# CS 45, Lecture 6

Command Line Environment

Akshay Srivatsan, Ayelet Drazen, Jonathan Kula

Spring 2023

**Outline**

## Contents

**Announcements**

- Assignment 1 is due today. Reach out if you don't think you will be able to get it done in time.
- Assignment 2 is out! It's due a week from today on Wednesday, April 26th at 11:59 PM.

## 1 Review

**Recap**

In the previous lecture, we saw:

- How to edit files in the terminal
- How to enter/exit a full screen program (`vim`)

In this lecture, we will see:

- How to configure and customize your shell
- How to multitask in the terminal
- How to run multiple programs side-by-side

Before we get into that, there are a few similar terms that are often confused; let's get into what they mean and how they're different.

### Terminal vs. Shell vs. Command Line

**Definition 1.1** (terminal)**.** The TERMINAL is the window you open. Think of it like a web browser.

**Definition 1.2** (shell)**.** The SHELL is the program you use to launch other programs. Think of it like Google.

**Definition 1.3** (cli)**.** A COMMAND LINE INTERFACE (CLI) is a generic term for a text-based program which runs within a terminal. Think of this like "the web". A CLI PROGRAM or a TUI PROGRAM is like a website.

You may often see a few acronyms in the context of user interfaces:

**CLI:** Command-Line Interface (like the shell)

**TUI:** Text User Interface (like `vim`)

**GUI:** Graphical User Interface (like Microsoft Word)

## 2 The Environment

We've already seen how you can control the behavior of CLI programs using flags, but there's another way to configure them. Each program runs in what's called an "environment".

### Contents of the Environment

The "environment" a program runs in includes several things:

- The user who's running it
- The files on the filesystem
- Environment variables (configuration variables)
- `stdin` and `stdout` (and `stderr`)

Each of these affects running programs in different ways. Some of them control what a program can or cannot do, others control the default behavior of programs.

### 2.1 Configuration

#### Input/Output

We already saw this in Lecture 2, but we can control the default input and output files of a program using REDIRECTION, i.e., the `<`, `>`, `>>`, and `|` operators.

By default, input comes from the terminal (`/dev/tty*` or `/dev/pts/*`); you can see the name of the "controlling terminal" of a program by running `tty`.

A CONTROLLING TERMINAL is essentially which terminal window a program is running in.

Input and output can be redirected, but a program is bound to a specific window. When that window is closed, the program will exit.

The command `tty` stands for "teletypewriter". A teletypewriter was, essentially, a typewriter than was plugged into a computer and used for input/output. As we talked about in Lecture 2, these got replaced later with "video terminals" (which had a basic screen instead of a piece of paper), which were themselves later replaced with terminal emulators (the program called "Terminal" on your computer). Nowadays, terminal emulators tell your OS to create what's called a "pseudo-terminal" (or "pty"), which lets them pretend to be a terminal even though they're just normal programs.

### Environment Variables

Environment variables are a way to configure a program's default behavior.

We've already seen shell scripting variables, environment variables are basically the same thing except they're "exported" so other programs can use them.

For example, the `$PATH` variable determines where programs can be located. If a program isn't found "on your `$PATH`", you'll get a "command not found" error.

Other common variables:

**`$TERM`:** Which terminal you're using.

**`$USER`:** Your username

**`$EDITOR`:** Which editor you prefer

**`$PWD`:** Your current directory

### PATH

My `$PATH` looks like this:

```
/home/akshay/.local/bin:/usr/local/bin:/usr/bin:/usr/local/sbin:
/var/lib/flatpak/exports/bin:/usr/bin/site_perl:
/usr/bin/vendor_perl:/usr/bin/core_perl
```

This is a list of directories, where each directory is separated by colons (`:`).

When you run a program like `grep`, the shell looks in each directory on your `$PATH` from left to right.

### Setting Environment Variables

You can "export" a shell variable to turn it into an environment variable as follows:

```
export MYVAR="hi"
python -c 'import os; print(os.getenv("MYVAR"))'
```

Note that even though I ran a Python program, not a shell program, it was able to read the value of `$MYVAR`.

