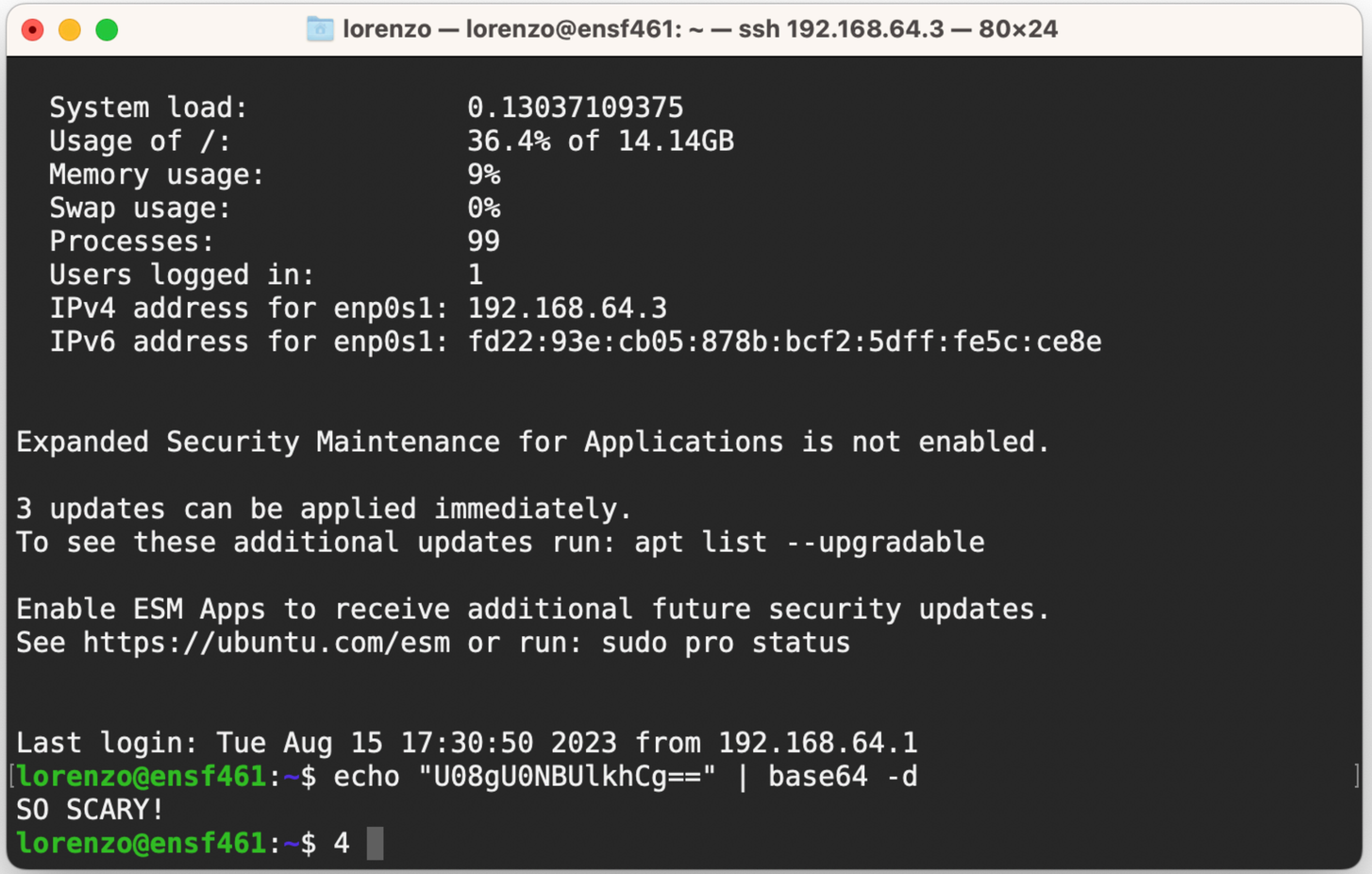
Lecture 02 - The shell 🐚

ENSF461 - Applied Operating Systems

Instructor: Lorenzo De Carli, University of Calgary (*lorenzo.decarli@ucalgary.ca*)
Slides by Lorenzo De Carli, partly based on material by Robert Walls (WPI)

What are we talking about today?

