

# 04 - More Files, Chaining Commands, and your First(?) Git Repository

CS 2043: Unix Tools and Scripting, Spring 2016 [1]

---

Stephen McDowell

February 3rd, 2016

Cornell University

# Table of contents

1. Recap on Permissions
2. File Compression
3. Assorted Commands
4. Chaining Commands
5. More Git: Forking a Repository

## Some Logistics

- Last day to add is today.

## Some Logistics

- Last day to add is today.
- (Poll) The demo last time.

## Recap on Permissions

---

## The Octal Version of **chmod**

Last time I linked you to this[2] website for a good explanation. For the formula hungry, you can represent **r**, **w**, and **x** as binary variables (where 0 is off, and 1 is on). Then the formula for the modes is

$$r \cdot 2^2 + w \cdot 2^1 + x \cdot 2^0$$

### Examples

- `chmod 755: rwxr-xr-x`
- `chmod 777: rwxrwxrwx`
- `chmod 600: rw- - - - -`

If that makes less sense to you, feel free to ignore it.

# Super Confused...

## Superuser Do

`sudo <command>`

- Execute `<command>` as the super user.
- The regular user (e.g. **student**) is executing the **sudo** command, *not* the **root**.
- You enter *your* user password.
- You can only execute **sudo** if you are an "administrator"\*.

So on the course VMs the **student** user originally had the password **student**, so that is what you would type if you were executing **sudo**. On your personal Mac (or native Linux install), you would be typing whatever your password is to login to the computer.

\* Note that where you look to see who can execute **sudo** varies greatly between distributions.

## Super Confused...

If you know the **root** password, then you can become **root** using **su** directly.

### Switch User

```
su <user_name>
```

- Switches to user **user\_name**.
- The password you enter is the password for **user\_name**.
- If no username is specified, **root** is implied.



## Super Confused...

If you know the **root** password, then you can become **root** using **su** directly.

### Switch User

**su** <user\_name>

- Switches to user **user\_name**.
- The password you enter is the password for **user\_name**.
- If no username is specified, **root** is implied.

- The commands **sudo su root** and **sudo su** are equivalent:

## Super Confused...

If you know the **root** password, then you can become **root** using **su** directly.

### Switch User

```
su <user_name>
```

- Switches to user **user\_name**.
- The password you enter is the password for **user\_name**.
- If no username is specified, **root** is implied.

- The commands **sudo su root** and **sudo su** are equivalent:
  - Since you typed **sudo** first, that is why you type the user password.

## Super Confused...

If you know the **root** password, then you can become **root** using **su** directly.

### Switch User

```
su <user_name>
```

- Switches to user **user\_name**.
  - The password you enter is the password for **user\_name**.
  - If no username is specified, **root** is implied.
- 
- The commands **sudo su root** and **sudo su** are equivalent:
    - Since you typed **sudo** first, that is why you type the user password.
  - If you just execute **su** directly, then you have to type the **root** password.

# Default Permissions

When you create files during a particular session, the mode you are running in determines what the permissions will be.

## User mask

`umask <mode>`

- Remove **mode** from the file's permissions.
- Similar syntax to **chmod**:
  - `umask 077`: full access to the user, no access to anybody else.
  - `umask g+w`: enables group write permissions.
- `umask -S`: display the current mask.

# Default Permissions

When you create files during a particular session, the mode you are running in determines what the permissions will be.

## User mask

`umask <mode>`

- Remove **mode** from the file's permissions.
- Similar syntax to **chmod**:
  - `umask 077`: full access to the user, no access to anybody else.
  - `umask g+w`: enables group write permissions.
- `umask -S`: display the current mask.

- Changing the **umask** only applies for the remainder of the session (e.g. until you close the terminal window you were writing this in).

# Default Permissions

When you create files during a particular session, the mode you are running in determines what the permissions will be.

## User mask

`umask <mode>`

- Remove **mode** from the file's permissions.
- Similar syntax to **chmod**:
  - `umask 077`: full access to the user, no access to anybody else.
  - `umask g+w`: enables group write permissions.
- `umask -S`: display the current mask.

- Changing the **umask** only applies for the remainder of the session (e.g. until you close the terminal window you were writing this in).
- If this has meaning, it is just a bit mask with **0o777**.

