

Introduction to Bash

Hawlader Al-Mamun

2024-07-12

Table of contents

Installing VS Code and FileZilla	5
First Time Accessing the Server	9
Accessing the Server	9
Using the SSH Command	9
1 Introducing the Shell	11
1.1 Accessing the Server	11
1.1.1 Using the SSH Command	11
1.1.2 What is a Shell?	12
1.1.3 Why Use the Shell?	12
1.1.4 The Shell Prompt	13
1.2 Understanding Directories	13
1.2.1 Home Directory	13
1.2.2 Scratch Directory	13
1.2.3 Working Directory	13
1.2.4 Navigating Directories	14
1.2.5 Listing the files in the current directory	14
1.3 Summary	14
2 Navigating Files and Directories	16
2.1 The File System	16
2.1.1 The Root Directory	16
2.1.2 The Home Directory	16
2.1.3 The Uses of /	16
2.1.4 Examples	17
2.2 Commands, Arguments, and Flags	17
2.2.1 Example with <code>ls</code> and the <code>-F</code> Flag	17
2.2.2 Other Useful Flags for <code>ls</code>	18
2.3 Clearing Your Terminal	18
2.3.1 Using the <code>clear</code> Command	19
2.3.2 Navigating Command History	19
2.4 Using Tab for Auto-Complete	19
2.5 Getting Help	20
2.5.1 Using <code>--help</code>	20

2.5.2	Using man	20
2.5.3	Reading the Output	20
2.6	Special Folders: . and	21
2.6.1	The . (Dot) Directory	21
2.6.2	The .. (Dot Dot) Directory	21
2.6.3	Nested .. Directories	22
2.7	Absolute Path and Relative Path	22
2.7.1	Absolute Path	22
2.7.2	Relative Path	22
2.7.3	Example Paths	23
2.7.4	Summary	23
3	Working With Files and Directories	25
3.1	Creating directories	25
3.1.1	Step one: see where we are and what we already have	25
3.1.2	Create a directory	25
3.2	Good names for files and directories	26
3.3	Create a text file	27
3.4	Viewing Files in the Terminal	27
3.4.1	cat	27
3.4.2	head	28
3.4.3	tail	28
3.4.4	less	28
3.4.5	more	29
3.4.6	Difference Between cat and less/more	29
3.5	Moving files and directories	29
3.6	Copying files and directories	30
3.7	Removing files and directories	31
3.8	Using rm Safely	31
3.9	Using Wildcards for Accessing Multiple Files at Once	32
3.9.1	The Asterisk (*)	32
3.9.2	The Question Mark (?)	33
3.9.3	Combining Wildcards	33
3.9.4	Handling Non-Matching Wildcards	33
3.10	Summary	33
4	Pipes and Filters	35
4.1	wc ‘word count’ command	35
4.2	Capturing output from commands	36
4.3	Filtering output	37
4.3.1	What Does sort -n Do?	37
4.4	The >> operator	38
4.4.1	Using >> in Bash Commands	38

4.4.2	Syntax	38
4.4.3	Example	38
4.4.4	Multiple Appends	39
4.5	Passing output to another command	39
4.6	Combining multiple commands	39
4.7	Summary	40

Installing VS Code and FileZilla

Welcome to the Introduction to BASH class! In this course, we will be exploring the basics of the Bash shell, a powerful tool for interacting with your computer's operating system. Bash is widely used in bioinformatics for tasks ranging from data processing to system administration. To enhance our learning experience and prepare us for using different bioinformatics software and accessing high-performance computing (HPC) facilities, we will use two key tools: VS Code and FileZilla.

Why Use VS Code and FileZilla?

VS Code (Visual Studio Code):

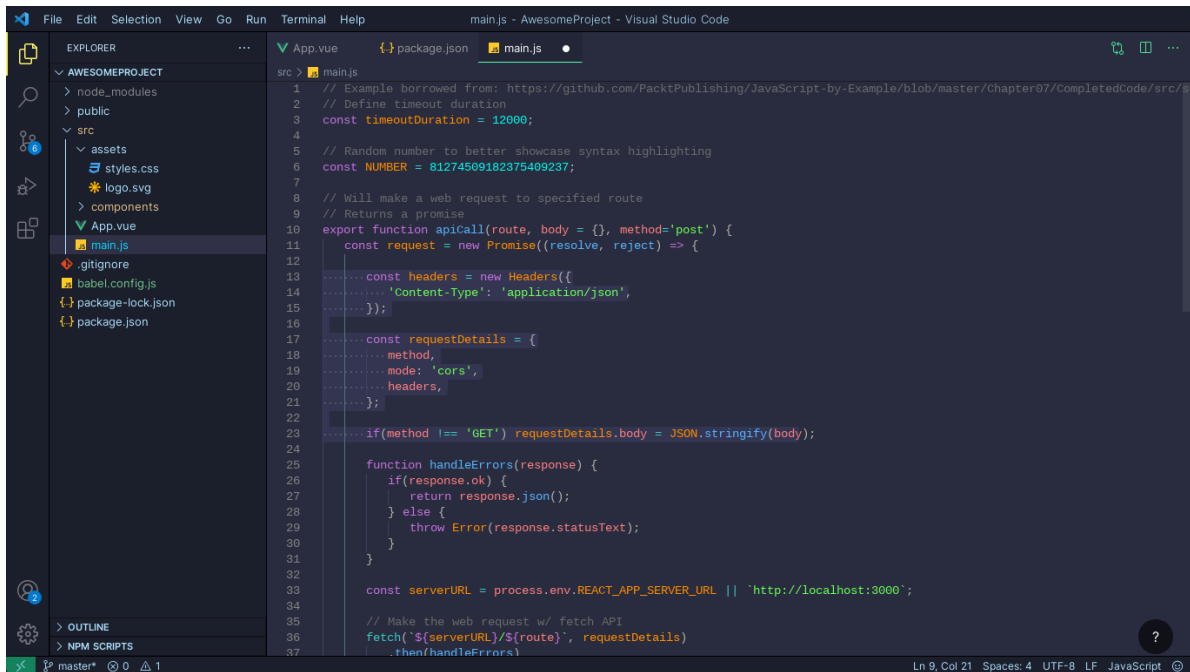
- **Integrated Terminal:** VS Code comes with an integrated terminal that supports Bash, making it a convenient all-in-one tool for writing and executing shell commands.
- **Cross-Platform Compatibility:** It works on Windows, macOS, and Linux, ensuring all students can use the same tool regardless of their operating system.
- **Text Editor:** VS Code also includes a powerful text editor that supports syntax highlighting and other features that make writing scripts easier and more efficient.

FileZilla:

- **File Transfer Protocol:** FileZilla is a free, open-source FTP client that allows you to transfer files between your local machine and a remote server. This is particularly useful for managing files and scripts in a remote environment.
- **Ease of Use:** It has an intuitive graphical interface that makes it easy to use, even for beginners.

