **1+1 = 10** is written below.

$$\begin{array}{lccc} & 1^{1} & & 1 \\ + & & & 1 \\ \hline 1 & 0 & & 0 \end{array}$$

Binary Addition Continued

- If you can carry, you can add any two numbers:

```
    1¹1¹ 1
+      1 1
 1 0 1 0
```

- Notice that **`1+1+1= (1+1)+1 = 10 + 1 = 11`**. It is now possible to write the binary numerals in sequence, until you get bored with them:

  `0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111, 10000, 10001, 10010, 10011, 10100, …`

Decimal to binary conversion

- Given a decimal number, ***x***, do the following to get its binary numeral:

1. Divide ***x*** by 2.
2. Write the remainder to the immediate left of the previous remainder. If this is the first remainder, right it down leaving room to its left for more bits.
3. If the quotient is not 0, replace ***x*** by the quotient and go back to step 1, otherwise go to step 4.
4. The binary numeral is the sequence of remainders in left-to-right order.

Example

x	Arithmetic	Quotient	Remainder
53	53/2=26 r 1	26	1
26	26/2=13 r 0	13	01
13	13/2 = 6 r 1	6	101
6	6/2 = 3 r 0	3	0101
3	3/2 = 1 r 1	1	10101
1	1/2 = 0 r 1	0	110101

So the binary for 53 is 110101.

Permissions as numbers

- Recall that the permission string has 3 ***bits*** for each of user, group, and others:

```
_ _ _     _ _ _     _ _ _
r w x     r w x     r w x
```

user **group** **others**

- Each of these bits is a binary bit that is either **`1`**, if the permission is on, or **`0`**, if it is off. Thus, the string **`r-x`** is equivalent to the bit pattern **`101`**. Since **`101`** is the number **`5`** in decimal, three decimal numbers can represent a permission string. (They are really octal numbers, but we ignore that for now.

Eight possible combinations

Permission String	Binary Equivalent	Decimal Equivalent
`---`	`000`	`0`
`--x`	`001`	`1`
`-w-`	`010`	`2`
`-wx`	`011`	`3`
`r--`	`100`	`4`
`r-x`	`101`	`5`
`rw-`	`110`	`6`
`rwx`	`111`	`7`

Permission strings as 3-digit numbers

- Since each set of three bits can be represented by a decimal number from **`0`** to **`7`**, the permissions on a file can be represented by a set of 3 numbers from **`0`** to **`7`**.
- For example, the number **`751`** represents the binary numeral **`111 101 001`**, which is converted to the permission string **`rwx r-x --x`**, meaning that the owner has full privilege, the group, read and execute, and everyone else, just execute.

Common Permission Values

- There are certain permissions that occur very frequently because of the concentric nature of security in UNIX.

755	- owner has full privilege; rest cannot write
700	- owner has full privilege; rest have none (secret stuff)
644	- owner has full privilege; rest cannot write, but file is not executable nor a directory
600	- owner has full privilege; rest have none (secret stuff), but file is not executable nor a directory
666	- everyone can read and write (certain device files look this way.)

Absolute permissions for `chmod`

- The `chmod` command can accept numeric flags instead of the more complicated flags described earlier.

  ```
  chmod 755 myfile
  ```

  for example sets the permission string of `myfile` to `755`, which is `rwxr-xr-x`, and

  ```
  chmod 600 privatestuff
  ```

  sets the permission of `privatestuff` to `rw-------`.

The **umask** command (may be skipped)

- When you create a file, it is given default permissions. For example, if you enter the commands

```
$ touch newfile
$ ls -l newfile
-rw-r--r--
```

  you see that the new file was created with a default of 644 in numeric form. The **umask** command controls these defaults.

- The **umask** command is given what looks like a permission string written in numeric form. ***It is a mask though, not a set of permissions.***

Masks and the `umask` (may be skipped)

- The **`umask`** is a ***mask***. This means that each **`1`** bit acts like a ***blocker*** and each **`0`** bit acts like a ***hole***. Masks hide stuff. A 1 means "hide" and a 0 means "don't hide."
- The **`umask 037`** is the binary numeral **`000 011 111`**, since **`3`** is **`011`** and **`7`** is **`111`**. This means that when UNIX creates a file with this mask, wherever there is a 1, there will be a 0 in the permission string, and wherever there is a 0, there may be a 1. Maybe not. If the file is not executable, then UNIX is smart enough not to try to turn that bit on.
- Typing **`umask`** by itself displays the current **`umask`**.

Example (may be skipped)

The following example show how the **umask** behaves.

```
$ umask
0037           # ignore the leading 0 for now
$ touch newfile
$ ls -l newfile
-rw-r-----  1 sweiss cs49366 0 Aug 10 21:32 newfile
$
```

The **umask 037** is **000 011 111**. The most permission possible is **111 100 000** (the numbers reversed.) But touch does not create executable files, so turn off the execute bits and you get **110 100 000**, which is **rw- r-- ---**.

