# CS 45, Lecture 2

Shell Tools

Akshay Srivatsan, Ayelet Drazen, Jonathan Kula

Spring 2023

**Outline**

## Contents



## 1 What is the Shell?

To understand what the shell is, we first have to understand the environment in which it was designed:

### UNIX

- The shell (as we recognize it) began with the UNIX *operating system* in 1969.[1]
- UNIX was made at Bell Labs by Ken Thompson and Dennis Ritchie.
- UNIX introduced what is now called "the UNIX philosophy."
- Almost all modern computing is derived from the legacy of UNIX.

In fact, every currently-mainstream operating system has some of UNIX in it. macOS is an actual descendant of UNIX (in particular, the Berkeley Software Distribution, or BSD), and Linux is designed to be UNIX-compatible, even though it was made independently. Even Windows, which historically held an entirely different philosophy and design, now bundles a bundles a Linux compatibility layer, which transitively gives it UNIX compatibility as well.

Now, we said UNIX is an operating system, but what exactly does that mean?

[1] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Commun. ACM*, 17(7):365–375, jul 1974

## 1.1 What is an Operating System?

An operating system, at its core, is an abstraction layer. A computer contains many different parts, each of which has a different purpose:

### Anatomy of a Computer

**Input** Keyboards, Mice, *Serial Ports*, etc.

**Output** Screens, *Serial Ports*, Speakers, etc.

**Storage** Memory (RAM), Disks, Disc Readers, etc.

**Compute** CPUs (math), FPUs (math with decimals), GPUs (math with matrices)

**Networking** Ethernet, Wi-Fi, *Serial Ports*, etc.

**Misc.** Fans, Power Supplies, Sensors, etc.

Different computers have different parts serving these purposes. My laptop, for example, has very different hardware than the computer running Carta. However, they both use the same operating system kernel (i.e., Linux), and I can therefore run the same programs on both.

One recent example you might be familiar with—while older Macs used CPUs made by Intel, newer Macs use CPUs made by Apple. These CPUs are incompatible with each other; essentially, they speak different machine languages, and can each only understand programs written in their native language. However, since they both have the same operating system (macOS), programs written for an Intel Mac can run on an ARM Mac; macOS automatically translates programs written in the Intel language to the ARM language, hiding the low-level details from both the programs and their programmers.

**Definition 1.1** (kernel)**.** An Operating System Kernel is a program that abstracts over different hardware, allowing the same software to run on different computers.

You might notice that serial ports show up quite a lot on this list. At the time UNIX was designed, serial ports were ubiquitous—any time you wanted to connect two devices together, you'd use a serial port. Serial ports at the time were like USB ports today—you'd use them for everything. Even today, you can still see the legacy of serial ports hiding in the design of most operating systems.

As an aside, computers didn't have screens at one point, not long before UNIX. Instead, they were connected (via a serial port) to a *typewriter*. When a computer "printed" output, it literally printed it out onto a piece of paper. Over time, these were replaced with "video terminals"; these things had a video screen instead of a piece of paper, usually tall enough to show 25 lines of text. While computer scientists still called it "printing", any output sent by the computer was instead displayed on the screen. Eventually, we progressed to graphical displays like the one you're reading this on. Instead of a dedicated video terminal, you can now open a "terminal emulator" (usually just called a "terminal"), which is a window that pretends to be a video terminal.

### Userspace

- A kernel by itself is kind of useless.
- Abstractions are great, but we want to *do* something with our computers.
- This is where userspace comes in.

**Definition 1.2** (userspace)**.** Userspace is the set of programs that come bundled with an OS kernel, which allow a user to perform various tasks.

Almost everything we'd call an "program" (or an "app") is a userspace program. While we can install programs/apps after-the-fact, every operating system must come with some set of them installed; otherwise, the operating system wouldn't be able to do anything.

| Operating System | Shell | Type | How you start programs |
|---|---|---|---|
| Windows | `explorer.exe` | Graphical | Start Menu, Desktop |
| macOS | Aqua | Graphical | Dock, Launchpad |
| iOS, Android | Home Screen | Graphical | Tap icon |
| Linux | GNOME, KDE, XFCE, … | Graphical | Various |
| Windows | `cmd.exe` | Text | Type name of `.exe` file |
| UNIX | `sh` | Text | *The rest of this lecture.* |
| Linux | `bash` | Text | Same as `sh` |
| macOS | `zsh` | Text | Same as `sh` |

Table 1: Shells across common operating systems

An operating system's userspace almost always contains the following:

- A text editor.
- A clock.
- A calendar.
- A calculator.
- An email program.
- A web browser.
- A file browser.
- An "app store" (package manager).

