Bash Workshop I: The Basics

Ben Zhang

JITech

2024

Table of Contents

Intro

I. File Tree

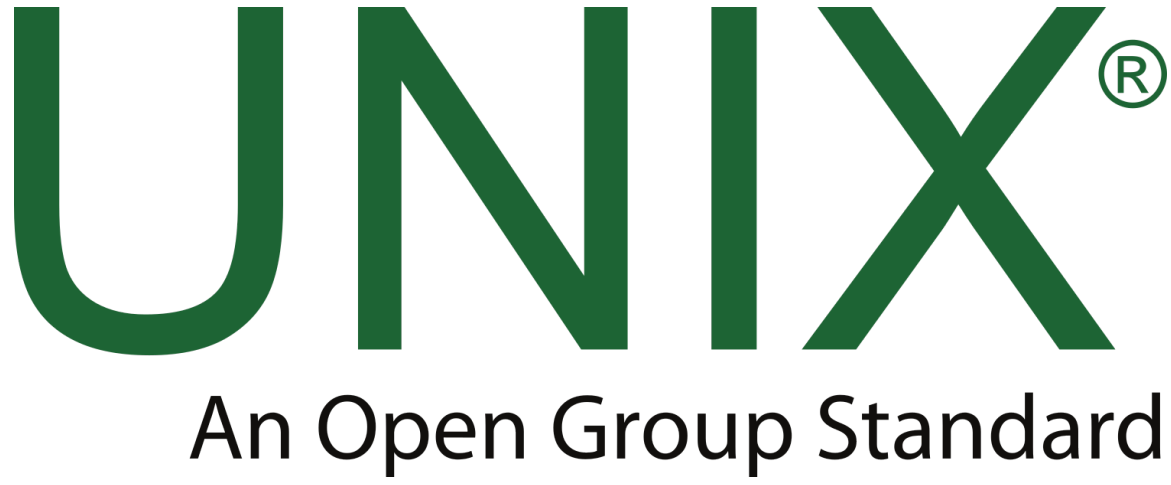
II. CLI

III. Pipes

A brief history of bash

- Born: 1989
- Probably played Pokémon on the Game Boy
- Is an umbrella term for zsh, fish, . . .
- Runs on Unix-like environments

A brief history of Unix



- Born: 1969
- Probably listened to Michael Jackson
- Gave rise to Linux, BSD, and Mac OS
- We call them “Unix-like”

Unix: The Good Part

The Unix philosophy (paraphrased):

- Store data in plain text
- Hierarchical file system
- Everything is a file
- One tool does one thing
- Tools together strong

Quote

The power of a system comes more from the relationships among programs than from the programs themselves.

— Brian Kernighan and Rob Pike [1]

[1] The UNIX Programming Environment. 1984. viii

Unix: The Chaotic Part

"Unix" is mostly created by these three groups of people who routinely disagree with each other:[2]

