latex input: mmd-article-header Title: Bash Commands Notes Author: Ethan C. Petuchowski Base Header Level: 1 latex mode: memoir Keywords: Bash, Unix, Linux, Shell, Command Line, Terminal, Syntax CSS: http://fletcherpenney.net/css/document.css xhtml header: copyright: 2015 Ethan C. Petuchowski latex input: mmd-natbib-plain latex input: mmd-article-begin-doc latex footer: mmd-memoir-footer

This document contains summaries of commands that I have found useful-enough to read through the documentation to create these summaries. They should be in alphabetical order.

For all these commands, probably the best *first* resource, is to run `tldr [cmd]`, using the tldr command from github.

ack -- better than grep

Basic:

```
ack pattern
ack --[no]language pattern
```

Specify output format

```
ack '^([^:]+)if:(.*)$' --output='$1 and $2 too!'
```

Options:

- **-w** --- match only whole words
- **-i** --- ignore case
- **-h** --- don't print filename
- **-l** --- only print filenames
- **-c** --- show count of matching lines in *all* files
- **-lc** --- show count of matching lines in *files with matches*
- **-C** --- specify lines of context (default: 2)
- **--pager=LESS** --- pipe it into less (same as `ack ... | less`), except that it can be added (not sure yet if I want this) to the `~/.ackrc`
- **--[no]smart-case** --- Ignore case distinctions in `PATTERN`, only if `PATTERN` contains no upper case. Ignored if **-i** is specified.

apropos --- pertaining too ...

```
$ apropos database
$ man -k database
```

Find all man-pages containing the word "database".

basename

given a filepath, return the part after the last slash

This is the **opposite of** `dirname`

```
$ basename "./dir space/other dir/file.txt"
file.txt

# DON'T do this by accident (viz. ALWAYS quote the filename)
$ basename ./dir space/other dir/file.txt
dir
other
file.txt
```

You can get it to strip off the file suffix, though this doesn't seem to work well with globbing.

```
$ basename args.h .h
args
```

cut --- extract column of text

```
cut -(b|c|f)range [optns] [files]
```

- -c5 --- extract the 5th character of each line
- -b3-5 --- extract the 3rd, 4th, and 5th byte of each line
- -f2,4 -d, --- extract the 2nd and 4th **fields** of each line, where a field delimiter is a comma
- by default, the *delimiter* (`-d`) is a `TAB`
    - to make it a space, try `-d' '`
- --output-delimiter=C --- when you're printing multiple fields, use this delimiter (default is `TAB`)
- -s --- suppress (don't print) lines not containing the delimiter character

diff --- print differences between text files

- -y --- view differences *side-by-side* (mind-blowing)
- -u --- use the +/- format (it's honestly not as nice)

dirname

returns path to the input file, not including the file itself in that path

this is the **opposite of `basename`**

```
$ dirname a/b/myfile
a/b

$ dirname a/myfile
a

$ dirname myfile
.
```

exec

With *no arguments*, change the current shell's *file descriptors*.

```
exec 2> /tmp/mylog  # redirect shell's stderr
exec 3< /file       # open new file descriptor
read var1 var2 <&3  # read from fd 3
exec 5>&2           # save sterr loc in fd 5
exec 2> /otherfile  # new stderr loc
...
exec 2>&5           # copy back saved stderr loc
exec 5>&-           # close fd 5
```

With arguments, `exec` starts a new program in its current process and control *never* returns to the shell.

```
exec java MyApp
```

find

recursive listing of all the files underneath given file

```
$ find .
./.DS_STORE
./file.txt
./a
./a/anotherFile.txt
./a space b
./a space b/file.txt
```

Useful-looking options

- `-(max|min)depth n`
- `-newer FILE` -- only files newer than `FILE`

fmt

Reformat paragraphs by changing line breaks to not exceed a given width. Think of `cmd-opt-q` in my Sublime.

```
$ fmt -w 20 << END
> this line is going to broken up into 20 char chunks
> END
this line is going
to broken up into 20
char chunks
```

grep

Origin: g/re/p means "global regex print" or something like that in the `ed` editor or something like that.