Installation Instructions

VS Code:



1. Windows:

- Download the [Visual Studio Code installer](#) for Windows.
- Once it is downloaded, run the installer (VSCodeUserSetup-{version}.exe). This will only take a minute.
- By default, VS Code is installed under C:\Users\Username\AppData\Local\Programs\Microsoft VS Code

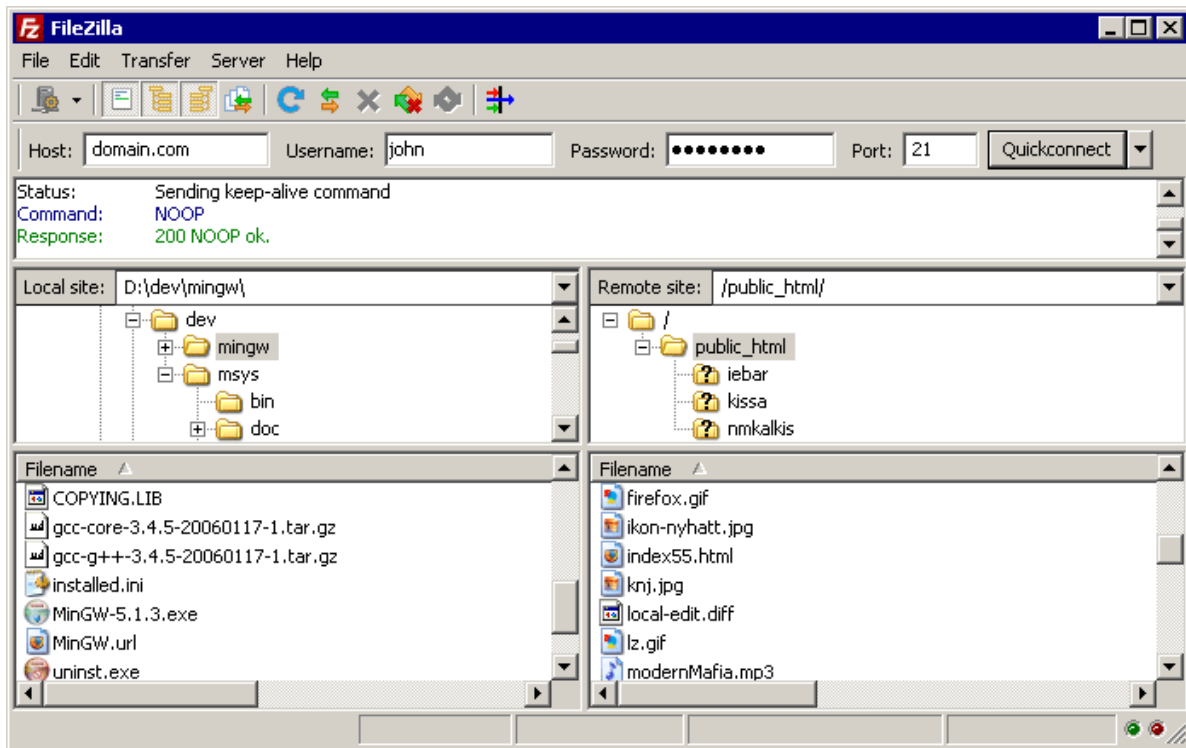
2. macOS:

- Download [Visual Studio Code](#) for macOS.
- Open the browser's download list and locate the downloaded app or archive.
- If archive, extract the archive contents. Use double-click for some browsers or select the 'magnifying glass' icon with Safari.
- Drag Visual Studio Code.app to the Applications folder, making it available in the macOS Launchpad.
- Open VS Code from the Applications folder, by double clicking the icon.
- Add VS Code to your Dock by right-clicking on the icon, located in the Dock, to bring up the context menu and choosing Options, Keep in Dock.

3. Linux:

- Visit the [VS Code download page](#).
- Follow the instructions specific to your distribution (e.g., Ubuntu, Fedora, etc.).

FileZilla:



1. Windows:

- Visit the [FileZilla download page](#).
- Click on the “Download FileZilla Client” button.
- Run the installer and follow the prompts to complete the installation.

2. macOS:

- Visit the [FileZilla download page](#).
- Click on the “Download FileZilla Client” button.
- Open the downloaded file and drag the FileZilla icon to the Applications folder.

3. Linux:

- You can install FileZilla via your package manager. For example, on Ubuntu, you can use the command:

```
sudo apt-get install filezilla
```

Preparation for Class

Please ensure you have both VS Code and FileZilla installed on your computer before attending the class. This will allow us to dive straight into the practical aspects of learning Bash without any delays.

We look forward to helping you develop your skills in Bash scripting and system administration. If you encounter any issues during installation, please do not hesitate to reach out for assistance.

First Time Accessing the Server

In this chapter, we will guide you through your first time accessing our server. You will learn how to use your assigned user ID and password to connect the server from your computer. Let's get started!

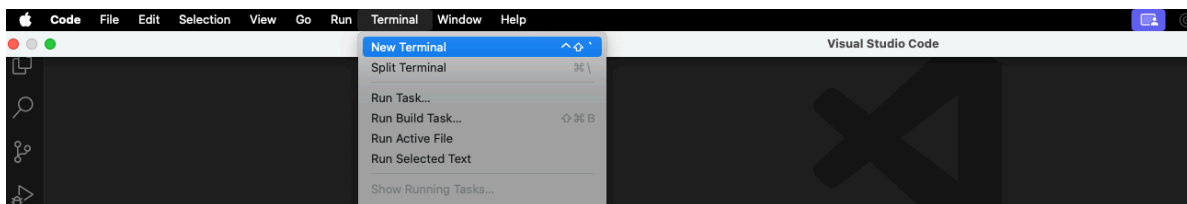
Accessing the Server

Each student will be provided with a unique user ID (e.g., s-1, s-2, s-3) and the server's IP address. To access the server, you will use the `ssh` (Secure Shell) command. SSH allows you to securely connect to the server and interact with it via the command line.

Using the SSH Command

1. Open VS Code Terminal:

- Launch Visual Studio Code.
- Open the integrated terminal by pressing `Ctrl + `` (backtick) or navigating to **Terminal > New Terminal** or **View > Terminal** (depending on VS Code version) from the menu.



2. Connect to the Server:

- In the terminal, type the following command:

```
ssh your_user_id@server_ip
```

Replace `your_user_id` with your assigned user ID (e.g., s-1) and `server_ip` with the provided IP address of the server.