# File Compression

---

# Making Archives: Zip

## Zip

```
zip <name_of_archive> <files_to_include>
```

- Note I said *files*.
  - E.g. `zip files.zip a.txt b.txt c.txt`
  - These will extract to `a.txt`, `b.txt`, and `c.txt` in the current directory.
- To do folders, you need recursion.
  - `zip -r folder.zip my_files/`
  - This will extract to a folder named `my_files`, with whatever was inside of it in tact.

## Unzip

```
unzip <archive_name>
```

**Note:** The original files DO stay in tact.



# Making Archives: Gzip

## Gzip

```
gzip <files_to_compress>
```

- Less time to compress, larger file: **--fast**
- More time to compress, smaller file: **--best**
- Read the **man** page, lots of options.

## Gunzip

```
gunzip <archive_name>
```

## Notes:

- By default, *replaces* the original files!
  - You can use **--keep** to bypass this.
- Does not bundle the files.
- Usually has better compression than **zip**.

# Making Archives: Tar

## Tape Archive

```
tar -cf <tar_archive_name> <files_to_compress>
```

- Create a tar archive.

```
tar -xf <tar_archive_name>
```

- Extract all files from archive.

## Notes:

- **tar** is just a bundling suite, creating a single file.
- By default, it does *not* compress.
- Original files DO stay in tact.
- Unlike **zip**, you do not need the **-r** flag for folders :)

# Making Archives: Tarballs

## Making tarballs

```
tar -c(z/j)f <archive_name> <source_files>
```

```
tar -x(z/j)f <archive_name>
```

- (z/j) here means *either* z or j, **not** both.
- The -z flag specifies **gzip** as the compression method.
- YOU have to specify the file extension.
  - Extension convention: **.tar.gz**
  - Example: **tar -cjf files.tar.gz files/**
- The -j flag specifies **bzip2** as the compression method.
  - Extension convention: **.tar.bz2**
  - Example: **tar -cjf files.tar.bz2 files/**

### Note:

- Extraction can *usually* happen automatically:
  - **tar -xf files.tar.gz** will usually work (no -z)

## Assorted Commands

---

Before we can Chain...

...we need some more interesting tools to chain together!

# Counting

## Word Count

```
wc [options] <file>
```

- l: count the number of lines.
- w: count the number of words.
- m: count the number of characters.
- c: count the number of bytes.

# Counting

## Word Count

```
wc [options] <file>
```

- l: count the number of lines.
- w: count the number of words.
- m: count the number of characters.
- c: count the number of bytes.

Great for things like:

# Counting

## Word Count

```
wc [options] <file>
```

- l: count the number of lines.
- w: count the number of words.
- m: count the number of characters.
- c: count the number of bytes.

Great for things like:

- revelling in the number of lines you have programmed.



# Counting

## Word Count

```
wc [options] <file>
```

- l: count the number of lines.
- w: count the number of words.
- m: count the number of characters.
- c: count the number of bytes.

Great for things like:

- revelling in the number of lines you have programmed.
- analyzing the verbosity of your personal statement.

## Word Count

```
wc [options] <file>
```

- l: count the number of lines.
- w: count the number of words.
- m: count the number of characters.
- c: count the number of bytes.

Great for things like:

- revelling in the number of lines you have programmed.
- analyzing the verbosity of your personal statement.
- showing people how cool you are.

# Sorting

## Sort

`sort [options] <file>`

- Default: sort by the **ASCII** code (roughly alphabetical) for the whole line.
- Use `-r` to reverse the order.
- Use `-n` to sort by numerical order.
- Use `-u` to remove duplicates.

```
>>> cat peeps.txt
Manson, Charles
Bundy, Ted
Bundy, Jed
Nevs, Sven
Nevs, Sven
```

```
>>> sort -r peeps.txt
Nevs, Sven
Nevs, Sven
Manson, Charles
Bundy, Ted
Bundy, Jed
```

```
>>> sort -ru peeps.txt
Nevs, Sven
Manson, Charles
Bundy, Ted
Bundy, Jed
# only 1 Nevs, Sven
```

# Advanced Sorting

The **sort** command is quite powerful, for example you can do

- **sort -n -k 2 -t "," <filename>**
  - sorts the file numerically by using the second column, separating by a comma as the separator instead of a space
- read the **man** page!

```
>>> cat numbers.txt
02,there
04,how
01,hi
06,you
03,bob
05,are
```

```
>>> sort -n -k 2 -t "," numbers.txt
01,hi
02,there
03,bob
04,how
05,are
06,you
```

## Unique

`uniq [options] <file>`

- No flags: discards all but one of successive identical lines.
- Use `-c` to prints the number of successive identical lines next to each line.