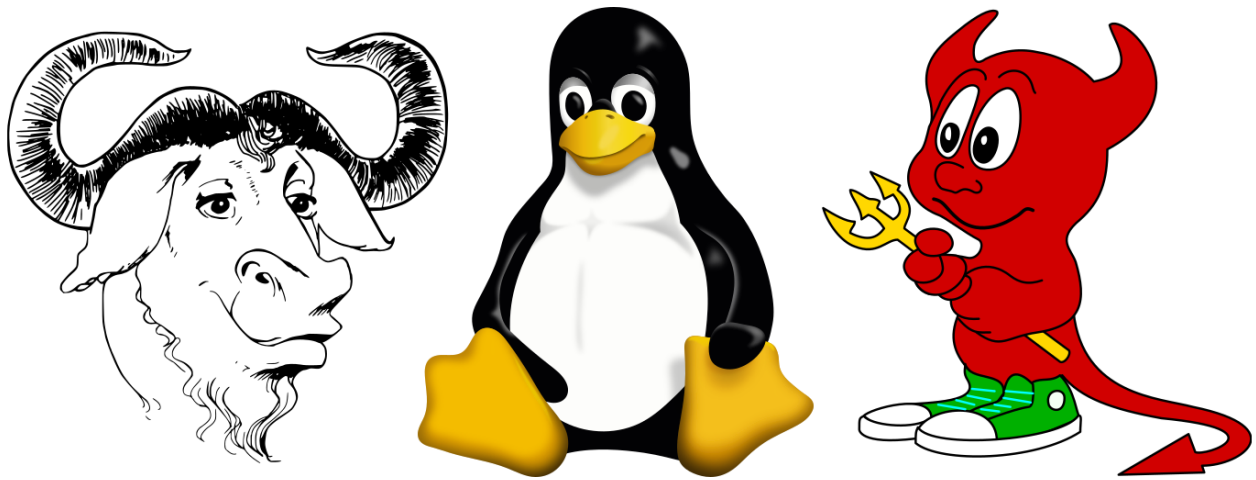


Figure: GNU, Linux, and BSD[3]

Despite this, they all hate Microsoft.
We will be talking about GNU today.

---

[2]Not convinced? Try `man ps`.

[3]Yes, I know there are many BSDs

Before we start

- This is **not** a Linux workshop (although I encourage you to use it)
- This is not a vim workshop either
- You should use a `monospace` font
- Anyone caught using PowerShell will be kicked out of the venue

Conventions in slides

- $ indicates a **bash command**. Do not type the $.
- # indicates a **comment**. Do not type it or anything after it.

For example, when you see:

```
$ echo hello  # printf("hello\n");
```

You are going to type:

```
echo hello
```

Then hit Enter. I encourage you to type commands by hand.

Table of Contents

Intro

I. File Tree

II. CLI

III. Pipes

Files

Each of these is a different **file**:

- `a`
- `.a` (Hidden)
- `a.txt`
- `A.txt`
- `A.TXT`

Note

The dot and suffix are part of the filename.

Avoid spaces and special characters (except `._-`). If you have to, surround filename in quotes: `'Lab Report (3) final FINAL-1.docx'`

`cat`: Printing a file

Open a bash terminal inside `01-files/`, then:

```
$ cat a
$ cat .a
$ cat a.txt
```

Explanation

`cat` is short for "concatenate" (to join together) but it's mostly used to print files.

`cp`, `mv`, `rm`: Relocating a file

Try this inside `01-files/`:

```
$ ls
$ cp a b
$ ls
$ mv a.txt b.txt
$ ls
$ rm b
$ ls
```

Explanation

- `ls` lists files
- **copy** `a` into a file called `b`
- **move** `a.txt` into a file called `b.txt`
- **remove** `b`

`cp`, `mv`: Overwriting and renaming

When the destination does not exist, `cp` and `mv` simply create that file. **Otherwise, it is destroyed and overwritten.**

Try this inside `01-files/`:

```
$ cp a b       # creates b
$ cat a
$ cp a a.txt   # overwrites a.txt
$ cat a.txt
```

Renaming a file in bash works like so:

```
$ mv b newb
```

Directories

Each of these is a **directory** ("dir" for short):

- `01-files/`
- `01-files/c/`
- `01-files/.c/` (Hidden dir)

Convention

For clarity, we add a slash (/) to the end of a directory in the slides. However, in reality it often makes no difference.

`cd`, `pwd`: Changing directory

Try this inside `01-files/`:

```
$ cd c/
$ pwd
$ cd ../
$ pwd
```

Explanation

- `cd`: "change directory"
- `pwd`: "print working directory"
- `../` means "parent directory"

`ls`: Listing directories

Try this inside `01-files/`:

```
$ ls
$ ls -a
$ ls -l
$ ls -la
$ ls c/
```

Explanation

- `ls`: "list"
- `-a` is short for `--all`
- `-l` enables long listing format
- `-la` = `-l` + `-a`

`mkdir`, `rm`: Creating and deleting directories

Try this inside `01-files/`:

```
$ mkdir dir/
$ rm -r dir/
$ mkdir -p dir/subdir/
$ ls dir/
```

Explanation

- `mkdir`: "make directory"
- `-r` is short for `--recursive`
- `-p` is short for `--parents`