- **-w** -- match only whole words
    - e.g. `grep -w the` will *not* match "thee" or "they"
- **-i** -- ignore case
- **-E** -- normal regexes
    - Otherwise, you have to e.g.
        - Use `\|` instead of `|` to **search for multiple patterns** at once

jot --- print sequential or random data

- *Very* similar to `seq`
- **-r** --- use random data
- **-b** --- just print a given word repeatedly
- Print some random ascii (this could be improved...)
    - `jot -s '' -r -c 100 A`

ln --- create a file that is a link to this file

```
ln [-sif] source target
```

- **i** --- ask before doing anything
- **f** --- don't ask for permission
- **s** --- make a symbolic/soft link instead of a hard link

Hard vs. Soft Links

- **Hard** --- create a new name for a pointer to the `source`-file's `inode` on disk
- **Soft** --- create a new file on disk whose contents hold the `source`-file's *name*
    - if the source file disappears this symbolic link will be broken

locate -- find file on file system

It's just like `find` only *way* faster and less thorough and has less features. For simple things it ought to suffice. It works based on a file-index that I don't know the details of. I do know that it is possible to force a re- build of that index.

lynx --- run the text-based browser

This is *just* too cool.

netstat -anp --- list all stocket usages

- `-a`, `--all` --- show both listening and non-listening sockets
- `-n`, `--numeric` --- Show numerical addresses instead of trying to determine symbolic host, port or user names
- `-p`, `--program` --- Show the PID and name of the program to which each socket belongs.

paste --- make multiple text files into a csv-type-thing

```
$ cat D1
A
B
C

$ cat D2
1
2
3

$ paste D1 D2
A   1
B   2
C   3

$ paste -d, D1 D2
A,1
B,2
C,3

# we can transpose too!!
$ paste -s D1 D2
A   B   C
1   2   3
```

pkg-config -- determine C compiler flags

Examples

```
$ pkg-config fuse --cflags

-D_FILE_OFFSET_BITS=64 -D_DARWIN_USE_64_BIT_INODE -I/usr/local/Cella
```

says to use pkg-config to determine what C compiler flags are necessary to compile a source file that makes use of FUSE.

```
$ pkg-config fuse --libs

-L/usr/local/Cellar/osxfuse/2.7.1/lib -losxfuse -pthread -liconv
```

does the same for the libs to link with.

You can use it in a Makefile like this

```
bbfs : bbfs.o log.o
	gcc -g -o bbfs bbfs.o log.o `pkg-config fuse --libs`

bbfs.o : bbfs.c log.h params.h
	gcc -g -Wall `pkg-config fuse --cflags` -c bbfs.c

log.o : log.c log.h params.h
	gcc -g -Wall `pkg-config fuse --cflags` -c log.c
```

printf --- basically like you'd expect

```
$ printf "hello %s, that's $%.2f please for %d hamburglers\n" Jann 2
# => hello Jann, that's $2.32 please for 3 hamburglers
```

ps -- Process Status

Description:

Prints information about running processes (and threads with option).

You can do things like

- List processes by memory usage
- List them by CPU usage
- List processes by other users
- List them by user

Useful examples

TODO: get some good examples in here.

There are some standard argument-sets that would be good to know. I know Patrick from Workday's included aux, so it was something like

```
ps aux
```

But I can't remember it exactly.

read

Read user input into local variable

Example

```
echo -n "Enter some text > "
read text
echo "You entered: $text"

Enter some text > this is some text
You entered: this is some text
```

You can read *multiple* variables at once, splitting the input string using `$IFS`. Recall that you can set `IFS` for the duration of a single command.

```
echo "thing1:thing2:thing3" | IFS=: read vA vB vC
echo $vB    # => thing2
```

```
$ read a b c
2 3 5   # type to STDIN
$ echo $a $b $c # => 2 3 5
```

rsync -- copy files well

- Copies files between two host, at most one of which may be remote.
- It is efficient because it uses its own algorithm to **transfer only the diffs** between the two sets of files using a "checksum-search" algorithm.
- Supports copying symlinks, owners, groups, permissions, exclusion options.
- Doesn't require root.
- Optimizes transfer latency.
- Can run over
    - SSH -- by using syntax `remote_host:path`
    - raw TCP -- by connecting to an rsync daemon, using syntax `remote_host::path`.