Back to shell scripts

- Now that you've learned a few commands and can edit files, you can start to write shell scripts.
- You need a few more tools to make this possible. For starters, you need to know how a shell script can access the words you type on its command line.
- For example, suppose I want to write a script named **`swap`** that when called like this:

```
swap  word1 word2
```

would output this:

```
word2 word1
```

Command line arguments

- Bash can access the words on the command line using the variables **$1**, **$2**, **$3**, and so on. The first word is stored in **$1**, the second, in **$2**, etc.
- The number of words on the line, ***excluding the command itself***, is stored in a variable named **$#**.
- Therefore, our **swap** script would be as simple as

```
#!/bin/bash
echo $2 $1
```

- This displays the second word, then the first word

Adding error-checking: `test`

- Scripts should always check that they have the correct number of arguments and that they are all valid. **`swap`** just has to check that there are exactly 2 words, so it needs to ***test*** whether **`$#`** is equal to **`2`**.
- The **`test`** command evaluates an expression. For example

```
test 1 -ne 2
```

is true because 1 is ***n***ot ***e***qual to 2.

```
test 2 -eq 2
```

is true because 2 ***eq***uals 2. The other operators are **`-le`**, **`-lt`**, **`-gt`**, and **`-ge`**. Can you guess what these mean?

Other forms of **`test`**

- If you read the man page for test, you will see that there are other ways to use it. For example

```
[ 1 -ne 2 ]
```

  is equivalent to

```
test 1 -ne 2
```

- You can put square brackets around an expression but there must be spaces on either side of them:

```
[1 -ne 2]
```

  would be an error.

Using the **`if`** command

- Every shell has a command named **`if`**. In **`bash`**, you have to follow a very specific set of rules to use it. In its simplest form it looks like this:

```
    if   test-command
    then
              any sequence of commands
    fi
```

- where the words **`if`**, **`then`**, and **`fi`** are on separate lines.

Example **if** command

```
if  test $# -ne 2
 then
       echo usage: swap word1 word2
       exit
 fi
```

- This piece of shell script will print a usage message if the number of words on the command line is not equal to 2.
- It will also quit immediately after printing the message, no matter what commands follow the word **fi**.

Putting it all together

- We can put the testing bit of stuff ahead of our **echo** command to do our input-checking, and we now have a safe script:

```
#!/bin/bash
 if  test $# -ne 2
 then
        echo usage: swap word1 word2
        exit
 fi
 echo $2 $1
```

Another Type of Test

- The test command has many different types of tests. Many are called file tests and they can be used to test whether a file exists, or is of a given type, or size, or has some other property. For example:

```
#!/bin/bash
 if test -e $1
 then
       echo “$1 exists”
 fi
```

Negating Tests

- There is no test that is true if a file does not exist. If you want to print an error message if the user did not supply a filename, you need to negate the test. The exclamation mark negates expressions:

  ```
  if test ! -e myfile
  ```

  is true if **`myfile`** does not exist, and is false if it does. If **`$1`** is a command line argument then

  ```
  if test ! -e $1
  ```

  is true if it is not the name of a file that exists.

More About if

- The **if** statement can also have an "**else**" part like this:

```
#!/bin/bash
if test -e $1
then
        echo "$1 exists"
else
        echo "$1 does not exist"
fi
```

- The statement after "**else**" is executed if the test evaluates to false.

More About if

- The **if** statement can also have multiple **elif** parts:

```
#!/bin/bash
if test $# -eq 0
then
    echo “No arguments”
elif test $# -eq 1
    echo “$1”
elif test $# -eq 2
    echo “$1  $2”
else
    echo “More than 2 arguments”
fi
```

Adding comments to the script

- A line that starts with # and no ! after it is called a ***comment***. The shell ignores everything to the right of the #.
- In fact, the # can be written to the right of a command and the shell will ignore the rest of the line after the #:

```
# Written by Stewart Weiss, 09/24/2009
# This script swaps words.
echo $2 $1   # swap first and second words
```

Shell comments

- Always add your authorship and other information in a comment :

```
#!/bin/bash
# Written by Stewart Weiss, 09/24/2009
if  test $# -ne 2
then
    echo usage: swap word1 word2
else
    echo $2  $1
fi
```

Things to try

- Try creating a few simple scripts of your own. It will give you practice using **`gedit`** if you are at a UNIX console, or **`vi`** or **`nano`** if you are not.
- Read about the **`test`** command and learn its tricky syntax.
- Play around with **`>`** to store the output of various commands.
- Get comfortable with binary numbers.