`cp`, `mv`: Into and out of directories

Try this inside `01-files/`:

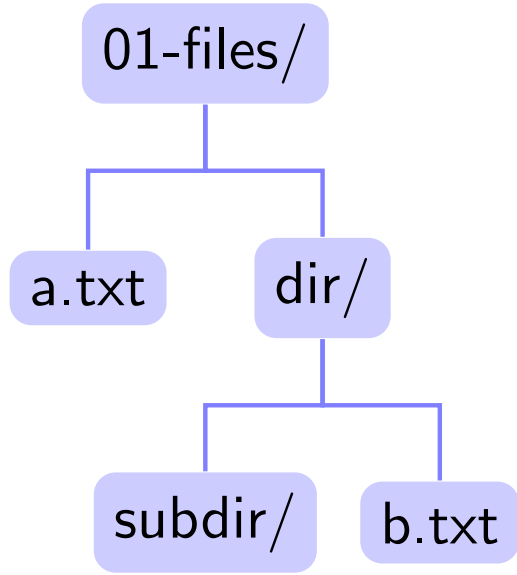
```
$ cp a dir/
$ mv b.txt dir/
$ mv dir/a b.txt
```

Explanation

- Copy `a` into `dir/`
- Move `b.txt` into `dir/`
- Move `dir/a` back into `b.txt`

File tree

Think of any directory as a tree.



Paths

File $\cup$ directory $=$ **path**. [4]

No paths under the same directory can bear the same name.
These **cannot** coexist:

- `01-files/data/`, a directory
- `01-files/data`, a regular file

[4] At least in the scope of this workshop.

Absolute & relative paths

- Paths beginning with `/` are absolute: `/usr/bin/cat`
- Otherwise it is relative: `01-files/`

If you know where you are, you can convert a relative path to an absolute one.

Example

Your location: `/home/you/`
Relative path: `bash-workshop/01-files/`
Absolute path: `/home/you/bash-workshop/01-files/`

Wildcard

`*` is a character to match any number of (including zero) characters.

Exception

Hidden paths will remain hidden unless you explicitly specify the dot: `.*`

Example

	`a/`	`b/`	`a-copy.txt`	`b.txt`
`*`	✓	✓	✓	✓
`a*`	✓		✓	
`*.txt`			✓	✓

Technically it's called a glob pattern but who cares. Also there are other weird symbols like `?` or `[]` but I swear `*` is most of us will ever use.

. and ..

Inside every dir[5] there are two special dirs:

- `./` — current dir
- `../` — parent dir

You can use them in relative paths.

Example

Your location: `/home/you/`
Relative path: `../friend/bash-workshop/01-files/`
(Note that `/home/you/../friend/` is just `/home/friend/`)
Absolute path: `/home/friend/bash-workshop/01-files/`

[5] Except `/`

Challenge

Inside `01-files/`:

- Enter `challenge/`
- Create `backup/`
- Copy `a.txt` into `dir/`
- Move `dir/` into `backup/`
- Verify using `ls`
- Delete `backup/`
- Go to the next section's directory

Solution

```
$ cd challenge/
$ mkdir backup/
$ cp a.txt dir/
$ mv dir/ backup/
$ ls
# Output: backup/  a.txt
$ ls backup/dir/
# Output: a.txt
$ rm -r backup/
$ cd ../../02-cli/
```

Table of Contents

Intro

I. File Tree

II. CLI

III. Pipes

The CLI

CLI stands for **command line interface**, as opposed to a GUI.