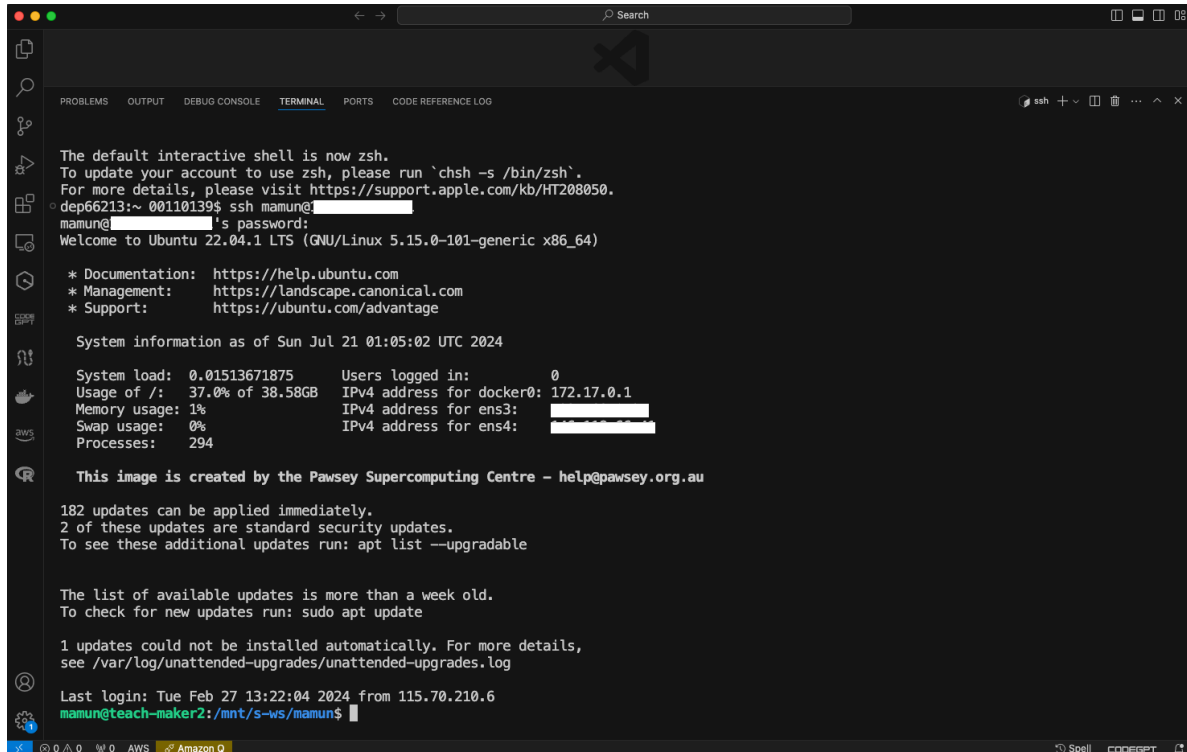
- Press Enter.

3. Enter Your Password:

- You will be prompted to enter your password. Note that in Linux, when you type your password, nothing will be displayed on the screen (no asterisks or dots) for security reasons. This is different from Windows, where you typically see asterisks.

4. Successful Connection:

- If your credentials are correct, you will be logged into the server and see a welcome message or command prompt.



```
The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/HT208050.
dep66213:~ 00110139$ ssh mamun@
mamun@:~$ 's password:
Welcome to Ubuntu 22.04.1 LTS (GNU/Linux 5.15.0-101-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

System information as of Sun Jul 21 01:05:02 UTC 2024

System load:  0.01513671875   Users logged in:      0
Usage of /:   37.0% of 38.58GB IPv4 address for docker0: 172.17.0.1
Memory usage: 1%             IPv4 address for ens3: 
Swap usage:  0%             IPv4 address for ens4: 
Processes:   294

This image is created by the Pawsey Supercomputing Centre - help@pawsey.org.au

182 updates can be applied immediately.
2 of these updates are standard security updates.
To see these additional updates run: apt list --upgradable

The list of available updates is more than a week old.
To check for new updates run: sudo apt update

1 updates could not be installed automatically. For more details,
see /var/log/unattended-upgrades/unattended-upgrades.log

Last login: Tue Feb 27 13:22:04 2024 from 115.70.210.6
mamun@teach-maker2:/mnt/s-ws/mamun$
```

1 Introducing the Shell

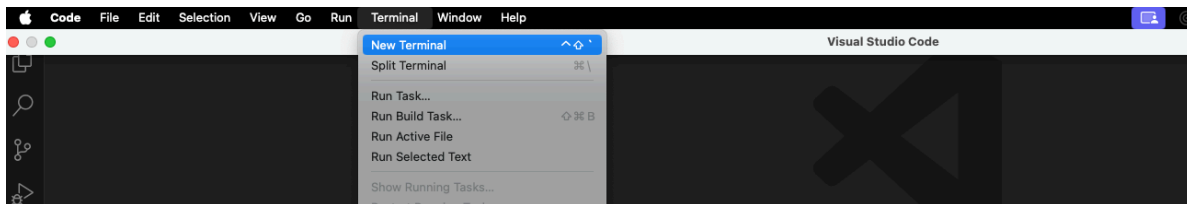
1.1 Accessing the Server

Each student will be provided with a unique user ID (e.g., s-1, s-2, s-3) and the server's IP address. To access the server, you will use the `ssh` (Secure Shell) command. SSH allows you to securely connect to the server and interact with it via the command line.

1.1.1 Using the SSH Command

1. Open VS Code Terminal:

- Launch Visual Studio Code.
- Open the integrated terminal by pressing `Ctrl + `` (backtick) or navigating to **Terminal** > **New Terminal** or **View** > **Terminal** (depending on VS Code version) from the menu.



2. Connect to the Server:

- In the terminal, type the following command:

```
ssh your_user_id@server_ip
```

Replace `your_user_id` with your assigned user ID (e.g., s-1) and `server_ip` with the provided IP address of the server.

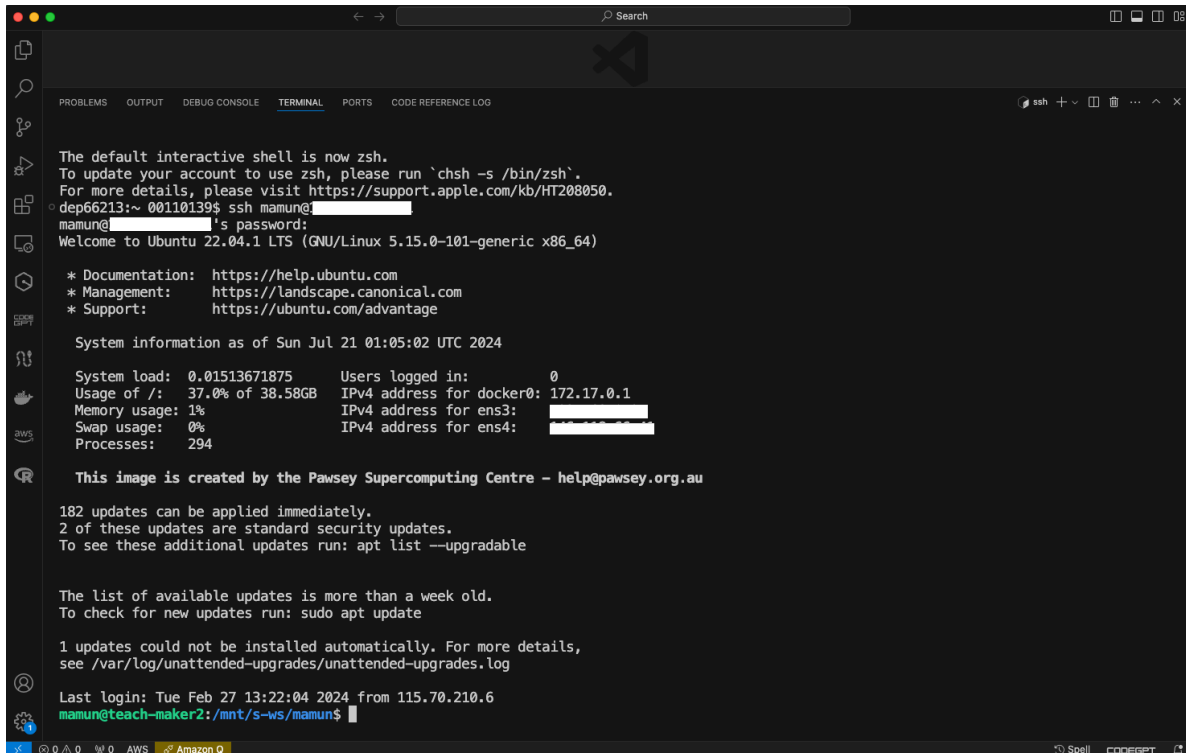
- Press Enter.