### Running Programs

- Now that we have a bunch of programs installed, we want to run them.
- We need something that wraps up all the programs and provides a common interface to them.

This common interface is what we call the SHELL.

**Definition 1.3** (shell)**.** A SHELL is the outermost layer of an operating system; it lets a user run userspace programs, which in turn let a user interact with their computer's hardware.

**Definition 1.4** (operating system)**.** An OPERATING SYSTEM is the combination of a kernel, a set of userspace programs, and a shell.

Modern operating systems have many different kinds of shells, depending on what they're used for. Some are graphical, some are text-based. While graphical ones are far more common in general nowadays, text-based ones are far more common for developers. Given that you're taking this class, you probably already know how to use graphical ones (you're using one right now to read this), and you probably want to know how to use a text-based one (which is the rest of this lecture!).

### Types of Shell

Modern UNIX-like operating systems use more modern shells, like `bash` and `zsh`, but they're all backwards-compatible with the classic `sh`. While Table 1 lists the default on each platform, some platforms let you change the default shell, and there are many `sh`-compatible shells available.

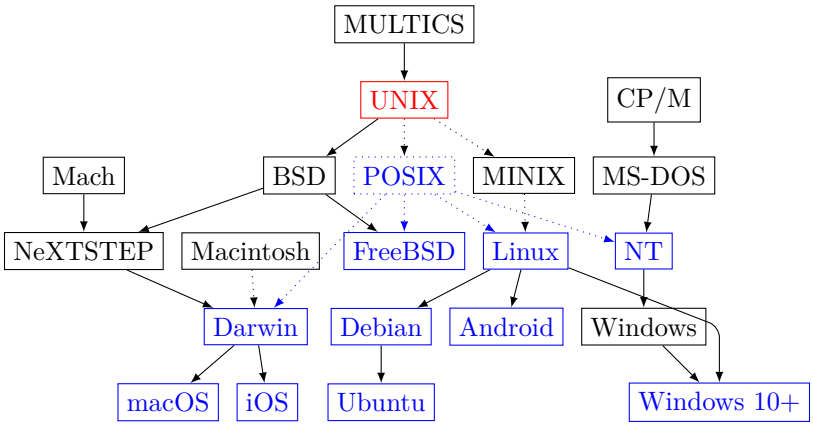
Windows also has a more modern UNIX-shell derivative called PowerShell, which is designed to be a hybrid of the traditional Windows command line, the UNIX shell, and Windows' native object-oriented interfaces. It's not fully compatible with the UNIX shell, and is much less widely used, but it's worth checking out if you're curious about alternative shells; it has some cool features not found on traditional UNIX shells.

While all of these shells can *start* programs, only the UNIX shell (and its derivatives) can *combine* them.[2]

This unique property, especially in combination with The UNIX Philosophy, has resulted in the UNIX shell growing into a powerful software ecosystem.

Since we're talking about operating systems anyway, let's take a moment to look at the (abridged) history of the common operating systems in the world today:

### Modern Operating Systems




There's a lot going on here, but the main takeaway is that *every* widely-used modern operating system is, in some way, descended from UNIX. The Apple OSes, macOS and iOS, are true descendants of UNIX, via the Berkeley Software Distribution (BSD). The Linux-based operating systems, including Android, are mostly-compatible clones of UNIX; they are compatible with the Portable Operating System Interface (POSIX), which was directly based on UNIX. Even recent versions of Windows, despite having a completely separate history from UNIX, have adopted POSIX via the Windows Subsystem of Linux (WSL), making them (technically) UNIX-compatible. Again, UNIX was super influential—the entire modern world of computing is based on it.

## 1.2 The UNIX Philosophy

The UNIX philosophy is a way of thinking about what a program should do.

### Original

As described in the Bell System Technical Journal in 1978 [3]:

1. Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features."
2. Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.
3. Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.
4. Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.