Basic usage

```
$ rsync -t *.c remote:dir/
```

This will transfer all `.c` files from the current directory *into* the `dir` directory on `remote`. We use the `-t` option to update modification times on the remote system to match those on this system. If we don't do that, when we create a file on the remote system as part of the replication, it will have the current time. Then, the next time the synchronization protocol runs, it will compare the modification times, find that they're different, and *recopy* all the `.c` files to `remote`; not what we want.

The slash syntax

- Having a slash or not at the end of the destination has no effect
- If we have a slash on the source, it means copy the *contents* of the directory, but *not* the specified directory object
- Without a slash on the source, it means copy the directory object *and* its contents
- It seems like one way to remember this is that `src/` is short for `src/*` (though I'm not 100% sure that's correct)

Another example

```
$ rsync -avz foo:src/bar /data/tmp  # aka
$ rsync --archive --verbose --compress foo:src/bar /data/tmp
```

Here, "archive" mode, is used to ensure that symbolic links, devices, attributes, permissions, ownerships, etc. are preserved in the transfer; "verbose" is used to increase verbosity; and "compress" is used to compress file data as it is sent to the destination, trading increased CPU for decreased network utilization. Recall that in the first example we used `-t` to update m-times, but here we don't need that because `-a` will transfer "attributes", including m-times.

Copy multiple specific files from remote host

```
$ rsync -av host:'dir1/file1 dir2/file2' /dest
```

Copies `file1` and `file2` into `/dest`. It says additional files *must* be separated by *exactly* one space. This leads to the observation that spaces that exist *within* filenames must be backslash-escaped, even if the path is in single-quotes.

Rsync daemon

If this program is running on the remote host, you are able to create a `/etc/rsyncd.conf` file on the remote host to configure how it handles rsync connections, both over plain TCP, and over SSH. To actually enable that use the double-colon syntax, and [optionally] specify SSH by specifying `--rsh=ssh`.

For example, one may use this to

- configure the maximum number of connections
- specify a log file
- set a timeout
- create "modules", which are short aliases to longer paths on the filesystem.

I guess for my current project, I'd rather specify this stuff on the local machine so that it works for all hosts at once, rather than have to set this thing up on all remote hosts.

Some useful-looking options

- **--update** -- skip files that are newer on the receiver
- **--archive** -- same as `-rlptgoD` (see below)
- **--append** -- so normally the rsync algorithm will create a new copy of the file and then move it into place.
- **--inplace** -- so normally the rsync algorithm will create a new copy of the file and then move it into place. This will prevent that. Note that if there are a few blocks in the middle of the file that were unchanged, doing `inplace` means rsync will have to recopy those blocks when it otherwise wouldn't-have. Note: the file will be in an *inconsistent state* during the transfer. Implies `--partial`.
- **--partial** -- normally rsync deletes partially-transferred files if transfer is interrupted. This prevents that.
- **--progress** -- prints progress of transfer, and summary at the end; implies `--verbose`
-

This is what archive does

- **-r, --recursive** -- recurse into directories
- **-l, --links** -- copy symlinks as symlinks
- **-p, --perms** -- preserve permissions
- **-t, --times** -- preserve times
- **-g, --group** -- preserve groups
- **-o, --owner** -- preserve owner
- **-D** -- preserve devices and special files

seq -- create a sequence of numbers

```
seq [first [incr]] last
```

- Numbers are floating point
- `first` and `incr` both default to 1
- **-s** --- set the separator
  - `$ seq -s \\t 3 => 1\t2\t3`
- **-f** --- use printf style formatters
  - `$ seq -f %.2f -s ' ' 1 .5 3`
  - `=> 1.00 1.50 2.00 2.50 3.00`
- **-t** --- add a terminator to the sequence
- **-w** --- set width by padding with zeros

sort

Concatenate the contents of the given files, and sort that list of lines.