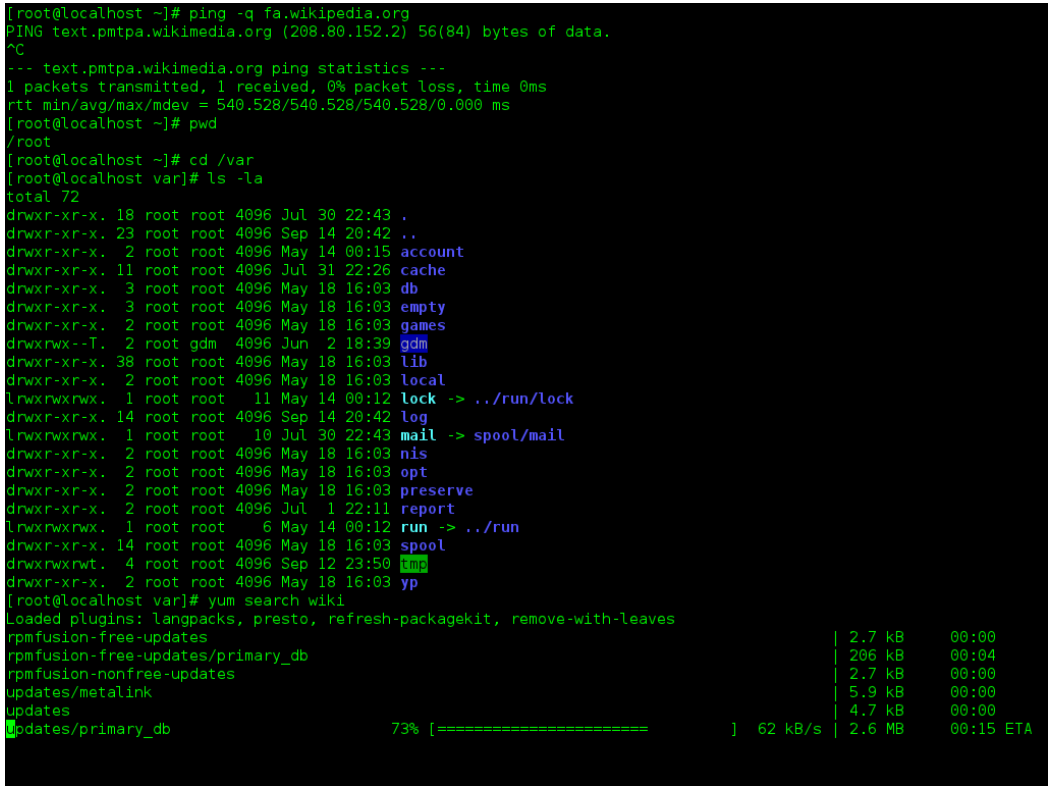


Figure: A stereotypical, Hollywood-like CLI.

Anatomy of a command

```
1 # get first 5 lines of file
2 $ head -n 5 longfile.txt
```

`head`	Executable file somewhere
`-n`	Option (aka flag)
`5`	Argument to `-n`
`longfile.txt`	Argument to `head`

Anatomy of a command

Anatomy of a command:

```
# list all files but *.o recursively in reverse
$ ls -Rr --ignore='*.o'
```

`ls`	Executable in `$PATH`
`-Rr`	Two options: `-R -r`
`--ignore=`	Long option
`'*.o'`	Argument to `--ignore`

Notes

- Not every program uses this `--long-option` convention
- The equal sign after `--ignore` is optional in this command
- `'*.o'` does *not* expand to a list of files. It is simply a string.

I can't possibly remember all `--this` and `--that`!

You don't need to, thanks to **man pages**! (Short for manual pages)

Try:

```
$ man ls
```

If it doesn't work, try https://man.archlinux.org/man/ls.1

Challenge

- Read the man page for `head`
- Experiment with files in `02-cli/`
- Find a command to generate the following:

```
==> p0.txt <==
MANIFESTO OF THE COMMUNIST PARTY.

==> p1.txt <==
I. BOURGEOIS AND PROLETARIANS.
```

Challenge

- Read the man page for `head`
- Experiment with files in `02-cli/`
- Find a command to generate the following:

```
==> p0.txt <==
MANIFESTO OF THE COMMUNIST PARTY.

==> p1.txt <==
I. BOURGEOIS AND PROLETARIANS.
```

Solution

```
$ head -n1 -v p0.txt p1.txt
```

Learning by doing

Inside `02-cli/`, run:

```
$ diff sway.1.conf sway.2.conf
```

Congratulations, you just learned how to use `diff`!