# Search and Replace

## Translate

```
tr [options] <set1> [set2]
```

- Translate or delete characters.
- Sets are strings of characters.
- By default, searches for strings matching **set1** and replaces them with **set2**.
- You can use POSIX and custom-defined sets (we'll get there soon!).

# Search and Replace

## Translate

```
tr [options] <set1> [set2]
```

- Translate or delete characters.
- Sets are strings of characters.
- By default, searches for strings matching **set1** and replaces them with **set2**.
- You can use POSIX and custom-defined sets (we'll get there soon!).

- The **tr** command only works with streams.

# Search and Replace

## Translate

```
tr [options] <set1> [set2]
```

- Translate or delete characters.
- Sets are strings of characters.
- By default, searches for strings matching **set1** and replaces them with **set2**.
- You can use POSIX and custom-defined sets (we'll get there soon!).

- The **tr** command only works with streams.
- Examples to come after we learn about chaining commands in the next section.



# Chaining Commands

---

## Your Environment and Variables

- There are various environment variables defined in your environment. They are almost always all capital letters.

## Your Environment and Variables

- There are various environment variables defined in your environment. They are almost always all capital letters.
- You obtain their value by dereferencing them with a \$.

# Your Environment and Variables

- There are various environment variables defined in your environment. They are almost always all capital letters.
- You obtain their value by dereferencing them with a \$.

```
>>> echo $PWD      # present working directory
>>> echo $OLDPWD    # print previous working directory
>>> printenv        # print all environment variables
```

# Your Environment and Variables

- There are various environment variables defined in your environment. They are almost always all capital letters.
- You obtain their value by dereferencing them with a \$.

```
>>> echo $PWD      # present working directory
>>> echo $OLDPWD    # print previous working directory
>>> printenv        # print all environment variables
```

- When you execute commands, they have something called an "exit code".

# Your Environment and Variables

- There are various environment variables defined in your environment. They are almost always all capital letters.
- You obtain their value by dereferencing them with a \$.

```
>>> echo $PWD      # present working directory
>>> echo $OLDPWD    # print previous working directory
>>> printenv        # print all environment variables
```

- When you execute commands, they have something called an "exit code".
- The exit code of the last command executed is stored in the **\$?** environment variable.

# What is Defined?

- The environment:

# What is Defined?

- The environment:
  - **env**: displays all environment variables.



# What is Defined?

- The environment:
  - `env`: displays all environment variables.
  - `unsetenv <name>`: remove an environment variable.

# What is Defined?

- The environment:
  - `env`: displays all environment variables.
  - `unsetenv <name>`: remove an environment variable.
- The local variables:

# What is Defined?

- The environment:
  - `env`: displays all environment variables.
  - `unsetenv <name>`: remove an environment variable.
- The local variables:
  - `set`: displays all shell / local variables.

# What is Defined?

- The environment:
  - **env**: displays all environment variables.
  - **unsetenv <name>**: remove an environment variable.
- The local variables:
  - **set**: displays all shell / local variables.
  - **unset <name>**: remove a shell variable.

# What is Defined?

- The environment:
  - `env`: displays all environment variables.
  - `unsetenv <name>`: remove an environment variable.
- The local variables:
  - `set`: displays all shell / local variables.
  - `unset <name>`: remove a shell variable.
- We'll cover these a little more when we talk about customizing your terminal shell.

# Exit Codes

There are various exit codes, here are a few examples:

```
>>> super_awesome_command
bash: super_awesome_command: command not found...
>>> echo $?
127
>>> echo "What is the exit code we want?"
>>> echo $?
0
```

- The success code we want is actually **0**. Refer to [3] for some more examples.

# Exit Codes

There are various exit codes, here are a few examples:

```
>>> super_awesome_command
bash: super_awesome_command: command not found...
>>> echo $?
127
>>> echo "What is the exit code we want?"
>>> echo $?
0
```

- The success code we want is actually **0**. Refer to [3] for some more examples.
- Remember that **cat /dev/urandom** trickery? You will have to **ctrl+c** to kill it, what would the exit code be?

