CS 45, Lecture 6
Command Line Environment

Winter 2023

Akshay Srivatsan, Ayelet Drazen, Jonathan Kula

Outline

1. Review
2. The Environment
3. Shell Configuration
4. Multitasking

Announcements

- Assignment 2 is due today; let us know if you need an extension (beyond the usual late days).
- Assignment 3 will be coming out later today.

Outline

1. Review
2. The Environment
3. Shell Configuration
4. Multitasking

Text Editors

In the previous lecture, we saw:

- How to edit files in the terminal

Text Editors

In the previous lecture, we saw:

- How to edit files in the terminal
- How to enter/exit a full screen program

Text Editors

In the previous lecture, we saw:

- How to edit files in the terminal
- How to enter/exit a full screen program

In this lecture, we will see:

- How to configure and customize your shell

Text Editors

In the previous lecture, we saw:

- How to edit files in the terminal
- How to enter/exit a full screen program

In this lecture, we will see:

- How to configure and customize your shell
- How to multitask in the terminal

Text Editors

In the previous lecture, we saw:

- How to edit files in the terminal
- How to enter/exit a full screen program

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

Terminal vs. Shell vs. Command Line

Definition (terminal)

The TERMINAL is the window you open. Think of it like a web browser.

Terminal vs. Shell vs. Command Line

Definition (terminal)

The TERMINAL is the window you open. Think of it like a web browser.

Definition (shell)

The SHELL is the program you use to launch other programs. Think of it like Google.

Terminal vs. Shell vs. Command Line

Definition (terminal)

The TERMINAL is the window you open. Think of it like a web browser.

Definition (shell)

The SHELL is the program you use to launch other programs. Think of it like Google.

Definition (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.

Outline

1. Review

2. The Environment

2.1 Configuration

2.2 Permissions

2.3 Shortcuts

3. Shell Configuration

4. Multitasking

Contents of the Environment

The “environment” a program runs in includes several things:

Contents of the Environment

The “environment” a program runs in includes several things:

- The user who's running it

Contents of the Environment

The “environment” a program runs in includes several things:

- The user who's running it
- The files on the filesystem

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)

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`), which we've already seen

Outline

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.

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`.

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`.

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

Environment Variables

ENVIRONMENT VARIABLES are a way to configure a program's default behavior.

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.

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.

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” an environment variable as follows:

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

Setting Environment Variables

You can “export” an environment variable as follows:

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

You can temporarily set an environment variable as follows:

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

Outline

Users and Groups

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

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` holds your username (although this is just a convention, there's no actual requirement that the contents match).

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` holds your username (although this is just a convention, there's no actual requirement that the contents match).

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

For example, I'm in the groups:

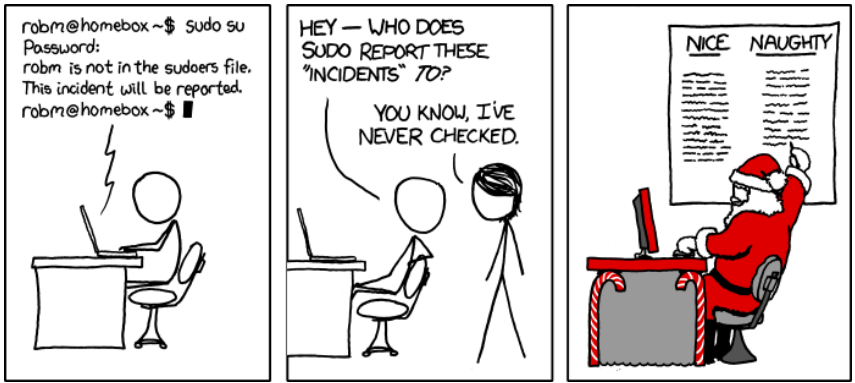
```
 % groups
docker uucp audio wheel akshay
```

Permissions

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

Permissions

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



Source: xkcd 838

File Permissions

Every file has an “owner” and a “group”.

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.

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)”.

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 (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 (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).

We've already seen this!

Changing Permissions

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

We've already seen this!

Example (chmod +x)

Make a shell script executable:

```
chmod +x myscript.sh
```

Changing Permissions

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

Example (chmod -w)

Make a file read-only.

```
chmod -w mysafefile.txt
```

Changing Permissions

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

Example (chmod -r)

Make a file non-readable:

```
chmod -r mysecret.txt
```

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”.

Outline

Symbolic Links

Definition (symlink)

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

Symbolic Links

Definition (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
```

Symbolic Links

Definition (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.

Aliases

Definition (alias)

A ALIAS is like a shortcut for a specific command.

Aliases

Definition (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'"
```

Aliases

Definition (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`.

Aside: Searching for Files

The FIND tool is a powerful way to search for files.

Aside: Searching for Files

The FIND tool is a powerful way to search for files.

Example (find -name)

Find all files named "hello":

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

Aside: Searching for Files

The FIND tool is a powerful way to search for files.

Example (find -executable)

Find all files marked "executable":

```
find . -executable
```

Aside: Searching for Files

The FIND tool is a powerful way to search for files.

Example (find -type)

Find all regular files, directories, and links:

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

Aside: Searching for Files

The FIND tool is a powerful way to search for files.

Example (find)

Find all regular files (but not links) which are marked executable and named "hello".

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

Outline

1. Review
2. The Environment
3. Shell Configuration
4. Multitasking

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
```

Outline

1. Review

2. The Environment

3. Shell Configuration

4. Multitasking

4.1 Job Control

4.2 Multiplexing

Outline

1. Review

2. The Environment

3. Shell Configuration

4. Multitasking

4.1 Job Control

4.2 Multiplexing

Jobs

Definition (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).

Jobs

Definition (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`!

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`.

Background Jobs

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

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 &
```

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`.

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.

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
```

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 program the `SIGTERM` signal (which politely asks it to exit).

Force-quitting Jobs

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

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

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

Or, equivalently:

```
kill -9 %1
```

Outline

1. Review

2. The Environment

3. Shell Configuration

4. Multitasking

4.1 Job Control

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.

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.

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

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.

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.

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.

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.

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:

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

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

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.

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!

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.

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.