Now, what does this command do?

```
$ comm -12 sway.1.conf sway.2.conf
```

Lifehacks[6]

- Use ↑↓
- `Ctrl-W`: delete one word to the left
- `Ctrl-U`: delete everything to the left
- `Ctrl-K`: delete everything to the right
- `Ctrl-7`: undo (might not work in Git Bash)
- `Ctrl-C`: abort
- `Ctrl-R`: search history
- `Ctrl-L`: clear screen

[6] Should work in most shells.

Table of Contents

Intro

I. File Tree

II. CLI

III. Pipes

stdout

When you `printf`, where does the string go?

Your screen? Yes but also no.

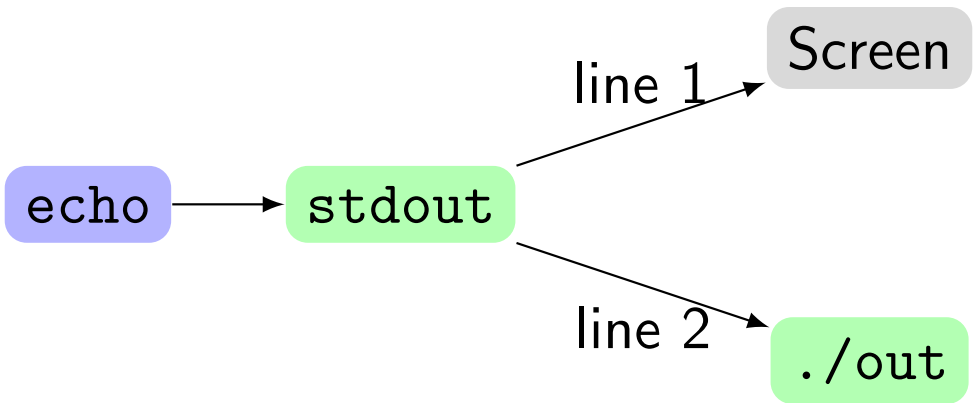
`stdout`, or **standard output**, is a special file where `printf` dumps its output.[7]

[7] Yes, there's `stderr` too, but we won't be talking about it.

Capturing stdout

You can capture the stdout of a command and direct it somewhere else than your screen, such as a file. Try this:

```
$ echo hello          # print to terminal
$ echo hello > out    # write to file
```



> and >>

Try >> instead of >. What happens?

```
$ echo hello >> out
$ cat out
```

Repeat a few times with both > and >>. What's the difference?

> and >>

Try >> instead of >. What happens?

```
$ echo hello >> out
$ cat out
```

Repeat a few times with both > and >>. What's the difference?

Observation

> overwrites the file, but >> appends to it.

stdin

The opposite of `stdout` is `stdin`: standard input. Some programs read from stdin when they expect a path but aren't given any.

Try this in `03-pipes/`:

```
$ ls random/ | head -n 5
```

stdin

The opposite of `stdout` is `stdin`: standard input. Some programs read from stdin when they expect a path but aren't given any.

Try this in `03-pipes/`:

```
1 $ ls random/ | head -n 5
```

Observation

Normally `head` expects a filename, but when none is given, it falls back to stdin — which is what `ls` printed to stdout.

Convention

We sometimes call the vertical bar (`|`) the **pipe** character.

The power of pipes

You can chain commands with pipes. Classic recipe (still in `03-pipes`):

```
$ cat numbers
$ cat numbers | sort
$ cat numbers | sort | uniq
$ cat numbers | sort | uniq | wc
```

The power of pipes

You can chain commands with pipes. Classic recipe (still in `03-pipes`):

```
$ cat numbers
$ cat numbers | sort
$ cat numbers | sort | uniq
$ cat numbers | sort | uniq | wc
```

