

# 05 – Archiving, Chaining, Piping & Redirection

CS 2043: Unix Tools and Scripting, Spring 2019 [2]

---

Matthew Milano

February 1, 2019

Cornell University

# Table of Contents

1. File Compression
2. Assorted Commands
3. Chaining Commands
4. Piping & Redirection

# File Compression

---

# Making Archives: Zip

## Package and Compress (Archive) Files

`zip <name_of_archive> <files_to_include>`

- E.g. `zip files.zip a.txt b.txt c.txt`
- Extracts to `a.txt`, `b.txt`, and `c.txt` in *current directory*.
- To do folders, you need recursion.
  - `zip -r folder.zip my_files/`
  - Extracts to folder named `my_files` in *current directory*.
  - Good practice to ALWAYS zip a folder and distribute with the name it will extract as.
    - `zip -r folder_name.zip folder_name/`
  - Drives me *crazy* when I get a `.zip` that extracts files in the same directory... very difficult to keep track of.

## List, Test and Extract Compressed Files in a **zip** Archive

`unzip <archive_name>`

- Use `-l` to list what would extract before doing it.
- **Note:** The original files DO stay intact.

# Making Archives: Gzip

## GNU zip

```
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.
- By default, *replaces* the original files!
  - You can use - -**keep** to bypass this.

## GNU unzip

```
gunzip <archive_name>
```

- Use -**l** to list what would extract before doing it.

### • Notes:

- Does not bundle the files.
- Reiterate: *replaces original* by default.
- Usually has better compression than **zip**.

## Additional Archive Formats

- This is a non-exhaustive list. There are **many** out there.
- Similar interface to **gzip**:
  - **bzip2**: “Burrows-Wheeler block sorting compression algorithm”
  - **xz**: “x”-zip, uses LZMA compression scheme (good)
- Honorable mentions:
  - **file.rar**: a “**RAR**” archive; used for distributing large files
    - **file.rar.001**, **file.rar.002**, etc: multiple archives needed to reconstruct whole.
    - You extract the first one, it looks for the others in same directory.
  - **file.7z**: “**7**”-zip, successor to RAR, uses LZMA
    - If you are choosing between **.rar** and **.7z**...choose **.7z**.
  - Install **unrar** to deal with these on Unix.
- **Moral**:
  - Working with **tar** and/or only Unix? **Use xz**.
  - Have to support Windows fools? **Use 7zip**.

# Making Archives: Tar

- Bundling files together to compress is easy!

## Tape archive

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

- Create a tar archive.

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

- Extract all files from archive.

- 
- **tar** is a stream tool. By default, it is expecting stream input.
  - Don't forget the **-f** if you are working with files!

## • 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[zjJ]f <archive_name> <source_files>
```

```
tar -x[zjJ]f <archive_name>
```

- [zjJ] here means *either z, j, or J* — only one.
- YOU have to specify the file extension.
- Use **gzip** compression method: -z (or --gzip)
  - Extension convention: **.tar.gz**
  - Example: **tar -czf files.tar.gz files/**
- Use **bzip2** compression method: -j (or --bzip2)
  - Extension convention: **.tar.bz2**
  - Example: **tar -cjf files.tar.bz2 files/**
- Use **xz** compression method: -J (or --xz)
  - Extension convention: **.tar.xz**
  - Example: **tar -cJf files.tar.xz files/**



## Pro Tip: Minimize your Keystrokes

- **Extraction** can *usually* happen automatically:
  - `tar -xf files.tar.gz` will usually work (no `-z`)
  - Best results when:
    - You are obeying filename conventions.
    - `tar` made the archive in the first place.
- **Compression**: no, you have to tell it what to do...
- It's the flag equivalent of the **tab** key.
  - Ok, maybe not...but just remember it!
  - This serves as a not-so-subtle reminder to obsessively hit your **tab** key ;)

# Assorted Commands

---

Before we can Chain...

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

# Counting

- Ever wanted to show off how cool you are?

## Word Count

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

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

- Great for things like:
  - Reveling in the number of lines you have programmed.
  - Analyzing the verbosity of your personal statement.
  - Showing people how cool you are.
  - Completing homework assignments?

# Sorting

## Sort Lines of Text

```
sort [options] <file>
```

- Default: sort by the **ASCII** code (*roughly* alphabetical, see [1]) for the whole line.
- Use **-r** to reverse the order.
- Use **-n** to sort by numerical order.
- Use **-u** to remove duplicates.

- Working with the demo file **peeps.txt**:

```
$ 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: Why?

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

```
$ sort -n -k 3 -t "," <filename>
#      || |||| |----|==> Use comma as delimiter
#      || ++++=====> Choose the third field as the sort key
#      ++=====> Sort numerically
```

- Sorts the file numerically by using the *third* column, separating by a comma as the delimiter instead of whitespace.
- Read the **man** page!
- Learning **sort** command is particularly worth your time:
  - Easy sorting of text  $\Rightarrow$  faster parsing / prototyping.
  - Many commands produce reliably ordered output.
  - Looking for a specific thing? Just sort with that as the **key**!
  - E.g. **grep -Hn**  $\Rightarrow$  use the **:** as your delimiter.
  - We'll learn **grep** soon!

# Advanced Sorting: Example

- The demo file `numbers.txt` contains:

```
$ cat numbers.txt
```

```
02,there,05
```

```
04,how,03
```

```
01,hi,06
```

```
06,you,01
```

```
03,bob,04
```

```
05,are,02
```

```
# Normal numeric sort
```

```
$ sort -n numbers.txt
```

```
01,hi,06
```

```
02,there,05
```

```
03,bob,04
```

```
04,how,03
```

```
05,are,02
```

```
06,you,01
```

```
# On the third column
```

```
$ sort -n -k 3 -t "," numbers.txt
```

```
06,you,01
```

```
05,are,02
```

```
04,how,03
```

```
03,bob,04
```

```
02,there,05
```

```
01,hi,06
```

- Reverse ordering in 3rd column not necessary, just an example.

## Unique — Report or Omit Repeated Lines

`uniq [options] <file>`

- No flags: discards all but one of successive identical lines.
  - Unique occurrences are merged into the *first* occurrence.
- Use `-c` to prints the number of successive identical lines next to each line.
- Use `-d` to only print *repeated* lines.



# Search and Replace

- Translate characters / sets (but not regular expressions) easily!

## Translate or Delete Characters (or Sets)

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

- Translate or delete characters / sets.
    - We will cover POSIX / custom sets soon.
  - By default, searches for strings matching **set1** and replaces them with **set2**.
  - If using **-d** to delete, only **set1** is specified.
  - Can use **-c** to invert (complement) the set.
- 
- 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 for your shell.
- 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
```

- There are also *local* variables you can use / set.
- Primary difference:
  - *Environment* variables are available in your shell, *and* in scripts.
  - *Local* variables are *only* available in your shell.
  - “Shell” here just means “current terminal session.”

# What is Defined?

- The environment:
    - **env**: displays all environment variables.
    - **unsetenv <var\_name>**: remove an environment variable.
    - Create an environment variable\*:
      1. **env ENV\_VAR\_NAME="value"**
      2. **export ENV\_VAR\_NAME="value"**
    - **export** is the most common. Exceptional explanation [here](#).
  - The local variables:
    - **set**: displays all shell / local variables.
    - **unset <var\_name>**: remove a local shell variable.
    - Create a local variable\*:
      1. **set local\_var="value"**
      2. **local\_var="value"**
- \* These only last for the current shell session; we will learn how to make them “permanent” soon.

## Brief Example: Environment Variable Manipulation

```
# MY_ENV_VAR is not set yet, so nothing prints
$ echo "My env var is: $MY_ENV_VAR"
My env var is:
```

```
# Set the environment variable (can also use `export` in bash)
$ env MY_ENV_VAR="Lemming King"
```

```
# Now that we have set it, print it
$ echo "My env var is: $MY_ENV_VAR"
My env var is: Lemming King
```

```
# "Delete" with `unsetenv`. Print again, confirming it's gone
# Emphasis: there is an `env` after `unset`
$ unsetenv MY_ENV_VAR
$ echo "My env var is: $MY_ENV_VAR"
My env var is:
```

## Brief Example: Local Variable Manipulation

```
# my_local_var is not set yet, so nothing prints
$ echo "My local var is: $my_local_var"
My local var is:

# Just declare it (can also use the `set` command)
$ my_local_var="King of the Lemmings"

# Now that we have set it, print it
$ echo "My local var is: $my_local_var"
My local var is: King of the Lemmings

# "Delete" with `unset`. Print again, confirming it's gone
# Emphasis: there is *not* an `env` after `unset`
$ unset my_local_var
$ echo "My local var is: $my_local_var"
My local var is:
```

# Exit Codes

- When you execute commands, they have an “exit code”.
  - This how you “signal” to others in the shell: through exit codes.
- The exit code of the *last command executed* is stored in `$?`
- 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?"
What is the exit code we want?
$ echo $?
0
```

- The success code we want is actually `0`. Refer to [3].
- 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:

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

- Execute conditioned upon exit code of **cmd1**:

```
$ 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 & Redirection

---

# 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>`

- Pass output from **command1** as input to **command2**.
- Works for almost every command.
  - Note: **echo** does not allow you to pipe to it!
- In some senses, the majority of commands you will learn in this course were designed to support this.

# Some Piping Examples

- 1, 2, 3...easy as ABC?

## Piping along...

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

- Scroll through the long list of programs in `/bin`

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

- The 10<sup>th</sup> - 19<sup>th</sup> most recent commands executed.

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

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

- In all of these examples, try executing it first without the |
  - First: execute `history`
  - Next: execute `history | tail -20`
  - Last: execute `history | tail -20 | head -10`

# Redirection

- The redirection operators are: `>`, `>>`, `<`, 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:

### Magic

```
tr -dc '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 -dc '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].
  - Hands-on examples in the [lecture demo](#).

## References

- [1] ASCII Table. *ASCII Character Codes and html, octal, hex, and decimal chart conversion*. 2010. URL: <http://www.asciitable.com/>.
- [2] Stephen McDowell, Bruno Abrahao, Hussam Abu-Libdeh, Nicolas Savva, David Slater, and others over the years. “Previous Cornell CS 2043 Course Slides”.
- [3] The Linux Documentation Project. *Exit Codes with Special Meanings*. 2017. URL: <http://tldp.org/LDP/abs/html/exitcodes.html>.
- [4] Wikipedia. *Redirection (Computing)*. 2017. URL: [https://en.wikipedia.org/wiki/Redirection\\_%28computing%29](https://en.wikipedia.org/wiki/Redirection_%28computing%29).