- The **command line**!
- A **mysterious interfaces** that **dark magicians** use to **control computers**
- Typically provided by a utility called a **shell**…
- Running in a **terminal**



No, seriously…

- Technically the way you interface to a OS is **irrelevant** to how it works
- In practice, however, a **low-level UI (command line)** is much more conducive to playing with **OS internals**
- Also, learning the command line is a **useful skill** in itself
  - Simplifies **system management**
  - Sometimes the **only UI available** (e.g., interfacing to remote server)

Important concepts

- The **structure** of the **command line**
- **Working directory**
- **Relative** vs. **absolute paths**.
  - The meaning of ./ or ../ in a path
- **Stdin** vs **Stdout** vs. **Stderr**

Yes, but what is a shell?

- A shell is a program that presents a **prompt** and waits for **text commands**
- **Commands** are typically the names of executable programs
- Most commonly, the shell finds the **program** indicated by the **command**, **executes it**, and **displays its output**
- **Note:** every time you see a line beginning with “**$**” in my slides, it means what follows is a shell command

More precisely…

- Users type **commands**, which correspond to files on the disk. Commands may include **arguments / parameters**
- These files are compiled programs (typically). They are often called **binaries**.
- Through a complicated set of actions, the OS loads the binary into memory and executes it.
- An executing program/binary is called **a process**.
- The OS stores metadata about each process, e.g., the **process ID (PID)**.
- **Command-line applications** often interact with each other and the user through files, e.g., file redirection, pipes, stdin, stdout, etc.
- Shells use **environment variables** to store configuration information, e.g., $PATH denotes where to look for binaries.

Common shells

- **sh:** Bourne Shell: the original UNIX shell, developed at Bell Labs in the 1970s
- **bash:** Bourne Again Shell: GNU project extension of sh (default in Linux)
- **zsh:** Z Shell: extended version of bash w/ different syntax (default in MacOS)

- In this course we are going to use **bash** (which is probably what you would use anyway in any situation where you need to interact with a shell)

Before we get to see these concepts in practice

Where to find examples used in class

- All **coding examples** seen in class will be uploaded at **`https://github.com/ldklab/ensf461F23`**
- You can clone the **git repository** by running the following command: **`git clone https://github.com/ldklab/ensf461F23.git`**
- If you are not sure what do, don’t worry just yet! In **lab 1** (September 7th) we will show you how to build **your own virtual machine** to build/execute code

Let’s see those concepts in action

Don’t worry if you don’t yet understand all the details!

Let’s see those concepts in action

Don’t worry if you don’t yet understand all the details!

- Example 1: running a program (**`ex01_basicprogram.c`**)

Let’s see those concepts in action

Don’t worry if you don’t yet understand all the details!

- Example 1: running a program (**`ex01_basicprogram.c`**)
- Example 2: command line arguments (**`ex02_parameters.c`**)

Let’s see those concepts in action

Don’t worry if you don’t yet understand all the details!

- Example 1: running a program (**`ex01_basicprogram.c`**)
- Example 2: command line arguments (**`ex02_parameters.c`**)
- Example 3: process id aka PID (**`ex03_pid.c`**)

This is a good time…

… to introduce the concept of **working directory**

- How does the shell knows **where to find programs**?
- Well, before we get to that we need to introduce the concept of **environment variable**

What is an environment variable?

- A shell maintain some **metadata** that **alter its functioning**, or that of the programs that are executed
- This metadata is stored as **key-value pairs**:

```
VAR1_NAME = VAR1_VALUE
VAR2_NAME = VAR2_VALUE
…
```

- Each key-value pair represents a **variable** and its **value**
- Variables can be **read and/or modified** by the shell, programs, or the user

An example: PATH

The most important variable of all! 👑

- PATH tells the shell in which folders it should look for programs
- We can check its content with the echo command:
  - Let's try `$ echo $PATH`

An example: PATH

The most important variable of all! 👑

- PATH tells the shell in which folders it should look for programs
- We can check its content with the echo command:
  - Let's try **`$ echo $PATH`**

`echo`: command line utility that prints a string to the terminal

An example: PATH

The most important variable of all! 👑

- PATH tells the shell in which folders it should look for programs
- We can check its content with the echo command:
  - Let's try `$ echo $PATH`