3. Enter Your Password:

- You will be prompted to enter your password. Note that in Linux, when you type your password, nothing will be displayed on the screen (no asterisks or dots) for security reasons. This is different from Windows, where you typically see asterisks.

4. Successful Connection:

- If your credentials are correct, you will be logged into the server and see a welcome message or command prompt.



```
The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
dep66213:~ 00110139$ ssh mamun@
mamun@:~$ 's password:
Welcome to Ubuntu 22.04.1 LTS (GNU/Linux 5.15.0-101-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

System information as of Sun Jul 21 01:05:02 UTC 2024

System load:  0.01513671875   Users logged in:      0
Usage of /:   37.0% of 38.58GB IPv4 address for docker0: 172.17.0.1
Memory usage: 1%             IPv4 address for ens3: 
Swap usage:  0%             IPv4 address for ens4: 
Processes:   294

This image is created by the Pawsey Supercomputing Centre - help@pawsey.org.au

182 updates can be applied immediately.
2 of these updates are standard security updates.
To see these additional updates run: apt list --upgradable

The list of available updates is more than a week old.
To check for new updates run: sudo apt update

1 updates could not be installed automatically. For more details,
see /var/log/unattended-upgrades/unattended-upgrades.log

Last login: Tue Feb 27 13:22:04 2024 from 115.70.210.6
mamun@teach-maker2: /mnt/s-ws/mamun$
```

1.1.2 What is a Shell?

A shell is a command-line interface (CLI) that allows users to interact with the operating system by typing commands. It acts as an intermediary between the user and the kernel, interpreting and executing commands entered by the user. The shell is a powerful tool for managing files, running programs, and automating tasks through scripts.

1.1.3 Why Use the Shell?

1. **Efficiency:** The shell allows users to perform complex tasks with simple commands, which can be more efficient than using a graphical user interface (GUI).
2. **Automation:** With the shell, users can write scripts to automate repetitive tasks, saving time and reducing errors.
3. **Flexibility:** The shell provides access to a wide range of tools and utilities, making it versatile for different types of tasks.

4. **Control:** Advanced users can have more control over the system and its processes compared to using a GUI.

1.1.4 The Shell Prompt

When you open a terminal window, you will see a shell prompt. This prompt indicates that the shell is ready to accept commands. The prompt usually looks something like this:

```
username@hostname:~$
```

Here's a breakdown of the components:

- **username:** Your user name on the system.
- **hostname:** The name of the computer or server.
- **~:** The current directory (in this case, ~ represents the home directory).
- **\$:** The prompt symbol for a standard user (it changes to # for the root user).

1.2 Understanding Directories

1.2.1 Home Directory

Upon logging in, you will be in your home directory, usually located at `/home/your_user_id`. This directory is your personal space on the server. However, the home directory is usually very small and should not be used for extensive work or large data storage.

1.2.2 Scratch Directory

The `/scratch` directory (or similar) is a designated area for temporary storage of large files and data processing. This space is typically much larger than your home directory and is intended for heavy computational tasks.

1.2.3 Working Directory

For our course, you will use `/mnt/s-ws/your_user_id` as your working directory. This directory provides ample space for your projects and assignments.

1.2.4 Navigating Directories

- **Who Am I? (whoami):**

- This command prints your current user ID.
- Usage:

```
whoami
```

- **Print Working Directory (pwd):**

- This command displays the full path of your current directory.
- Usage:

```
pwd
```

- **Change Directory (cd):**

- Use the `cd` command to navigate to your working directory.
- Example:

```
cd /mnt/s-ws/your_user_id
```

1.2.5 Listing the files in the current directory

- **ls will list of the contents of the current directory**

- Example:

```
ls
```

- **Command not found**

- If the shell can't find a program whose name is the command you typed, it will print an error message such as:

```
ks
```

1.3 Summary

By following these steps, you will be able to access the server and navigate the directories effectively. Here's a quick summary:

1. **Use SSH to connect to the server.**

2. Enter your password (nothing will be shown as you type).
3. Understand the purpose of the home directory and /scratch.
4. Use /mnt/s-ws/your_user_id as your working directory.
5. Learn basic commands (whoami, pwd, ls) and use Tab for auto-completion.

2 Navigating Files and Directories

2.1 The File System

The part of the operating system responsible for managing files and directories is called the **file system**. It organizes our data into files, which hold information, and directories (also called “folders”), which hold files or other directories.

2.1.1 The Root Directory

The root directory is the top-level directory in a file system. It is represented by a single forward slash (/). Every other file or directory in the file system is contained within the root directory, either directly or indirectly. Think of it as the starting point of the file system hierarchy.

2.1.2 The Home Directory

The home directory is a special directory designated for a specific user. It is where users have their personal space to store files and directories. Each user on the system has their own home directory. In most Linux systems, the home directory for a user named “username” would be `/home/username`. For the root user, the home directory is usually `/root`.

2.1.3 The Uses of /

The forward slash (/) serves two important purposes in the file system:

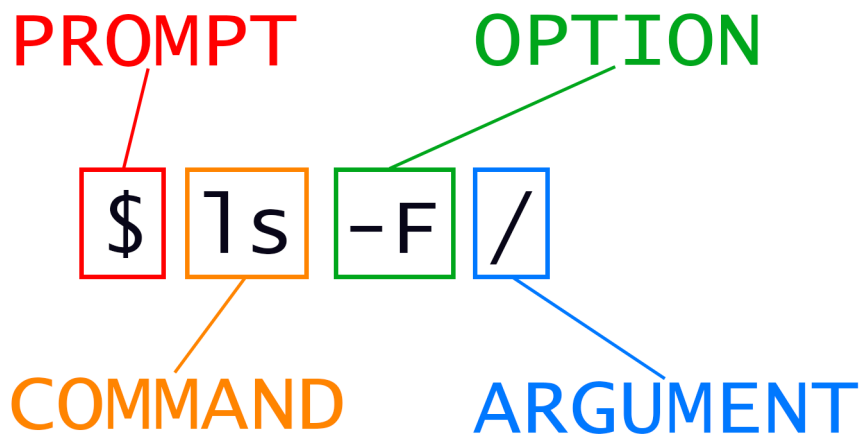
1. **Root Directory:** As mentioned, the single forward slash represents the root directory. It is the base of the file system hierarchy.
2. **Path Separator:** The forward slash is also used to separate directories and files in a path. For example, in the path `/home/username/Documents`, the slashes separate the directories `home`, `username`, and `Documents`.