Observations

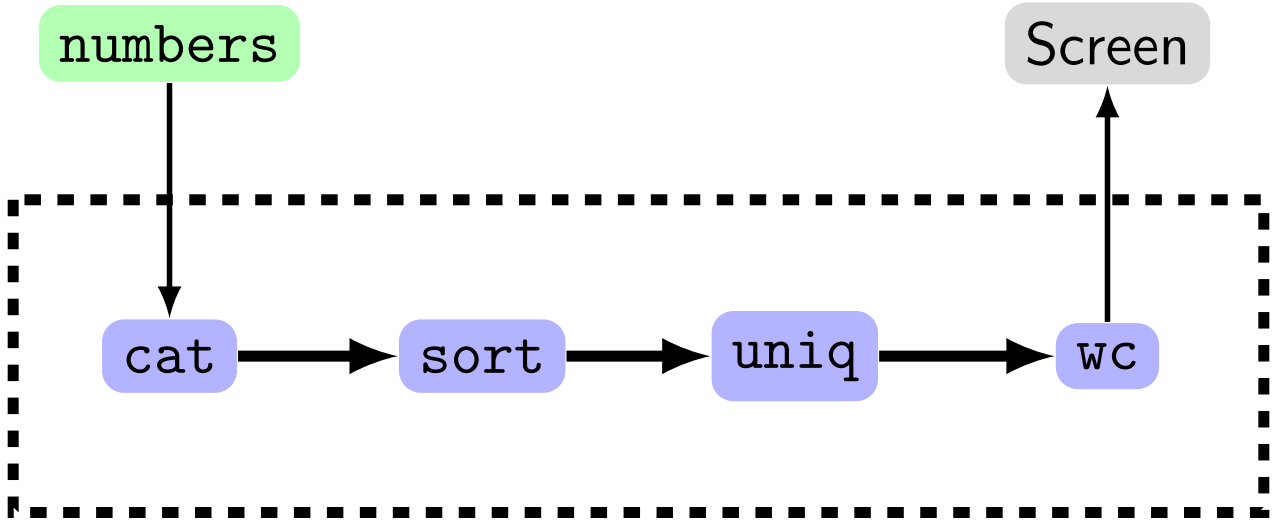
- Each program takes the last one's stdout as stdin
- Only the final program will print to terminal

Explanation

- By default, `sort` sorts stdin in dictionary order (check man page for more)
- `wc` is short for "word count", although the first thing it prints is the number of lines.

Pipes illustrated

```
1 $ cat numbers | sort | uniq | wc
```



Challenge

Can you think of a way to eliminate a pipe? (Hint: man sort)

grep

Try this in `03-pipes`: [8]

```
1 $ ls random/ | grep JI
```

[8] For Mac users, instead of `grep` you might have to type `ggrep`

grep

Try this in `03-pipes`: [8]

```
1 $ ls random/ | grep JI
```

Explanation

`grep` is a powerful tool to match a substring. By default, it takes a file (or stdin), and prints all lines containing a pattern ("JI").

[8] For Mac users, instead of `grep` you might have to type `ggrep`

Challenge

Inside `03-pipes/random/`:

- List all filenames containing "FDU"
- List all filenames containing "UM" (upper and lower cases) (Hint: man grep)
- List all filenames containing "UM" but not "FDU" (upper and lower cases for both substrings)

Solution

```
$ ls | grep FDU
$ ls | grep -i UM
$ ls | grep -i UM | grep -i -v FDU
```

Conclusion

- Files and directories form a tree
- One tool does one thing, but flags specify how
- When in doubt, read documentation
- Chain together tools and unleash immense power

The End

Thank You For Coming!

Credits

- The Free Software Foundation, logo of GNU Bash. https://commons.wikimedia.org/wiki/File:Gnu-bash-logo.svg
- Robin Nicholas, Patricia Saunders and The Open Group, logo of UNIX. https://commons.wikimedia.org/wiki/File:UNIX_logo.svg
- The Free Software Foundation, logo of GNU. https://commons.wikimedia.org/wiki/File:The_GNU_logo.png
- Larry Ewing, logo of Linux. https://commons.wikimedia.org/wiki/File:Linux_logo.jpg
- Poul-Henning Kamp, Beastie. https://commons.wikimedia.org/wiki/File:Daemon-phk.svg
- ZxxZxxZ. Linux command-line. https://commons.wikimedia.org/wiki/File:Linux_command-line._Bash._GNOME_Terminal._screenshot.png