`echo`: command line utility that prints a string to the terminal

`$VARNAME`: when the shell encouters "**`$`**", it replaces it with the value of the variable that follows

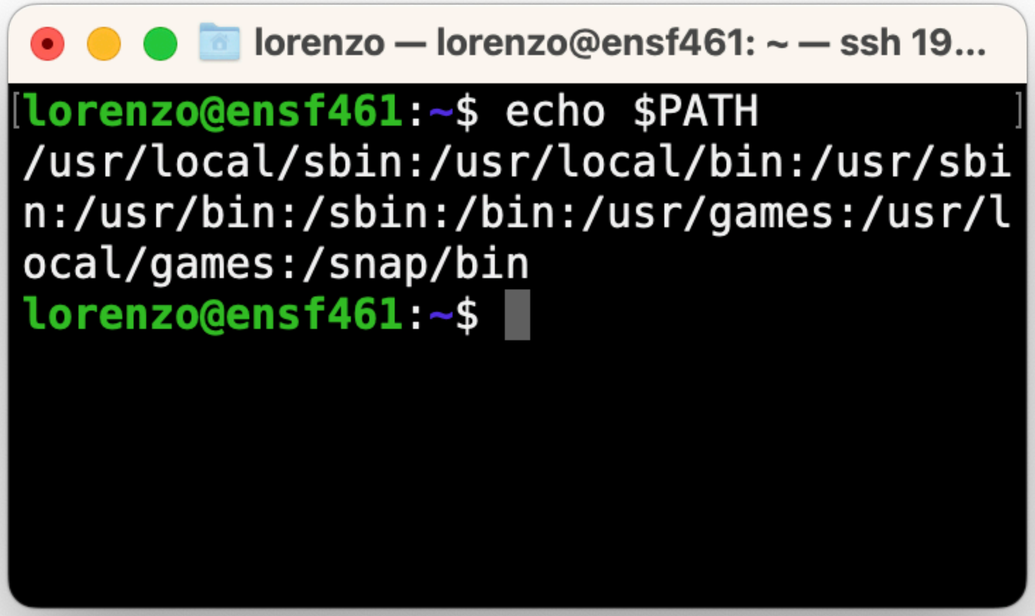
An example: PATH

The most important variable of all! 👑

- PATH tells the shell in which folders it should look for programs
- We can check its content with the echo command:
  - Let's try **`$ echo $PATH`**

`echo`: command line utility that prints a string to the terminal

`$VARNAME`: when the shell encouters "**`$`**", it replaces it with the value of the variable that follows



What if…

my program is not in one of the folders in PATH?

- Well, one option would be to add your folder to PATH:
  `$ PATH=$PATH:/path/to/my/folder`
- This is **very bad** and **thou shall never do it**! 😱☠️
- Why?
  - **Confusing:** you end up not knowing from where you are executing
  - **Bad for security**: you are not executing the program you think you are

What if…

my program is not in one of the folders in PATH?

- Well, one option would be to add your folder to PATH:
  `$ PATH=$PATH:/path/to/my/folder`
- This is **very bad** and **thou shall never do it**! 😱☠️
- Why?
  - **Confusing:** you end up not knowing from where you are executing
  - **Bad for security**: you are not executing the program you think you are

Incidentally, this is how you **set** an environmental variable

Well, what do I do then?

- The shell provides a **syntax** to express **“run a program from the current working directory”**
- But what does the above mean?

Well, what do I do then?

- The shell provides a **syntax** to express **“run a program from the current working directory”**
- But what does the above mean?
- **Any idea?**

Working directory

- The **command line** is different from the typical graphical UI
- Actions depend on the **context** where they are run…
- …where context mostly means a **location (directory)** in the **file system**

Working directory/2

- Bash (and other shells) offer a **shorthand** for “run a command from the current working directory”
- Simply **prepend** the name of the command with “**`./`**“
- E.g. “**`$ ./mycmd`**” will look for a program called **`mycmd`** in the current directory and run it
- Print current working directory: **`$ pwd`**
- Change working directory: **`$ cd path/to/new/working/directory`**