2.1.4 Examples

1. **Root Directory:** The command `cd /` changes the current directory to the root directory.
2. **Home Directory:** The command `cd ~` or `cd /home/username` changes the current directory to the home directory of the user “username”.
3. **Path Separator:** In the path `/home/username/Documents/file.txt`, the slashes are used to navigate through the directories from the root to the file `file.txt`.

2.2 Commands, Arguments, and Flags

When working with the command line, you will often use **commands**. Commands are instructions you give to the computer to perform a specific task. A command can be followed by **arguments** (or parameters) and **flags** (or options) to modify its behavior.



- **Command:** The base instruction to perform a task. For example, `ls` is a command that lists directory contents.
- **Arguments:** Additional information you provide to the command. For example, in `ls /home/username`, `/home/username` is an argument specifying the directory to list.
- **Flags:** Options that modify the behavior of the command. Flags are usually preceded by a hyphen. For example, `ls -F` uses the `-F` flag to modify the output.

2.2.1 Example with `ls` and the `-F` Flag

The `ls` command lists the contents of a directory. By default, it shows the names of the files and directories. Adding the `-F` flag to `ls` appends a character to each file name to indicate

the type of file. For instance, a forward slash (/) is added to directories, an asterisk (*) to executable files, and an at sign (@) to symbolic links.

Example:

```
# ls -F /home/username  
ls -F /mnt/s-ws/everyone/
```

This command lists the contents of `/home/username` and appends type-indicating characters to each name.

2.2.2 Other Useful Flags for `ls`

- `-l`: Lists in long format, providing detailed information like permissions, number of links, owner, group, size, and timestamp.

```
# -l gives you more details on all files  
ls -l /mnt/s-ws/everyone/
```

- `-a`: Lists all files, including hidden files (those starting with a dot).

```
# -l gives you more details on all files  
ls -a /mnt/s-ws/everyone/
```

- `-h`: With `-l`, shows sizes in human-readable format (e.g., KB, MB).

```
ls -l -h /mnt/s-ws/everyone/  
  
# you can also merge flags  
ls -lh /mnt/s-ws/everyone/
```

- `-R`: Lists all subdirectories recursively.
- `-t`: Sorts by modification time, with the newest files first.
- `-r`: Reverses the order of the sort.
- `-d`: Lists directories themselves, not their contents.

2.3 Clearing Your Terminal

When working in the terminal, your screen can quickly become cluttered with output from various commands. To clear the terminal and create a fresh workspace, you can use the `clear` command.

2.3.1 Using the `clear` Command

The `clear` command clears all the previous output in the terminal, giving you a clean slate. Simply type `clear` and press Enter:

```
clear
```

This will remove all previous text from the screen, making it easier to focus on new commands and their output.

2.3.2 Navigating Command History

The terminal keeps a history of the commands you have entered. You can quickly navigate through your previous commands using the up and down arrow keys:

- **↑ Up Arrow:** Press the ↑ (up arrow) key to scroll through the commands you have previously entered, starting with the most recent.
- **↓ Down Arrow:** Press the ↓ (down arrow) key to scroll forward through the commands if you have gone back in your command history.

Using these keys can save you time, especially when you need to re-run or modify previous commands.

2.4 Using Tab for Auto-Complete

The tab key can significantly enhance your efficiency in the terminal by providing auto-completion for commands, file names, and directories. Here's how you can use it:

- **Auto-Complete Commands:** Start typing a command and press the tab key. If the command is unique, the terminal will auto-complete it. For example, typing `cle` and pressing tab will auto-complete to `clear` if no other commands start with `cle`.
- **Auto-Complete File and Directory Names:** When typing file or directory names, you can use the tab key to quickly complete the names. For instance, if you have a file named `document.txt` in the current directory, typing `doc` and pressing tab will auto-complete the name to `document.txt`.
- **List Possible Completions:** If there are multiple possible completions, pressing the tab key twice will list all the possible completions. For example, if you have files named `document1.txt`, `document2.txt`, and `document3.txt`, typing `doc` and pressing tab twice will list all three options.

Using the tab key for auto-completion can save you time and help avoid errors when typing long commands or file names.

2.5 Getting Help

When working in the terminal, you often need more information about commands and their options. Two essential tools for this are the `--help` option and the `man` (manual) command.

2.5.1 Using `--help`

Most commands in the terminal have a `--help` option that provides a brief overview of how to use the command, along with a list of available options and flags. For example, to get help on the `ls` command, you can type:

```
ls --help
```

This command will display a brief description of `ls` and its options. The output typically includes:

- A summary of what the command does.
- A list of available flags and options.
- Short descriptions of each flag and option.

2.5.2 Using `man`

The `man` command displays the manual page for a command, providing more detailed information than `--help`. To read the manual page for the `ls` command, you can type:

```
man ls
```

The manual page will include:

- **NAME:** The name of the command and a brief description.
- **SYNOPSIS:** The syntax for using the command.
- **DESCRIPTION:** A detailed description of what the command does.
- **OPTIONS:** A comprehensive list of all options and flags, with explanations.
- **EXAMPLES:** Examples of how to use the command.

2.5.3 Reading the Output

When you use `--help` or `man`, the output will appear in the terminal. Here's how to navigate and quit these outputs:

- **--help Output:** The `--help` output typically fits within a single screen. You can scroll through it using the scroll bar or your mouse.

- **man Output:** The `man` output is displayed in a pager (usually `less`), which allows you to scroll through the document.

2.5.3.1 Navigating man Output

- **Scroll Down:** Use the down arrow key or the `Page Down` key to scroll down.
- **Scroll Up:** Use the up arrow key or the `Page Up` key to scroll up.
- **Search:** Press `/` followed by a keyword to search within the manual page.
- **Quit:** Press `q` to quit and return to the terminal prompt.

2.6 Special Folders: `.` and `..`

In the file system, there are two special directory references: `.` (dot) and `..` (dot dot). These are used to represent the current directory and the parent directory, respectively. Understanding these references is crucial for navigating the file system efficiently.

2.6.1 The `.` (Dot) Directory

The `.` directory refers to the current directory. It is useful when you need to execute commands or scripts in the current directory or specify the current directory explicitly.

2.6.1.1 Example:

```
# List the contents of the current directory
ls .
```

2.6.2 The `..` (Dot Dot) Directory

The `..` directory refers to the parent directory, which is one level up in the file system hierarchy. This is helpful for moving up directories without specifying the full path.

2.6.2.1 Example:

```
# Move up one directory
cd ..
```

2.6.3 Nested .. Directories

You can chain multiple .. references to move up several levels in the directory hierarchy. Each .. moves you up one directory level.