You can temporarily set an environment variable as follows:

```
MYVAR=hi python -c 'import os; print(os.getenv("MYVAR"))'
```

In this case, only that one line gets run with `$MYVAR` set to "hi"; after that, `$MYVAR` goes back to its old value (or becomes undefined).

Environment variables are "inherited"—child programs (and their descendants) will be able to see their value, but **not** any other programs.

Environment variables are ubiquitous in UNIX, and pretty much every programming language has a way to read and write them. The shell makes it incredibly easy by treating them as normal variables; in other languages, you'd have to call a function to get or set them.

By default, environment variables only persist until you exit the shell (or close the terminal window). We'll see later how to make them persist.

## 2.2 Permissions

### Users and Groups

We also talked about this a bit in Lecture 2, but every command you run runs as a specific user.

The variable `$USER` conventionally holds your username (although this isn't guaranteed); you can also run `whoami` to see who is logged in.

Every user may belong to one or more "groups", which you can see by running `groups`.

For example, I'm in the groups:

```
 % groups
docker uucp audio wheel akshay
```

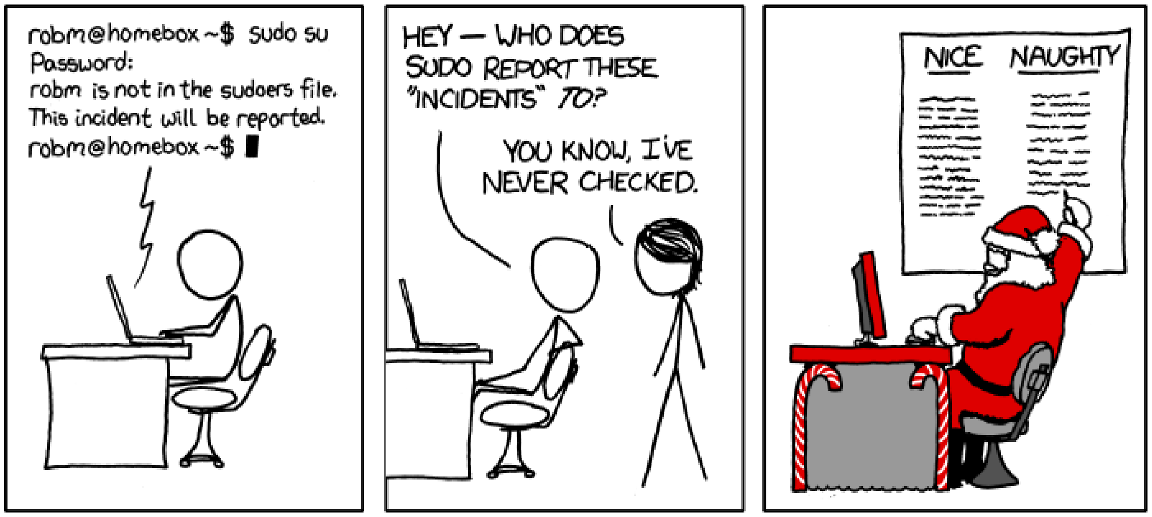
Each of these groups gives me special permissions and abilities. For example:

- The group "docker" means I can run the program "Docker" (which usually only root can run) without using `sudo`.
- The group "uucp" means I can read and write to serial ports (the name is a holdover from the old "unix-to-unix copy" program).
- The group "audio" means I can use my computer's speakers (you wouldn't have these permissions on a remote server like `myth`, for example).
- The group "wheel" gives me permission to use the program `sudo`. Trying to use `sudo` without this permission will cause an error.
- The group "akshay" gives me permission to... access my own files? This is kind of useless nowadays (I can *always* access my own files, since I own them), but it used to be used as a way to share files among different users on the same computer.

Historically, groups were used to share files among different users on a multi-user system (like *myth*). For example, Professor Engler's research group might have had a corresponding UNIX group named `engler_group`, so we could all collaborate on files and source code. Nowadays this has been replaced with other means of collaboration like `git`, and everyone has their own laptop, so this use has become more rare.

### Permissions

On UNIX, you must have the appropriate "permissions" to do certain actions.


```
robm@homebox ~$ sudo su
Password:
robm is not in the sudoers file.
This incident will be reported.
robm@homebox ~$ █
```
HEY — WHO DOES SUDO REPORT THESE "INCIDENTS" TO?
YOU KNOW, I'VE NEVER CHECKED.
NICE
NAUGHTY

Source: xkcd 838

One of the most important types of permissions (and probably the only one you'll see frequently) is permissions on specific files.