## Executing Multiple Commands in a Row

With exit codes, we can define some simple rules to chain commands together:

- Always execute:



## Executing Multiple Commands in a Row

With exit codes, we can define some simple rules to chain commands together:

- Always execute:

```
>>> cmd1; cmd2    # exec cmd1 first, then cmd2
```

## Executing Multiple Commands in a Row

With exit codes, we can define some simple rules to chain commands together:

- Always execute:

```
>>> cmd1; cmd2    # exec cmd1 first, then cmd2
```

- Execute conditioned upon exit code:

## Executing Multiple Commands in a Row

With exit codes, we can define some simple rules to chain commands together:

- Always execute:

```
>>> cmd1; cmd2    # exec cmd1 first, then cmd2
```

- Execute conditioned upon exit code:

```
>>> cmd1 && cmd2 # exec cmd2 only if cmd1 returned 0
```

```
>>> cmd1 || cmd2 # exec cmd2 only if cmd1 returned NOT 0
```

## Executing Multiple Commands in a Row

With exit codes, we can define some simple rules to chain commands together:

- Always execute:

```
>>> cmd1; cmd2    # exec cmd1 first, then cmd2
```

- Execute conditioned upon exit code:

```
>>> cmd1 && cmd2 # exec cmd2 only if cmd1 returned 0  
>>> cmd1 || cmd2 # exec cmd2 only if cmd1 returned NOT 0
```

- Kind of backwards, in terms of what means continue for *and*, but that was likely easier to implement since there is only one **0** and many not **0**'s.

# Piping Commands

Bash scripting is all about combining simple commands together to do more powerful things. This is accomplished using the "pipe" character.

## Piping

```
<command1> | <command2>
```

- Passes the output from **command1** to be the input of **command2**.
- Works for *heaps* of programs that take input and provide output to the terminal.

## Some Piping Examples

### Piping along...

```
>>> ls -al /bin | less
```

- Allows you to scroll through the long list of programs in `/bin`

```
>>> history | tail -20 | head -10
```

- Displays the 10<sup>th</sup> - 19<sup>th</sup> previous commands from the previous session.

```
>>> echo * | tr ' ' '\n'
```

- Replaces all spaces characters with new lines.
- Execute just `echo *` to see the difference.

## Redirection

To redirect input / output streams, you can use one of `>`, `>>`, `<`, or `<<`.

# Redirection

To redirect input / output streams, you can use one of `>`, `>>`, `<`, or `<<`.

- To redirect standard output, use the `>` operator.



# Redirection

To redirect input / output streams, you can use one of `>`, `>>`, `<`, or `<<`.

- To redirect standard output, use the `>` operator.
  - `command > file`

# Redirection

To redirect input / output streams, you can use one of `>`, `>>`, `<`, or `<<`.

- To redirect standard output, use the `>` operator.
  - `command > file`
- To redirect standard input, use the `<` operator.

# Redirection

To redirect input / output streams, you can use one of `>`, `>>`, `<`, or `<<`.

- To redirect standard output, use the `>` operator.
  - `command > file`
- To redirect standard input, use the `<` operator.
  - `command < file`

# Redirection

To redirect input / output streams, you can use one of `>`, `>>`, `<`, or `<<`.

- To redirect standard output, use the `>` operator.
  - `command > file`
- To redirect standard input, use the `<` operator.
  - `command < file`
- To redirect standard error, use the `>` operator and specify the stream number `2`.

# Redirection

To redirect input / output streams, you can use one of `>`, `>>`, `<`, or `<<`.

- To redirect standard output, use the `>` operator.
  - `command > file`
- To redirect standard input, use the `<` operator.
  - `command < file`
- To redirect standard error, use the `>` operator and specify the stream number 2.
  - `command 2> file`

# Redirection

To redirect input / output streams, you can use one of `>`, `>>`, `<`, or `<<`.

- To redirect standard output, use the `>` operator.
  - `command > file`
- To redirect standard input, use the `<` operator.
  - `command < file`
- To redirect standard error, use the `>` operator and specify the stream number 2.
  - `command 2> file`
- Combine streams together by using `2>&1` syntax.

# Redirection

To redirect input / output streams, you can use one of `>`, `>>`, `<`, or `<<`.