- You can sort by columns
- You can use offsets within columns
- You can use multiple columns, each with its own offset

Check this out

```
$ cat D3
B,2
A,1
E,4
D,5
C,3

# sort by column two, with separator=,
$ sort --key=2 --field-separator=, D3
$ sort -k2 -t, D3
A,1
B,2
C,3
E,4
D,5
```

Useful-looking Options

- `-o FILE, --output=FILE` -- write to file
- `-r, --reverse` -- reverse the output
- `-f, --ignore-case` -- ignore case
- `-u` -- deduplicate lines

ssh

- For logging in and executing commands on a remote machine
- The communication is *both* **secure** and **encrypted** (even over an insecure network)
- It can be used to forward
    - X11 connections
    - arbitrary TCP ports
    - UNIX-domain sockets

SSH Tunnelling

```
ssh -L [<local host>:]<local port>:<remote host>:<remote port> <gat
```

By default, `<local host>` will be `localhost`.

What this does is, start a serversocket listening a local(`host:port`) using the "SSH client". When a connection to that is established, traffic received by that client will be encrypted, and forwarded to the `sshd`[aemon] listening on port 22 of `gateway`. Once traffic is received, the `sshd` will establish a (normal, unencrypted) connection to remote(`host:port`), and forward the data received by the SSH client there. Response traffic originating from remote(`host:port`) will go back to the `sshd`, back through the encrypted tunnel to the SSH client, and back to whomever connected to that client.

tail --- print the end of the file

```
tail [optns] [files]
```

- **-N** --- show last `N` lines
- **+N** --- show all but first `N` lines
- **-f** --- (*watch file*) keep file open, and when new content gets appended, print it too (super useful)
- **-F** --- if a log file is currently getting written to, `tail` will keep the connection to `stdout` open so that all live input into the log file gets written out to the terminal

tr -- translate characters

I think you *have* to pipe or carrat stuff into this thing, there's no place for an input file

To map all uppercase characters to their corresponding lowercase variants:

```
tr A-Z a-z
```

- **-d** --- **delete** all specified characters

  ```
  tr -d ' '   # removes all spaces
  ```

- **-c** --- set **complement**

  ```
  tr -cd ' '  # removes *everything but* the spaces
  ```

- **-s** --- **squeeze** multiple repetitions into a single instance

  ```
  tr -s ' ' '\t'  # convert spaces to tabs
  ```

- **-cs** --- convert and squeeze (used by Doug McIlroy)

```
# convert multiple non-letters into a newline
tr -cs A-Za-z '\n' < /tmp/a.txt
```

uniq

Remove lines that are duplicates of the previous line

`uniq -c --count` --- prefix lines by the number of consecutive occurrences

```
$ sort < a.txt | uniq -c
4 a
2 b
```

wait

Wait for all background jobs to finish

xargs

> This one is my favorite.

It takes a command and a bunch of arguments. It can be used (via `args | xargs -n 1 [cmd]`) to goes through each argument and pass it to a separate invocation of `[cmd]`. By default, `[cmd] = echo`.

- **-n** *num* -- how many of the given arguments to pass to each execution of the given command

```
$ echo a b c d e f| xargs -n 2
a b
c d
e f
```

- **-p** -- prompt the user for confirmation for each potential invocation by revealing to them the derived command that will actually be passed to the shell
- **-t** -- similar to `-p`, except that it just prints the derived command but does not ask for permission to execute it
- **-P** *num* -- execute using `num` *parallel* instances of `[cmd]` at a time
- **-I{}** *Positional argument*

```
echo "foo" | xargs -I{} echo {} "bar"  # => foo bar
```

. (dot)

Read and execute commands contained in a separate file.