### File Permissions

Every file has an "owner" and a "group".

Every file has three sets of permissions: owner permissions, group permissions, and everyone else permissions.

`r` means "permission to read", `w` means "permission to write", and `x` means "permission to execute (i.e., run)".

You can see file permissions by running `ls -l`.

### File Permissions Example

*Output of `ls`*

```
-rwxr-xr-x 1 root root      153736 Sep  4 07:33 grep
```

These are the permissions on my `/usr/bin/grep` binary, as given by `ls -l`.

### File Permissions Example

*Owner*

```
-rwxr-xr-x 1 root root      153736 Sep  4 07:33 grep
```

The owner (root) can read, write, and execute `/usr/bin/grep`.

### File Permissions Example

*Group*

```
-rwxr-xr-x 1 root root      153736 Sep  4 07:33 grep
```

The members of the group "root" can read and execute `/usr/bin/grep`, but **not** write to it.

### File Permissions Example

*Everyone*

```
-rwxr-xr-x 1 root root      153736 Sep  4 07:33 grep
```

Everyone else can read and execute `/usr/bin/grep`, but **not** write to it.

### Changing Permissions

*Owner*

We can change the owner or group of a file using the `chown` and `chgrp` commands.

*Example* 2.1 (chown). Changing the owner of a file `hello.txt` to the user `akshay`:

```
chown akshay hello.txt
```

### Changing Permissions

*Group*

We can change the owner or group of a file using the `chown` and `chgrp` commands.

*Example* 2.2 (chgrp). Changing the group of a file `hello.txt` to the group `staff`:

```
chgrp staff hello.txt
```

### Changing Permissions

We can change the permissions on a file using the `chmod` command (CHANGE FILE MODE).

A file's MODE is the combination of its read, write, and execute permissions, for example `rw` for readable/writable or `rx` for readable/executable. See the `chmod` man page for more info!

We've already seen this!

*Example* 2.3. <only@2>[chmod +x] Make a shell script executable:

```
chmod +x my_script.sh
```

*Example* 2.4. <only@3>[chmod -w] Make a file read-only.

```
chmod -w my_safe_file.txt
```

*Example* 2.5. <only@4>[chmod -r] Make a file non-readable:

```
chmod -r my_secret.txt
```

By default, `chmod` changes the permissions for everyone at once. You can also specifically change one of the three sets of permissions:

```
chmod u+x my_script.sh
chmod g+rw group_plan.txt
chmod o-r my_secret.txt
chmod 777 open_permissions.txt
```

In the last example, I set the permission bits of the file directly. They are represented as an octal number; each of the digits represents three bits (read, write, and execute respectively). There are three digits for the three kinds of permissions: owner, group, and everyone. For example, a file that's 644 is readable and writable by the owner, but only readable by everyone else.

Note that, if you are the owner of a file, you can easily override the permissions. These permissions are mostly to protect your files from *others*, not from yourself.

#### Types of File

There are a few types of files, with different properties. You can tell them apart by the first character in the output of `ls -l`.

```
lrwxrwxrwx 1 root root       21 Oct  8 16:05 os-release -> ../usr/lib/os-release
drwxr-xr-x 1 root root       18 Oct  8 16:15 ostree
-rw-r--r-- 1 root root       79 Nov 29 02:14 ostree-mkinitcpio.conf
```

This is from my `/etc` directory, which is where programs store their configuration files.

#### Types of File

**-** A regular file.

**b** A block device (like a hard disk).

**c** A character device (like a serial port).

**d** A directory.

**l** A symbolic link.

**n** A network file.

**p** A "named pipe".

**s** A "named socket".

Of these, the ones you'll see most are regular files, directories, and symbolic links. We haven't seen symbolic links before, so let's look at them in a little more detail.

## 2.3 Shortcuts

#### Symbolic Links

**Definition 2.6** (symlink)**.** A SYMBOLIC LINK (or "symlink") is a shortcut to a file or directory.

You can create one with the `ln -s` command, as follows:

```
ln -s $target $link_name
```

When you try to read from a symlink, you actually read from the file it's pointing to. The `readlink` command tells you where a symlink points.