2.6.3.1 Example:

```
# Move up two directories and then into a folder named folder_name
# cd ../../folder_name
cd ../../data
```

In this example: - The first .. moves up one level. - The second .. moves up another level. - After moving up two levels, the command then moves into the directory **data**.

2.7 Absolute Path and Relative Path

2.7.1 Absolute Path

An absolute path is a complete path from the root directory (/) to the desired file or directory. It always starts with a forward slash and provides the full location, ensuring that you can access the file or directory from anywhere in the file system.

2.7.1.1 Example:

If you have a directory named **FastQC** inside **/mnt/s-ws/everyone**, the absolute path to this directory would be:

```
/mnt/s-ws/everyone/FastQC
```

2.7.2 Relative Path

A relative path specifies a location relative to the current directory. It does not start with a forward slash and can use special directory references like . (current directory) and .. (parent directory) to navigate through the file system.

2.7.2.1 Example:

Assuming your current directory is `/mnt/s-ws/everyone`:

- To access the **FastQC** directory, you can use the relative path:

```
cd FastQC
```

- To go up one level and then into another directory (e.g., **Quiz**), you can use:

```
cd ../Quiz
```

2.7.3 Example Paths

Given the absolute path `/mnt/s-ws/everyone` and a folder **FastQC** inside it:

1. **Accessing FastQC using an absolute path:**

```
cd /mnt/s-ws/everyone/FastQC
```

This command changes the directory to **FastQC** from anywhere in the file system.

2. **Accessing FastQC using a relative path:**

```
cd FastQC
```

This command changes the directory to **FastQC** assuming you are already in `/mnt/s-ws/everyone`.

2.7.4 Summary

- The **file system** manages files and directories.
- The **root directory** (`/`) is the top-level directory.
- The **home directory** is a personal directory for each user.
- The **forward slash** (`/`) is used both as the root directory and as a separator in file paths.
- **Commands** are instructions, **arguments** provide additional information, and **flags** modify behavior.
- The **ls** command lists directory contents, and flags like `-F`, `-l`, `-a`, `-h`, and `-R` enhance its functionality.
- **pwd** prints the user's current working directory.
- **ls [path]** prints a listing of a specific file or directory; **ls** on its own lists the current working directory.
- **cd [path]** changes the current working directory.
- Most commands take options that begin with a single `-`.
- `/` on its own is the root directory of the whole file system.

- `.` on its own means ‘the current directory’; `..` means ‘the directory above the current one’.
- **Absolute Path:** Provides the full path from the root directory, ensuring access from anywhere in the file system. Example: `/mnt/s-ws/everyone/FastQC`.
- **Relative Path:** Specifies the location relative to the current directory. Example: FastQC from `/mnt/s-ws/everyone`.

3 Working With Files and Directories

3.1 Creating directories

We now know how to explore files and directories, but how do we create them in the first place?

3.1.1 Step one: see where we are and what we already have

```
pwd
# move to the directory in /mnt/s-ws/ designated for you
# cd /mnt/s-ws/{your id}
cd /mnt/s-ws/mamun
# view the current contents
ls -F
```

3.1.2 Create a directory

Let's create a new directory called thesis using the command `mkdir thesis` (which has no output):

```
mkdir thesis
```

As you might guess from its name, `mkdir` means 'make directory'. Since `thesis` is a relative path (i.e., does not have a leading slash, like `/what/ever/thesis`), the new directory is created in the current working directory:

```
ls -F
```

Since we've just created the `thesis` directory, there's nothing in it yet:

```
ls -F thesis
```

Note that `mkdir` is not limited to creating single directories one at a time. The `-p` option allows `mkdir` to create a directory with nested subdirectories in a single operation:

```
mkdir -p ../project/data ../project/results
```

The `-R` option to the `ls` command will list all nested subdirectories within a directory. Let's use `ls -FR` to recursively list the new directory hierarchy we just created in the project directory:

```
ls -FR ../project
```

3.2 Good names for files and directories

Complicated names of files and directories can make your life painful when working on the command line. Here we provide a few useful tips for the names of your files and directories.

Don't use spaces.

Spaces can make a name more meaningful, but since spaces are used to separate arguments on the command line it is better to avoid them in names of files and directories. You can use `-` or `_` instead (e.g. `north-pacific-gyre/` rather than `north pacific gyre/`). To test this out, try typing `mkdir north pacific gyre` and see what directory (or directories!) are made when you check with `ls -F`.

Don't begin the name with `-` (dash).

Commands treat names starting with `-` as options.

Stick with letters, numbers, `.` (period or 'full stop'), `-` (dash) and `_` (underscore).

Many other characters have special meanings on the command line. We will learn about some of these during this lesson. There are special characters that can cause your command to not work as expected and can even result in data loss.

If you need to refer to names of files or directories that have spaces or other special characters, you should surround the name in single quotes (`'`).

3.3 Create a text file

Let's change our working directory to thesis using `cd`, then run a text editor called Nano to create a file called `draft.txt`:

```
cd thesis
nano draft.txt
```

Let's type in a few lines of text.

```
GNU nano 2.0.6      File: draft.txt      Modified
It's not "publish or perish" any more,
it's "share and thrive".
█
```

```
^G Get Help    ^O WriteOut    ^R Read File   ^Y Prev Page   ^K Cut Text    ^C Cur Pos
^X Exit        ^J Justify     ^W Where Is    ^V Next Page   ^U UnCut Text  ^T To Spell
```

Once we're happy with our text, we can press `Ctrl+O` (press the `Ctrl` or `Control` key and, while holding it down, press the `O` key) to write our data to disk. We will be asked to provide a name for the file that will contain our text. Press `Return` to accept the suggested default of `draft.txt`.

Once our file is saved, we can use `Ctrl+X` to quit the editor and return to the shell.

3.4 Viewing Files in the Terminal

When working with files in the terminal, several commands can help you view their contents efficiently. Here, we will discuss `cat`, `head`, `tail`, `less`, and `more`, highlighting their uses and differences.

3.4.1 cat

The `cat` command (short for “concatenate”) is used to display the contents of a file. It is best suited for small files because it outputs the entire content to the terminal.

3.4.1.1 Example:

```
cat filename.txt
```

This command will display the entire content of `filename.txt`.

3.4.2 head

The `head` command displays the first few lines of a file. By default, it shows the first 10 lines, but you can specify a different number of lines with the `-n` option.

3.4.2.1 Example:

```
head filename.txt  
head -n 20 filename.txt
```

The first command shows the first 10 lines of `filename.txt`, while the second command shows the first 20 lines.

3.4.3 tail

The `tail` command displays the last few lines of a file. By default, it shows the last 10 lines, but you can specify a different number of lines with the `-n` option.

3.4.3.1 Example:

```
tail filename.txt  
tail -n 20 filename.txt
```

The first command shows the last 10 lines of `filename.txt`, while the second command shows the last 20 lines.

3.4.4 less

The `less` command is a pager that allows you to view the contents of a file one screen at a time. It is particularly useful for large files, as it does not load the entire file into memory at once.