That's written very confusingly, but it's all just a really wordy way of saying this:

[2] Okay, technically the other shells *can* also combine programs to some extent, but the UNIX shell can do so to a much greater extent.

[3] M. D. McIlroy, E. N. Pinson, and B. A. Tague. UNIX time-sharing system: Foreword. *The Bell System Technical Journal*, 57(6):1899–1904, July 1978

**Simplified**

**The UNIX Philosophy**

Build lots of small tools, each of which does exactly one thing well, but which can be combined to do more powerful things.

The UNIX Philosophy is also deeply tied to the UNIX file abstraction. Remember how we said that the job of an OS kernel is to abstract over different hardware? Well, in UNIX (and UNIX-derived OSes):

## 1.3 The UNIX File Abstraction

**The UNIX File Abstraction**

- In UNIX, *everything is a file* (including hardware!).
- Most files are text.
- Programs which operate on text can operate on almost everything.

The idea that *everything is a file* is not an exaggeration. Want to write to your screen? Write to `/dev/fb` (the "framebuffer"). Want to get input from the mouse? Read from `/dev/input/mice`. Want to play a beep from the speaker? Write to `/dev/snd/pcm*`. While this has reduced a bit on modern OSes, and there are now higher-level abstractions for many things, UNIX-based operating systems model an insane number of different things as files.

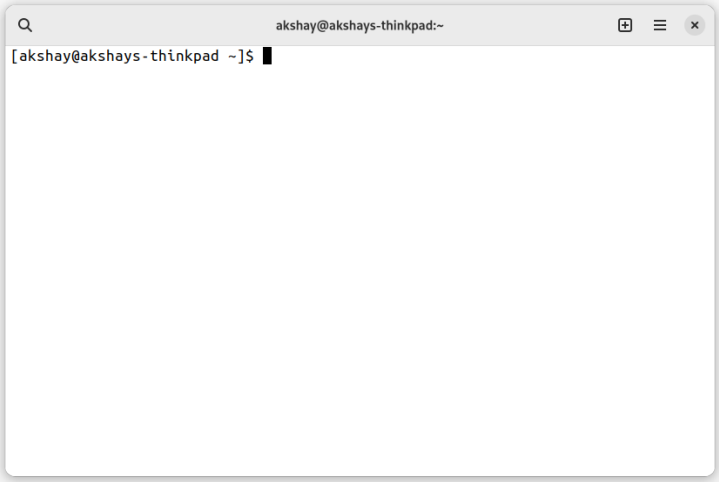
The combination of the UNIX Philosophy and the UNIX File Abstraction means we have hundreds of small, single-purpose programs which operate on text files, and we have every possible thing the computer could do represented as text files. The UNIX shell lets us combine these programs and these devices/text files to do… whatever we want, really.

# 2 The UNIX Shell

When you open the "Terminal" application on your computer, you'll be greeted by the UNIX shell (from now on, just "the shell"').[4]

**The UNIX Shell**

[4] You can open the "Terminal" application the same way you'd open a normal application; on macOS it's located in the "Utilities" folder of "Applications", on Windows it's in your start menu (it might be called "Ubuntu"), and on Linux it'll be in your desktop environment's normal app launcher.




In particular, the shell will give you a "prompt"; a sequence of information, followed by a cursor where you can type commands.

### The Prompt

*Example* 2.1 (prompt). An default shell prompt might look like this:

```
[akshay@akshays-thinkpad ~]$
```

This PROMPT will print every time the shell is ready to accept another command. It probably looks different on your computer, since different shells have different defaults. Regardless of what it looks like right now, you can customize it to look like whatever you want. In most cases, you can get back to it by pressing CTRL-C on your keyboard.

**Definition 2.2** (prompt)**.** The prompt is a sequence of information printed by the shell when it's ready to accept a new command.

Let's look at the pieces of information in more detail:

### The Prompt

*Username*

*Example* 2.3 (prompt: username). This part of the prompt is your username:

```
[akshay@akshays-thinkpad ~]$
```

This is probably the same as the username you use to log into your computer. By default, you're auto-logged into the shell using your normal user account.