They're useful in cases where you want the same file to exist in multiple places, but you don't want to have a bunch of copies (which can get out of sync if someone edits one but not the others). For example, I use them to have copies of files in both my "Dropbox" and "Documents" folders.

Permissions are *shared* between a symlink and the target file. Trying to change the permissions on the link will change the permissions on the file itself.

In addition to shortcuts to files, we can also define shortcuts to commands. These are called aliases.

### Aliases

**Definition 2.7** (alias)**.** A ALIAS is like a shortcut for a specific command.

You can create one with the `alias` command, as follows:

```
alias hi="echo 'hello'"
```

Running an alias will run the command it points to. You can see what an alias named "hi" does by running `alias hi`.

Just like environment variables, aliases only last until you exit the shell.

### Aside: Searching for Files

The FIND tool (which has the unusually logical name `find`) is a powerful way to search for files. We're not going to go into too much detail, because it's ultimately not all that important, but here are a few examples:

*Example* 2.8. <only@2>[find -name] Find all files named "hello":

```
find . -name "hello"
```

*Example* 2.9. <only@3>[find -executable] Find all files marked "executable":

```
find . -executable
```

*Example* 2.10. <only@4>[find -type] Find all regular files, directories, and links:

```
find . -type f,d,l
```

*Example* 2.11. <only@5>[find] Find all regular files (but not links) which are marked executable and named "hello".

```
find . -type f -name "hello" -executable
```

*Example* 2.12. <only@3>[find -type] Find all regular files, directories, and links:

```
find . -type f,d,l
```

If you want to learn more about `find`, look at the man page! There are a lot of examples in there.

## 3 Shell Configuration

Just like you can configure `vim` using `.vimrc`, you can configure your shell using a special "run commands" file.

### Configuring your Shell

If you're using `bash`, your shell configuration file is called `~/.bashrc`. If you're using `zsh`, it's called `~/.zshrc`.

This file is a shell script that's run every time your shell starts. You can use it to define aliases and environment variables.

For example, my `.bashrc` includes the lines:

```
alias ls='ls --color=auto'
PS1='[\u@\h \W]\$ '
export EDITOR=vim
export PATH=$PATH:~/bin
```

The first line defines an alias for the command `ls`, which automatically runs it in "color" mode.

The second line configures my prompt, using the prompt variable `$PS1`.[1]

The third line sets my default editor to be `vim`. A lot of programs use this, so set it to your favorite terminal editor! You can also set it to VSCode using `export EDITOR='code --wait'`.

The fourth line adds the directory `~/bin` to my `$PATH`. This means any executable files within `~/bin` (for example, a shell script) can be run as ordinary programs from anywhere.

Note that `$PS1` is a shell variable (not exported), while `$EDITOR` and `$PATH` are environment variables (exported).

The shell (and most CLI programs) are incredibly customizable, far beyond what even a whole quarter could cover. If you're running a program and you wish it did something slightly different by default, there's probably a way to make that happen.

## 4 Multitasking

We know how to run one command at a time right now, but if we want to run two commands at once we have to open a new terminal. Honestly, nowadays opening a new terminal window isn't too bad, but in the olden days when a "terminal" was a piece of hardware, it wasn't super feasible.

### 4.1 Job Control

#### Jobs

**Definition 4.1** (job)**.** A JOB is a task you're doing in the terminal, usually corresponding to a program that you're running. You can have one FOREGROUND JOB and many BACKGROUND JOBS running at the same time. You can also have many SUSPENDED JOBS which are frozen (i.e., not running).

Whenever we run a program from the shell, we're starting a new foreground job. Jobs are tied to their "controlling terminal", and will exit when the terminal window is closed.

#### Suspending Jobs

You can "suspend" a job (put it to sleep) by pressing CONTROL-Z on your keyboard. Try it from `vim`!

You can see all the jobs in your current terminal and their statuses by running `jobs`.

[1] If you want to learn how to customize your own prompt, try `man bash` or `man zshmisc`.

### Background Jobs

You can "background" a suspended job (wake it up, but hide it) by running `bg`.