3.4.4.1 Example:

```
less filename.txt
```

Use the arrow keys to scroll through the file. Press **q** to quit and return to the terminal.

3.4.5 more

The **more** command is another pager similar to **less**. It allows you to view the contents of a file one screen at a time. However, **more** is less powerful and flexible than **less**.

3.4.5.1 Example:

```
more filename.txt
```

Use the spacebar to scroll down one screen at a time. Press **q** to quit.

3.4.6 Difference Between cat and less/more

- **cat**: Displays the entire content of a file at once. It is useful for small files but can be overwhelming for large files since all content is outputted at once.
- **less/more**: Both are pagers that display one screen of content at a time, making them more suitable for large files. **less** is generally preferred over **more** because it provides more features, such as backward navigation and better performance.

3.5 Moving files and directories

In our thesis directory we have a file `draft.txt` which isn't a particularly informative name, so let's change the file's name using **mv**, which is short for 'move':

```
mv thesis/draft.txt thesis/quotes.txt
```

The first argument tells **mv** what we're 'moving', while the second is where it's to go. In this case, we're moving `thesis/draft.txt` to `thesis/quotes.txt`, which has the same effect as renaming the file. Sure enough, **ls** shows us that `thesis` now contains one file called `quotes.txt`:

```
ls thesis
```

One must be careful when specifying the target file name, since `mv` will **silently overwrite any existing file with the same name**, which could lead to data loss. By default, `mv` will not ask for confirmation before overwriting files. However, an additional option, `mv -i` (or `mv --interactive`), will cause `mv` to request such confirmation.

Note that `mv` also works on directories.

Let's move `quotes.txt` into the current working directory. We use `mv` once again, but this time we'll use just the name of a directory as the second argument to tell `mv` that we want to keep the filename but put the file somewhere new. (This is why the command is called 'move'.) In this case, the directory name we use is the special directory name `.` that we mentioned earlier.

```
mv thesis/quotes.txt .
```

3.6 Copying files and directories

The `cp` command works very much like `mv`, except it copies a file instead of moving it. We can check that it did the right thing using `ls` with two paths as arguments — like most Unix commands, `ls` can be given multiple paths at once:

```
cp quotes.txt thesis/quotations.txt
ls quotes.txt thesis/quotations.txt
```

We can also copy a directory and all its contents by using the [recursive](#) option `-r`, e.g. to back up a directory:

```
cp -r thesis thesis_backup
```

We can check the result by listing the contents of both the `thesis` and `thesis_backup` directory:

```
$ ls thesis thesis_backup
```

It is important to include the `-r` flag. If you want to copy a directory and you omit this option you will see a message that the directory has been omitted because `-r not specified`.

```
$ cp thesis thesis_backup
cp: -r not specified; omitting directory 'thesis'
```

3.7 Removing files and directories

Returning to the shell-lesson-data/exercise-data/writing directory, let's tidy up this directory by removing the quotes.txt file we created. The Unix command we'll use for this is rm (short for 'remove'):

```
rm quotes.txt
```

We can confirm the file has gone using ls:

```
ls quotes.txt
```

Deleting Is Forever

The Unix shell doesn't have a trash bin that we can recover deleted files from (though most graphical interfaces to Unix do). Instead, when we delete files, they are unlinked from the file system so that their storage space on disk can be recycled. Tools for finding and recovering deleted files do exist, but there's no guarantee they'll work in any particular situation, since the computer may recycle the file's disk space right away.

3.8 Using rm Safely

What happens when we execute `rm -i thesis_backup/quotations.txt`? Why would we want this protection when using `rm`?

Solution (Solution).

```
rm: remove regular file 'thesis_backup/quotations.txt'? y
```

The `-i` option will prompt before (every) removal (use Y to confirm deletion or N to keep the file). The Unix shell doesn't have a trash bin, so all the files removed will disappear forever. By using the `-i` option, we have the chance to check that we are deleting only the files that we want to remove.

If we try to remove the thesis directory using `rm thesis`, we get an error message:

```
rm thesis
```

This happens because `rm` by default only works on files, not directories.

`rm` can remove a directory and all its contents if we use the recursive option `-r`, and it will do so without any confirmation prompts:

```
rm -r thesis
```

Given that there is no way to retrieve files deleted using the shell, `rm -r` should be used with great caution (you might consider adding the interactive option `rm -r -i`).

3.9 Using Wildcards for Accessing Multiple Files at Once

Wildcards are special characters that allow you to access multiple files at once. They are particularly useful for handling groups of files without needing to list each one individually.

3.9.1 The Asterisk (*)

The `*` wildcard represents zero or more characters. For example, in the `shell-lesson-data/exercise-data/alkanes/` directory:

- `*.pdb` matches all files ending with `.pdb`, such as `ethane.pdb`, `propane.pdb`, and any other files with the `.pdb` extension.
- `p*.pdb` matches files starting with the letter `p` and ending with `.pdb`, such as `pentane.pdb` and `propane.pdb`.

```
cd ../mamun/shell-lesson-data/exercise-data/alkanes/  
ls *.pdb
```

This command lists all `.pdb` files in the current directory. If you use:

```
ls p*.pdb
```

It lists only `.pdb` files that start with `p`.

3.9.2 The Question Mark (?)

The `?` wildcard represents exactly one character. For example:

- `?ethane.pdb` could match `methane.pdb` but not `ethane.pdb`, because `?` represents exactly one character.
- `*ethane.pdb` matches both `ethane.pdb` and `methane.pdb`, as the `*` can represent zero or more characters.

3.9.3 Combining Wildcards

Wildcards can be combined for more specific patterns. For example:

- `???ane.pdb` matches files with exactly three characters followed by `ane.pdb`, such as `cubane.pdb`, `ethane.pdb`, and `octane.pdb`.

3.9.4 Handling Non-Matching Wildcards

If a wildcard pattern does not match any files, the shell passes the pattern as it is to the command, which can result in an error. For example, if you type:

```
ls *.pdf
```

in a directory containing only `.pdb` files, you will receive an error stating that no such file or directory exists.

3.10 Summary

- `cp [old] [new]` copies a file.
- `mkdir [path]` creates a new directory.
- `mv [old] [new]` moves (renames) a file or directory.
- `rm [path]` removes (deletes) a file.
- **cat**: Use for small files to quickly view the entire content.
- **head**: Use to view the beginning of a file.
- **tail**: Use to view the end of a file.
- **less**: Use for large files to navigate through content efficiently.
- **more**: Similar to **less**, but with fewer features.
- `*` matches zero or more characters in a filename, so `*.txt` matches all files ending in `.txt`.
- `?` matches any single character in a filename, so `? .txt` matches `a.txt` but not `any.txt`.

- Use of the Control key may be described in many ways, including `Ctrl-X`, `Control-X`, and `^X`.
- The shell does not have a trash bin: once something is deleted, it's really gone.
- Most files' names are **something.extension**. The extension isn't required, and doesn't guarantee anything, but is normally used to indicate the type of data in the file.
- Depending on the type of work you do, you may need a more powerful text editor than Nano.