### The Prompt

*Hostname*

*Example* 2.4 (prompt: hostname). This part of the prompt is your computer's hostname:

```
[akshay@akshays-thinkpad ~]$
```

This is your computer's name on whatever network it's connected to. Generally you don't really care about this unless you have multiple computers.

#### The Prompt

*Current Directory*

*Example* 2.5 (prompt: current directory). This part of the prompt is your current directory.

```
[akshay@akshays-thinkpad ~]$
```

This is your WORKING DIRECTORY; "directory" is just a fancy name for "folder", like you'd have in Windows Explorer or macOS Finder.

**Definition 2.6** (directory)**.** A DIRECTORY is a folder. It can contain files and other directories.

**Definition 2.7** (working directory)**.** Your WORKING DIRECTORY is the directory you're "in"; until you change your working directory, you can see only the files and folders in this directory. This is also often called your CURRENT DIRECTORY. It can be abbreviated as `..`

Just like in Windows Explorer or macOS Finder, you're "in" a particular directory at any point in time, in which case you can see the files in that directory.

By default, you start in your HOME DIRECTORY, which is the folder that contains `Documents`, `Downloads`, `Pictures`, `Videos`, etc. The home directory is abbreviated as a tilde (~) because it's so common.

**Definition 2.8** (home directory)**.** Your HOME DIRECTORY is the directory all your files are in. Everything "under" your home directory belong to you, while everything "above" it belongs to someone else (probably the OS itself).

The shell is a command interpreter: you type in commands, and it does them (or gives you an error). Unfortunately, the commands were all named half a century ago when bytes were expensive, so the names cryptic and short. Let's go through a few useful commands to get started.

# 3 Basic Commands

## 3.1 Directories

#### Listing Files

In Windows Explorer or macOS Finder, the current directory is always visible. In the shell, you have to ask for the list of current files manually.

*The LIST command is called `ls`.*

*Example* 3.1 (ls). On my computer, the results look like this:

```
[akshay@akshays-thinkpad ~]$ ls
Desktop    Downloads  Music     Public     Videos
Documents  Dropbox    Pictures  Templates
[akshay@akshays-thinkpad ~]$
```

These are all the SUBDIRECTORIES of my home directory.

**Definition 3.2** (subdirectory)**.** A SUBDIRECTORY is a directory contained within another directory.

### You are here

- Before we go exploring, let's do one more command in the home directory.

*The PRINT WORKING DIRECTORY command is called `pwd`.*

*Example* 3.3 (pwd). Print the current working directory:

```
[akshay@akshays-thinkpad ~]$ pwd
/home/akshay
[akshay@akshays-thinkpad ~]$
```

Note that `~` is actually short for `/home/akshay`. Its PARENT DIRECTORY is called "home". The parent directory of "home" doesn't have a name at all!

**Definition 3.4** (parent directory)**.** The PARENT DIRECTORY of a subdirectory is the directory which contains the subdirectory.

**Definition 3.5** (root directory)**.** The ROOT DIRECTORY is the topmost directory on the filesystem. It's often called `/`.

**Definition 3.6** (filesystem)**.** The FILESYSTEM is the hierarchy of directories which contain every file on your computer. Some of these files are "real", i.e., they exist on your hard drive, and some are "virtual", i.e., they're made up by the OS kernel to represent hardware.

N.B.: your computer might say something slightly different depending on your OS. I think macOS would say `/Users/<username>`.

### Changing Directories

- The home directory isn't too interesting on its own.
- Let's go somewhere you probably know well: the Desktop!
- In Explorer or Finder you could just click on a folder to enter it.

*The CHANGE DIRECTORY command is called `cd`.*

*Example* 3.7 (cd). Change (`cd`) into the `Desktop` directory:

```
[akshay@akshays-thinkpad ~]$ cd Desktop
[akshay@akshays-thinkpad Desktop]$
```

Note that my prompt changed to say `Desktop` instead of `~`.

### Not Changing Directories

- There's a special name that always means "the current directory": `.`
- `cd .` says "change directory to the current directory".