If you try to background a program like `vim`, it'll immediately suspend itself again because it needs to be connected to a terminal. However, if you have a long-running command like a download, you can background it without any issues.

If you have multiple jobs suspended, `bg` will run the most recent one. You can specify a different one using the job number from `jobs`:

```
bg %1
```

### Background Jobs

You can also run a new job in the background by adding an ampersand (`&`) to the end of the command:

```
sleep 5 &
```

You can also do this to a set of commands:

```
(sleep 5 && printf "\a") &
```

### Foregrounding Jobs

You can "foreground" a suspended or background job (wake it up and let it take over the terminal) by running `fg`.

If you have multiple suspended or background jobs, `fg` will run the most recent one. You can specify a different one using the job number from `jobs`:

```
fg %1
```

### Quitting Jobs

Usually, you can "kill" a foreground job (quit it) by pressing CONTROL-C on your keyboard.

You can "kill" a suspended or background job (wake it up and let it take over the terminal) by running `kill`. You must specify a job number from `jobs`:

```
kill %1
```

Note that it may take some time for the program to exit, and this may not work on certain programs like `vim`.

### Force-quitting Jobs

The `kill` command works by sending the process (program) the `SIGTERM` signal (which politely asks it to exit).

`SIGTERM` is a bit like pressing the "close" button on a window. Most of the time it'll work, but sometimes the program will just not respond.

Some processes (programs) may ignore `SIGTERM`. In this case, you can use `SIGKILL` to force-quit it.

```
kill -s KILL %1
```

Or, equivalently:

```
kill -9 %1
```

Sending `SIGKILL` is the equivalent of ending a program from Task Manager (on Windows) or force-quitting a program (on macOS). Any unsaved work will be lost, and the program may exit in an invalid state (which might cause error messages the next time you open it). Don't do this unless you absolutely have to.

You might also want to know about the `killall` and `pkill` commands, which are similar to `kill` but take the *name* of a process rather than the process's ID.

## 4.2 Multiplexing

### Splitting the Terminal

Sometimes we want to have multiple terminal programs open *at the same time*. In other words, we want to "split" our terminal window.

You might be wondering why you wouldn't just open a second terminal window. If you're working locally, that's actually a great option! It'll integrate with the rest of your non-terminal programs far better than anything else. However, a lot of the time you'll be working on a remote shell over `ssh`; in this case, opening a new terminal window is a hassle.

Job control will only let us open one program in the foreground at a time.

Unfortunately, there is *no built-in way* to have multiple programs open at the same time.

Fortunately, the shell is almost 60 years old, and other people have solved this problem for us.

### Terminal Multiplexers

A TERMINAL MULTIPLEXER is a program which splits one "real" terminal (i.e., one window) into many "virtual" terminals.

There are a few terminal multiplexers around:

- `screen` is old but installed on most computers
- `tmux` is new but needs to be installed manually

For this class, we'll be talking about `tmux`!

### Prefix Keys

We need some way to "talk" to `tmux` to give it commands.

But we also want to talk to the program running inside `tmux` so we can use it!

`tmux` solves this problem using a PREFIX KEY; any time you want to talk to `tmux`, you start by pressing CONTROL-B.

If you want to send a CONTROL-B to a program *inside* tmux, press CONTROL-B twice in a row.

### Using `tmux`

If you run `tmux`, you're given a shell prompt with a status bar at the bottom.

There's a bunch of keyboard shortcuts to do various things in `tmux`. Remember to press CONTROL-B before using any of them!

**Splitting the screen (vertically):** `%`

**Splitting the screen (horizontally):** `"`

**Going to the next "pane":** `o`

**Going to a specific pane:** `q <number>`

**Close the current pane** `x`

Check out `https://tmuxcheatsheet.com/` or `https://quickref.me/tmux` for more!

### Advanced `tmux`

`tmux` has another use; you can "detach" from your virtual terminal and reattach to it from another terminal window.

To detach: CONTROL-B `d`

To attach: `tmux attach`

### Why `tmux`?

Where `tmux` really shines is when used with `ssh`.

- You only need to enter your `ssh` password once.
- If your Wi-Fi drops and you lose your `ssh` connection, your programs keep running.
- You can detach a `tmux` session containing a long-running job and come back to check on it later.