4 Pipes and Filters

Now that we know a few basic commands, we can finally look at the shell's most powerful feature: the ease with which it lets us combine existing programs in new ways. We'll start with the directory `shell-lesson-data/exercise-data/alkanes` that contains six files describing some simple organic molecules. The `.pdb` extension indicates that these files are in Protein Data Bank format, a simple text format that specifies the type and position of each atom in the molecule.

First copy the `shell-lesson-data` folder into your working directory if you have not done this before.

```
cp -r /mnt/s-ws/everyone/shell-lesson-data .  
  
cd shell-lesson-data/exercise-data/alkanes/  
  
ls
```

4.1 `wc` 'word count' command

`wc` is the 'word count' command: it counts the number of lines, words, and characters in files (returning the values in that order from left to right).

Let's run an example command:

```
wc cubane.pdb
```

If we run the command `wc *.pdb`, the `*` in `*.pdb` matches zero or more characters, so the shell turns `*.pdb` into a list of all `.pdb` files in the current directory:

```
wc *.pdb
```

Note that `wc *.pdb` also shows the total number of all lines in the last line of the output.

If we run `wc -l` instead of just `wc`, the output shows only the number of lines per file:

```
wc -l *.pdb
```

The `-m` and `-w` options can also be used with the `wc` command to show only the number of characters or the number of words, respectively.

Why Isn't It Doing Anything?

What happens if a command is supposed to process a file, but we don't give it a filename? For example, what if we type:

```
$ wc -l
```

but don't type `*.pdb` (or anything else) after the command? Since it doesn't have any filenames, `wc` assumes it is supposed to process input given at the command prompt, so it just sits there and waits for us to give it some data interactively. From the outside, though, all we see is it sitting there, and the command doesn't appear to do anything. If you make this kind of mistake, you can escape out of this state by holding down the control key (Ctrl) and pressing the letter C once: Ctrl+C. Then release both keys.

4.2 Capturing output from commands

Which of these files contains the fewest lines? It's an easy question to answer when there are only six files, but what if there were 6000? Our first step toward a solution is to run the command:

```
$ wc -l *.pdb > lengths.txt
```

The greater than symbol, `>`, tells the shell to **redirect** the command's output to a file instead of printing it to the screen. This command prints no screen output, because everything that `wc` would have printed has gone into the file `lengths.txt` instead. If the file doesn't exist prior to issuing the command, the shell will create the file. If the file exists already, it will be silently overwritten, which may lead to data loss. Thus, **redirect** commands require caution.

`ls lengths.txt` confirms that the file exists:

```
$ ls lengths.txt
```

We can now send the content of `lengths.txt` to the screen using `cat lengths.txt`. The `cat` command gets its name from 'concatenate' i.e. join together, and it prints the contents of files one after another. There's only one file in this case, so `cat` just shows us what it contains:

```
$ cat lengths.txt
```

4.3 Filtering output

Next we'll use the `sort` command to sort the contents of the `lengths.txt` file. But first we'll do an exercise to learn a little about the `sort` command:

4.3.1 What Does `sort -n` Do?

The file `shell-lesson-data/exercise-data/numbers.txt` contains some lines with numbers:

```
sort ../numbers.txt  
  
sort -n ../numbers.txt
```

The `-n` option specifies a numerical rather than an alphanumerical sort.

We will also use the `-n` option to specify that the sort is numerical instead of alphanumerical. This does *not* change the file; instead, it sends the sorted result to the screen:

```
sort -n lengths.txt
```

We can put the sorted list of lines in another temporary file called `sorted-lengths.txt` by putting `> sorted-lengths.txt` after the command, just as we used `> lengths.txt` to put the output of `wc` into `lengths.txt`. Once we've done that, we can run another command called `head` to get the first few lines in `sorted-lengths.txt`:

```
sort -n lengths.txt > sorted-lengths.txt  
head -n 1 sorted-lengths.txt
```

Using `-n 1` with `head` tells it that we only want the first line of the file; `-n 20` would get the first 20, and so on. Since `sorted-lengths.txt` contains the lengths of our files ordered from least to greatest, the output of `head` must be the file with the fewest lines

4.4 The >> operator

4.4.1 Using >> in Bash Commands

In Bash, the >> operator is used to append the output of a command to a file. If the file does not already exist, it will be created. This operator is particularly useful when you want to add content to the end of an existing file without overwriting its current content.

4.4.2 Syntax

```
command >> filename
```

- **command:** The command whose output you want to append.
- **filename:** The file to which the output will be appended.

4.4.3 Example

Consider you have a file named `logfile.txt` and you want to append the current date and time to it each time a certain script runs.

4.4.3.1 Step 1: Create or check the initial content of `logfile.txt`

```
echo "Initial log entry" > logfile.txt  
cat logfile.txt
```

4.4.3.2 Step 2: Append the date and time to `logfile.txt`

```
date >> logfile.txt
```

4.4.3.3 Step 3: Check the updated content of `logfile.txt`

```
cat logfile.txt
```

4.4.4 Multiple Appends

You can use the `>>` operator multiple times to append different outputs to the same file. For example:

```
echo "First append" >> logfile.txt
echo "Second append" >> logfile.txt
```

4.5 Passing output to another command

In our example of finding the file with the fewest lines, we are using two intermediate files `lengths.txt` and `sorted-lengths.txt` to store output. This is a confusing way to work because even once you understand what `wc`, `sort`, and `head` do, those intermediate files make it hard to follow what's going on. We can make it easier to understand by running `sort` and `head` together:

```
$ sort -n lengths.txt | head -n 1
```

```
9  methane.pdb
```

The vertical bar, `|`, between the two commands is called a **pipe**. It tells the shell that we want to use the output of the command on the left as the input to the command on the right.

This has removed the need for the `sorted-lengths.txt` file.

4.6 Combining multiple commands

Nothing prevents us from chaining pipes consecutively. We can for example send the output of `wc` directly to `sort`, and then send the resulting output to `head`. This removes the need for any intermediate files.

We'll start by using a pipe to send the output of `wc` to `sort`:

```
$ wc -l *.pdb | sort -n
```

```
9 methane.pdb
12 ethane.pdb
15 propane.pdb
20 cubane.pdb
21 pentane.pdb
30 octane.pdb
107 total
```

We can then send that output through another pipe, to `head`, so that the full pipeline becomes:

```
$ wc -l *.pdb | sort -n | head -n 1
```

```
9 methane.pdb
```

This is exactly like a mathematician nesting functions like $\log(3x)$ and saying ‘the log of three times x ’. In our case, the algorithm is ‘head of sort of line count of `*.pdb`’.

4.7 Summary

- `wc` counts lines, words, and characters in its inputs.
- `cat` displays the contents of its inputs.
- `sort` sorts its inputs.
- `head` displays the first 10 lines of its input.
- `tail` displays the last 10 lines of its input.
- `command > [file]` redirects a command’s output to a file (overwriting any existing content).
- `command >> [file]` appends a command’s output to a file.
- `[first] | [second]` is a pipeline: the output of the first command is used as the input to the second.
- The best way to use the shell is to use pipes to combine simple single-purpose programs (filters).