*Example* 3.8 (cd .). Don't change directories:

```
[akshay@akshays-thinkpad Desktop]$ pwd
/home/akshay/Desktop
[akshay@akshays-thinkpad Desktop]$ cd .
[akshay@akshays-thinkpad Desktop]$ pwd
/home/akshay/Desktop
```

| Shortcut | Name |
|---|---|
| `~` | Home Directory |
| `/` | Root Directory |
| `.` | Current Directory |
| `..` | Parent Directory |

Table 2: Directory Shortcuts

| Command | Description | Argument | Required |
|---|---|---|---|
| `ls` | List Directory | Directory Name | No, defaults to `.` |
| `cd` | Change Directory | Directory Name | No, defaults to `~` |
| `pwd` | Print Working Directory | N/A | N/A |
| `mkdir` | Make Directory | Directory Name | Yes |
| `rmdir` | Remove Directory | Directory Name | Yes |

Table 3: Directory Commands

From now on, I'm going to omit the part of the prompt before the `$` because it's not very useful for our purposes, and it varies from computer to computer anyway.

### Making Directories

- Now that we're on the desktop, let's create a new directory to do some experiments in.
- This is the equivalent of right-clicking and selecting "New Folder".

*The* MAKE DIRECTORY *command is called* ***`mkdir`****.*

*Example* 3.9 (mkdir). Create a directory called "cs45-test-directory" and `cd` into it:

```
$ mkdir cs45-test-directory
$ cd cs45-test-directory/
```

You can go back to the Desktop by typing `cd ..` or `cd ~/Desktop`.

In general, `..` is a special name which always refers to the parent of the current directory. Regardless of where you are, you can always go to the parent directory by typing `cd ..`.

Now that we can make directories, you might be wondering how to delete them. The process is actually almost the same, just with a different command.

*The* REMOVE DIRECTORY *command is* ***`rmdir`****.*

This is used almost the same way as mkdir; one thing to note is that `rmdir` can only be used on an empty directory.

Let's review what we've covered about directories:

### Directory Review

**Definition 3.10** (argument)**.** An ARGUMENT is an input provided to a command. This may be a filename or arbitrary text, depending on the command. It is passed to a command by specifying it after the command name. By default, arguments are separated by spaces; if you want to provide an argument with a space, use double- or single-quotes around the argument.

## 3.2 Files

Now that we can create, delete, and move around directories, let's start working with files. As mentioned in subsection 1.3, everything in UNIX is a file, so (in a way) we're actually learning to work with *everything*.

However, before we work with files, let's take a brief diversion to look at something a little more basic: input/output.

### Output

- Sometimes we just want to print something out, like "hello, world".

*The* PRINT *command is called* `echo`.

*Example* 3.11 (echo). Print the text "hello, world":

```
$ echo "hello, world"
hello, world
$
```

The name "echo" is a bit less intuitive than the ones we've seen before, but it makes sense: it "echoes" back anything you tell it.

If you're familiar with programming in C, you might also like the PRINT FORMATTED command, `printf`. You can use it the same way as the C `printf` function, except you don't need parentheses.

*Example* 3.12 (printf). Print the text "hello, world" using `printf`:

```
$ printf "%s, %s\n" "hello" "world"
hello, world
$
```

### Input

- Sometimes we want to read input from the user.
- Unfortunately, this is trickier: there are multiple commands which can be used for input. Let's use the simplest one.

*The* CONCATENATE *command is called* `cat`. It can also be used for input.

*Example* 3.13 (cat). Read text from the user:

```
$ cat
this is a test
this is a test
line 2
line 2
$
```

The `cat` command will print out whatever you type into it… forever. To get it to stop, you can either "kill" it by pressing CTRL-C, or tell it "end of file" by pressing CTRL-D. Note that this is actually the CONTROL key, even on Macs. Using the COMMAND key will not work.

Aside: Windows and Linux users might be wondering how you copy and paste from the terminal if CTRL-C kills programs instead of copying anything. The answer is: it's complicated. The UNIX shell predates

CTRL-C and CTRL-V for copy-and-paste, so they use those shortcuts for other things. Different terminal emulators (the program called "Terminal" on your computer) have different ways of doing copy-and-paste; mine uses CTRL-SHIFT-C and CTRL-SHIFT-V, but yours might use something else. Mac users don't have to worry about this, since they can use CMD-C and CMD-V as usual.

Enough on I/O (as input/output is often called); let's get back to files.

### Creating Files

- There are a few different ways to create files. Let's start with the simplest.

