1. Review

2. The Environment

3. Shell Configuration

4. Multitasking
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

2. The Environment

3. Shell Configuration

4. Multitasking
Recap

In the previous lecture, we saw:

- How to edit files in the terminal
Recap

In the previous lecture, we saw:

- How to edit files in the terminal
- How to enter/exit a full screen program (vim)
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
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
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
Definition (terminal)
The **TERMINAL** is the window you open. Think of it like a web browser.
## Terminal vs. Shell vs. Command Line

<table>
<thead>
<tr>
<th>Definition (terminal)</th>
<th>Definition (shell)</th>
</tr>
</thead>
<tbody>
<tr>
<td>The <strong>TERMINAL</strong> is the window you open. Think of it like a web browser.</td>
<td>The <strong>SHELL</strong> is the program you use to launch other programs. Think of it like Google.</td>
</tr>
<tr>
<td>Definition (terminal)</td>
<td>The <strong>TERMINAL</strong> is the window you open. Think of it like a web browser.</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------------------------------------------------------------------</td>
</tr>
<tr>
<td>Definition (shell)</td>
<td>The <strong>SHELL</strong> is the program you use to launch other programs. Think of it like Google.</td>
</tr>
<tr>
<td>Definition (cli)</td>
<td>A <strong>COMMAND LINE INTERFACE (CLI)</strong> is a generic term for a text-based program which runs within a terminal. Think of this like “the web”. A <strong>CLI PROGRAM</strong> or a <strong>TUI PROGRAM</strong> is like a website.</td>
</tr>
</tbody>
</table>
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
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)
Outline

1. Review

2. The Environment
   2.1 Configuration
   2.2 Permissions
   2.3 Shortcuts

3. Shell Configuration

4. Multitasking
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.
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`. 
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.

- $PATH: Determines where programs can be located. If a program isn't found "on your $PATH", you'll get a "command not found" error.
- $TERM: Which terminal you're using.
- $USER: Your username.
- $EDITOR: Which editor you prefer.
- $PWD: Your current directory.
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
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:

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

You can temporarily set an environment variable as follows:

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

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

2. The Environment
   2.1 Configuration
   2.2 Permissions
   2.3 Shortcuts

3. Shell Configuration

4. Multitasking
We also talked about this a bit in Lecture 2, but every command you run runs as a specific user.
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.
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
```
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”.
Every file has an “owner” and a “group”.

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

<table>
<thead>
<tr>
<th>Mode</th>
<th>Owner</th>
<th>Group</th>
<th>Others</th>
<th>Date</th>
<th>Time</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>-rwxr-xr-x</td>
<td>root</td>
<td>root</td>
<td></td>
<td>153736</td>
<td>Sep 4 07:33</td>
<td>grep</td>
</tr>
</tbody>
</table>

The owner (root) can read, write, and execute /usr/bin/grep.
## File Permissions Example

### Group

```
-rwrxr-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**

```
-rwrxr-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`:

```bash
chown akshay hello.txt
```
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:

```bash
chmod +x my_script.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 my_safe_file.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 my_secret.txt`
We can change the permissions on a file using the `chmod` command (CHANGE FILE MODE).

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

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

1. Review

2. The Environment
   2.1 Configuration
   2.2 Permissions
   2.3 Shortcuts

3. Shell Configuration

4. Multitasking
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.
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.

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.
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`.
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:

```bash
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”:

```bash
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
```
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
1. Review

2. The Environment

3. Shell Configuration

4. Multitasking
   4.1 Job Control
   4.2 Multiplexing
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).
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.
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
```
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
```
You can also run a new job in the background by adding an ampersand (&) to the end of the command:

```bash
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) &
```
Foregrounderg Jobs

You can “foreground” a suspended or background job (wake it up and let it take over the terminal) by running `fg`.
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
```
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`.
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`. 
The `kill` command works by sending the process (program) the SIGTERM signal (which politely asks it to exit).
Force-quitting Jobs

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

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
```
1. Review

2. The Environment

3. Shell Configuration

4. Multitasking
   4.1 Job Control
   4.2 Multiplexing
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:
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`!
We need some way to “talk” to \texttt{tmux} to give it commands.
Prefix Keys

We need some way to “talk” to \texttt{tmux} to give it commands.

But we also want to talk to the program running inside \texttt{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!
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.