- To redirect standard output, use the `>` operator.
  - `command > file`
- To redirect standard input, use the `<` operator.
  - `command < file`
- To redirect standard error, use the `>` operator and specify the stream number 2.
  - `command 2> file`
- Combine streams together by using `2>&1` syntax.
  - This says: send standard error to where standard output is going.

# Redirection

To redirect input / output streams, you can use one of `>`, `>>`, `<`, or `<<`.

- To redirect standard output, use the `>` operator.
  - `command > file`
- To redirect standard input, use the `<` operator.
  - `command < file`
- To redirect standard error, use the `>` operator and specify the stream number 2.
  - `command 2> file`
- Combine streams together by using `2>&1` syntax.
  - This says: send standard error to where standard output is going.
  - Useful for debugging / catching error messages...



# Redirection

To redirect input / output streams, you can use one of `>`, `>>`, `<`, or `<<`.

- To redirect standard output, use the `>` operator.
  - `command > file`
- To redirect standard input, use the `<` operator.
  - `command < file`
- To redirect standard error, use the `>` operator and specify the stream number 2.
  - `command 2> file`
- Combine streams together by using `2>&1` syntax.
  - This says: send standard error to where standard output is going.
  - Useful for debugging / catching error messages...
  - ...or ignoring them (you will often see that sent to `/dev/null`).

## Redirection Example

- Bash processes I/O redirection from left to right, allowing us to do fun things like this:

## Redirection Example

- Bash processes I/O redirection from left to right, allowing us to do fun things like this:

Magic

```
tr -cd '0-9' < test1.txt > test2.txt
```

## Redirection Example

- Bash processes I/O redirection from left to right, allowing us to do fun things like this:

### Magic

```
tr -cd '0-9' < test1.txt > test2.txt
```

- Deletes everything but the numbers from **test1.txt**, then store them in **test2.txt**.

## Redirection Example

- Bash processes I/O redirection from left to right, allowing us to do fun things like this:

### Magic

```
tr -cd '0-9' < test1.txt > test2.txt
```

- Deletes everything but the numbers from `test1.txt`, then store them in `test2.txt`.
- **CAUTION:** do not *ever* use the same file as output that was input.

## Redirection Example

- Bash processes I/O redirection from left to right, allowing us to do fun things like this:

### Magic

```
tr -cd '0-9' < test1.txt > test2.txt
```

- Deletes everything but the numbers from `test1.txt`, then store them in `test2.txt`.
- CAUTION: do not *ever* use the same file as output that was input.
  - Example: `tr -cd '0-9' < original.txt > original.txt`

## Redirection Example

- Bash processes I/O redirection from left to right, allowing us to do fun things like this:

### Magic

```
tr -cd '0-9' < test1.txt > test2.txt
```

- Deletes everything but the numbers from `test1.txt`, then store them in `test2.txt`.
- CAUTION: do not *ever* use the same file as output that was input.
  - Example: `tr -cd '0-9' < original.txt > original.txt`
  - You will lose all your data, you cannot read and write this way.

# Redirection Example

- Bash processes I/O redirection from left to right, allowing us to do fun things like this:

## Magic

```
tr -cd '0-9' < test1.txt > test2.txt
```

- Deletes everything but the numbers from `test1.txt`, then store them in `test2.txt`.
- CAUTION: do not *ever* use the same file as output that was input.
  - Example: `tr -cd '0-9' < original.txt > original.txt`
  - You will *lose* all your data, you cannot read and write this way.

- Piping and Redirection are quite sophisticated, please refer to the Wikipedia page in [4].



## More Git: Forking a Repository

---

In class demo...

<https://github.com/cs2043-sp16/lecture-demos/tree/master/lec04>

## References I

- [1] B. Abrahao, H. Abu-Libdeh, N. Savva, D. Slater, and others over the years.  
Previous cornell cs 2043 course slides.
- [2] C. Hope.  
Linux and unix chmod command help and examples.  
<http://www.computerhope.com/unix/uchmod.htm>,  
2016.
- [3] T. L. D. Project.  
Exit codes with special meanings.  
<http://tldp.org/LDP/abs/html/exitcodes.html>.

- [4] Wikipedia.  
Redirection (computing).  
[https://en.wikipedia.org/wiki/Redirection\\_  
%28computing%29](https://en.wikipedia.org/wiki/Redirection_%28computing%29).