*The TOUCH FILE command is called,* ***`touch`****.* While we don't care about "touching" files as such, it has the handy side effect of creating files.[5]

*Example* 3.14 (touch). To create a file called "text.txt":

```
$ touch test.txt
$ ls
test.txt
$
```

We can also rename or move the file we just created.

### Renaming Files

*The MOVE FILE command is called* ***`mv`****.* It can also be used to rename files.

*Example* 3.15 (mv). To rename a file called "text.txt" to "empty.txt":

```
$ mv test.txt empty.txt
$ ls
empty.txt
$
```

### Deleting Files

*The REMOVE FILE command is called* ***`rm`****.*

***This is irreversible!***

This command is dangerous! It does **not** move the file to a "trash" folder; it permanently and irreversibly deletes it.

*Example* 3.16 (rm). To remove a file called "text.txt":

```
$ rm test.txt
$
```

[5] "Touching" a file means changing its last-modified timestamp to the current time; essentially like opening a file, saving it, and closing it again. This isn't useful very often, so most people think of `touch` as the "create file" command.

### Writing to Files

- We have a problem though: the file we created is empty. We can't do much with a bunch of empty files.
- We can check this by running `ls` with a special FLAG asking for extra info (including the file size).

**Definition 3.17** (flag)**.** A FLAG is a special argument to a function which configures its behavior. By convention, flags can be short (a single letter) or long (a word), and start with either one or two hyphens to distinguish them from normal arguments.

*Example* 3.18 (ls -l)*.* To print extra information about files:

```
$ ls -l
total 0
-rw-r--r-- 1 akshay akshay 0 Dec 18 11:59 test.txt
```

This prints out a bunch of information we don't really care about right now: the file permissions, the file's owner, the file's group, and the file's last-modified time. What we do care about is the "0" highlighted in red above: the file is zero bytes long.

As an aside, the real purpose of the `touch` command we just used is to update the modification time of the file. If you touch the file and run `ls -l` again, you should see the last-modified time update.

One very useful flag which is supported by almost every command is `--help`. This will usually print out information about how to use that command. On some commands the short form `-h` will also work, but that's sometimes used for something else.

### Writing to Files

- Everything is a file, including the output of our commands.
- By default, this is called STANDARD OUTPUT, and goes to our terminal.
- The shell lets us REDIRECT standard output to go to a file instead.

*Example* 3.19 (output redirection)*.* To create a file called "hello.txt" with the contents `hello, world`:

```
$ echo "hello, world" > hello.txt
$
```

To append to an existing file, you can use `>>` instead of `>`.

### Reading from Files

- Just like STANDARD OUTPUT, the input to our programs is also a file.
- By default, this is called STANDARD INPUT, and comes from our terminal.
- The shell also lets us REDIRECT standard input to come from a file.

*Example* 3.20 (input redirection)*.* To print a file called "hello.txt":

```
$ cat < hello.txt
hello, world
$
```

| Operator | File | Overwrite? |
|---|---|---|
| `<` | `/dev/stdin` | |
| `>` | `/dev/stdout` | Overwrite |
| `>>` | `/dev/stdout` | Append |
| `2>` | `/dev/stderr`[7] | Overwrite |
| `2>>` | `/dev/stderr` | Append |

Table 4: UNIX Shell Redirection Operators

This is actually slightly redundant, since the `cat` command lets us specify a specific file to use as input directly, so `cat hello.txt` would do the exact same thing. The redirection approach is more general though: it works with any command that accepts input.

When the shell runs a program, it sets up its input and output using special files.[6]

### I/O(/E?)

**Definition 3.21** (standard input)**.** STANDARD INPUT (`/dev/stdin`) is the file from which a program reads its input.

**Definition 3.22** (standard output)**.** STANDARD OUTPUT (`/dev/stdout`) is the file to which a program writes its output.

**Definition 3.23** (standard error)**.** STANDARD ERROR (`/dev/stderr`) is the file to which a program writes its error messages.

Since these files are so widely used, the shell provides easy ways to redirect them. We've already seen a few of these.

### Redirection Operators

You might have seen the `>` operator before, in the form `> /dev/null`. `/dev/null` is a special file that discards anything written to it, so it's common to use it when the output of a command is unimportant.

As it turns out, if we're doing multiple commands in a sequence, we don't even need to write the results out to a file until the very end—we can use PIPES to send the output of one command directly into another command.

# 4 Pipes

### Environment Variables

Some programs need configuration that's too annoying to provide as arguments every time.

**Definition 4.1** (environment variable)**.** An ENVIRONMENT VARIABLE is a configuration value that's set globally by a program, which applies to itself and any other programs it runs.

We'll talk about environment variables some more in *Lecture 5: Command Line Environment*. For now, we just need to know that they exist and are widely used.

### All the Environment Variables

*The ENVIRONMENT VARIABLE command* ***`env`*** *prints all the environment variables which are currently set.*

[6] Technically, these are FILE DESCRIPTORS rather than files, but the distinction isn't very important for this class so we're glossing over it. If you're interested in learning more, come talk to us or take CS111.

*Example* 4.2 (env). To print every environment variable:

```
$ env
MAIL=/var/spool/mail/akshay
PWD=/home/akshay
XDG_SESSION_TYPE=wayland
PATH=/usr/local/bin:/usr/bin:/usr/local/sbin
HOME=/home/akshay
USERNAME=akshay
[...]
```

This is quite long on my computer…

### Connecting Programs

- Let's see how many environment variables we have!
- *The* WORD COUNT *command is called* `wc`.
- It has a flag `--lines` (or `-l` on Macs) which counts lines in its input instead of words.

Now, we want to get the output of `env` into `wc`. There's two ways we could do this:

*Example* 4.3 (count environment variables with a temporary file). We can write the output into a temporary file, and give it as input to `wc`:

```
$ env > /tmp/env.txt
$ wc -l < /tmp/env.txt
78
```

This is kind of annoying though…we need to do two separate commands, and we need to create a temporary file somewhere.

As an aside, it's conventional to put temporary files in `/tmp`. Most systems will keep `/tmp` entirely in RAM, so any files there will automatically get deleted when you reboot.

However, we can actually do this in a one-liner, without needing any temporary files at all.

### Pipes

We want to send the output of `env` into `wc -l`:

*Example* 4.4 (count environment variables with a pipe). We can connect the output of `env` and the input of `wc` with a PIPE:

```
$ env | wc -l
78
```

### Benefits of Pipes

**Definition 4.5** (pipe)**.** A PIPE is a direct connection between the output of one program and the input of another. It can be set up using the `|` (pipe) operator, which connects `stdout` of whatever is on the left with `stdin` of whatever is on the right.

Pipes are superior to temporary files for several reasons:

- They are **parallel**: the programs on the left and right can run at the same time. This is especially useful when the programs need to do lots of computation or external I/O; the right-side program can process the data it has already read, while the left-side program is waiting for or computing data to write.
- They are **lazy**: the program on the right can read exactly as much data as it needs from the program on the left. In many cases, the right-side program may only need part of its input, in which case the left-side program doesn't need to compute the rest. This is especially useful when the left-side program is infinite or long-lived.

### More Piping

*Example* 4.6. <only@1>[the first $n$ environment variables] With the `head` command, we can extract only the first few lines from a file:

```
$ env | head --lines=3
MAIL=/var/spool/mail/akshay
PWD=/home/akshay
XDG_SESSION_TYPE=wayland
$
```

*Example* 4.7. <only@2>[random numbers] We can lazily evaluate part of an infinitely long "file" such as `/dev/random`:

```
$ cat /dev/random | hexdump | head --lines 1
0000000 4730 003c 6c22 1d16 49ef 6eff 91b2 a9f0
```

# 5 Conclusion

### Getting Help

- The shell is far more complicated than we can possibly cover in an 80-minute lecture (or even a quarter-long class, honestly).
- Today's lecture was just the starting point—try things out and explore! Just like anything other skill, it's super important to practice using the shell on your own. It'll feel slow and clunky at first, but you'll get the hang of it soon!
- Most commands have lots of flags and options.
- We already talked about the `--help` flag, which usually gives you a brief summary of how to use a command.

### The System Manual

- One super-useful resource is the UNIX system manual, which is pre-installed on most UNIX-like systems.
- *The* MANUAL *command is* ***man***; it takes as an argument the name of a command, and it displays the manual page ("man page").
- If you don't know the name of a command, you can search the manual using the command `apropos` (or, equivalently, `man --apropos`).

*Example* 5.1. <only@4>[man wc] To open the UNIX manual page for the `wc` word-count tool:

```
$ man wc
```

*Example* 5.2. <only@5>[man man] To open the UNIX manual page for the manual itself:

```
$ man man
```

Man pages open in what's called a PAGER; this is a program that makes a long file readable on a short terminal. You start at the beginning of the file, and you can scroll up or down using the arrow keys. You can quit by pressing `q`.

**To be continued...**

- We'll continue exploring shell tools in Lecture 3: *Data Manipulation and Shell Scripting*.
- We'll post Assignment 1 soon, but it'll only be due on Wednesday, January 19th (fourteen days from now). It'll cover Lectures 2, 3, and 4 (this week and next week)

**To be continued... (Continued)**

- In the meantime: try doing file management from the terminal. We didn't cover every command you'll need, so if you don't know how to do something, try searching the manual using `apropos` or searching the web.
- Take a look at the UNIX filesystem hierarchy layout, documented in `man hier`. You don't have to understand everything in there (lots of it is archaic/irrelevant to modern PC usage), but it's good to know roughly what's stored where.
- Use `man` pages, http://cheat.sh/, or https://devhints.io/bash to find out more about shell commands.

***Be Careful!***

The shell often doesn't warn you when you're doing dangerous things! Be sure to read the `man` page before running commands you find on the internet. Be especially careful with the REMOVE FILE command, `rm`, or when using the `>` (overwrite) operator.

Normally, most OS-critical files on your computer are protected using FILE PERMISSIONS, which we'll learn about in a week or two. However, there's a special command (which we saw briefly on Monday) which can override file permissions and do potentially catastrophic things.

**Sudo**

***Using `sudo` responsibly***

- Some commands require the use of `sudo`, the SUPERUSER DO command.
- This gives that command full access to do anything on your computer!
- Sometimes `sudo` won't even ask for your password!
- If you're using `sudo`, make sure you know what the command after it will do.
- `sudo` is necessary for certain tasks (we'll see some in the next few lectures), but it's always good to be careful around it.

**Definition 5.3** (superuser)**.** The SUPERUSER is a special account, often called the ROOT ACCOUNT, with full power over a UNIX system.[8] They are the equivalent of the Windows Administrator account. Some regular accounts, called "sudoers", have the ability to act as the superuser using the `sudo` command.

And it's not just me who thinks that `sudo` can be dangerous, the first time you run `sudo` it'll give you a warning about how dangerous it is:

### Sudo Warning

```
We trust you have received the usual lecture from the local System
Administrator. It usually boils down to these three things:

    #1) Respect the privacy of others.
    #2) Think before you type.
    #3) With great power comes great responsibility.
```

All that said, there are legitimate reasons to use `sudo` (for example, installing software). Just be careful around it, and don't run commands as `sudo` just because you got a "permission denied" error; make sure there *is* a reason you have to use `sudo`.

Here are a few interesting commands to look up/try out before next class; try looking up their `man` pages and how to use them:

### Interesting Commands

`head`: Get the beginning of a file (or pipe).

`tail`: Get the end of a file (or pipe).

`grep`: Search within a file.

`sed`: Find-and-replace.

`cut`: Get a specific "column" of a file (e.g., a CSV file).

`ping`: Test your internet connection.

`sort`: Sort lines in a file.

`uniq`: Remove duplicate lines in a file.

`exit`: Exit the terminal.

Note that `uniq` requires its input to be sorted; it's not super clear why the authors made this decision, it's generally good practice to pipe through `sort` before piping into `uniq`.

### Questions?

[8] Recent versions of macOS have an extra layer of protection, called "System Integrity Protection", that stops even the superuser from doing *really* dangerous things. However, the superuser can still do a lot of things an ordinary user cannot.


It's a UNIX system. I know this.

# References

[1] M. D. McIlroy, E. N. Pinson, and B. A. Tague. UNIX time-sharing system: Foreword. *The Bell System Technical Journal*, 57(6):1899–1904, July 1978.

[2] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Commun. ACM*, 17(7):365–375, jul 1974.