Working directory/3

- **Physical analogy:** image that you are the **shell** and I hand you a list of **instructions** to execute.
- **One of those instructions says:** turn around and grab the book labeled “Algorithm Design” from the bookshelf.
- If you are executing those instructions **in my office**, then you will have no problems. But if you execute **in a classroom** then **there is no book to grab**.

```
1 #!/bin/bash
2
3 echo "Currently in: `pwd`"
4 cat meow.txt
```

Working directory/3

- **Physical analogy:** image that you are the **shell** and I hand you a list of **instructions** to execute.
- **One of those instructions says:** turn around and grab the book labeled “Algorithm Design” from the bookshelf.
- If you are executing those instructions **in my office**, then you will have no problems. But if you execute **in a classroom** then **there is no book to grab**.

```
1 #!/bin/bash
2
3 echo "Currently in: `pwd`"
4 cat meow.txt
```

Incidentally, this is a **shell script** (more on this later)

Working directory/3

- **Physical analogy:** image that you are the **shell** and I hand you a list of **instructions** to execute.
- **One of those instructions says:** turn around and grab the book labeled “Algorithm Design” from the bookshelf.
- If you are executing those instructions **in my office**, then you will have no problems. But if you execute **in a classroom** then **there is no book to grab**.

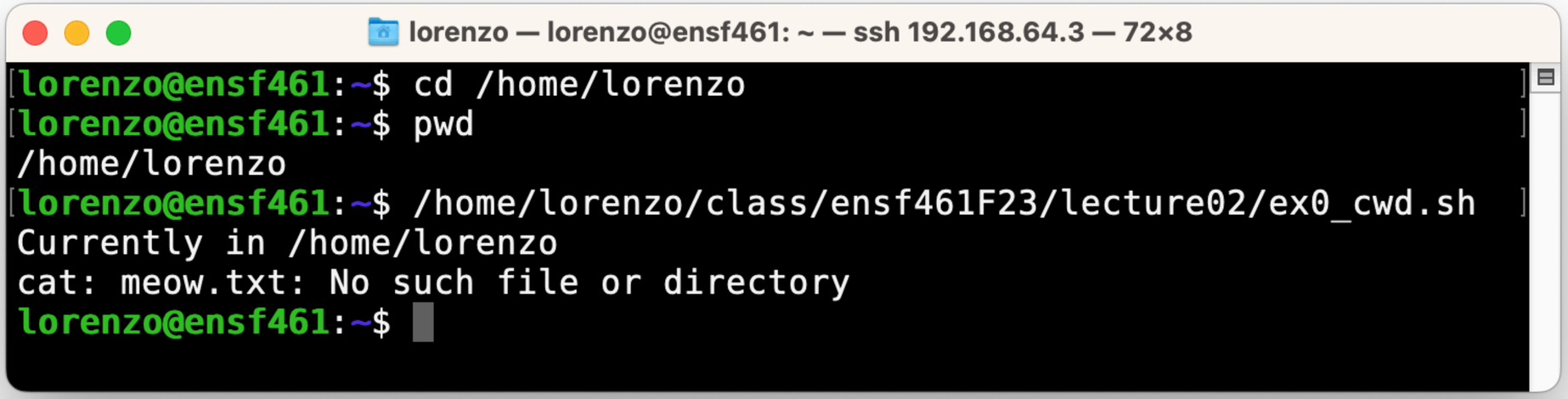
```
1 #!/bin/bash
2
3 echo "Currently in: `pwd`"
4 cat meow.txt
```

Incidentally, this is a **shell script** (more on this later)

Let’s try to execute this script in different folders

Working directory/4

- You can run a command in **any directory** by specifying the **full path**
- Keep in mind, however, that the working directory is going to be **the one active at the moment** (i.e., the one returned by **pwd**)



Working directory/5

A final note

- Directories are (as you know) hierarchical
- You can use "**../**" to refer to the parent directory of the one you are currently in
- Examples:
  - **`$ cd ../`** → change the current working directory to the parent
  - **`$ ../myprog`** → executes **`myprog`** in the parent directory

And that’s the end of the detour!

…now you know what is a working directory

Let’s take a break and take a quiz

Navigate to D2L->Quizzes->Quiz 2

More examples

Interaction between programs

- **Example:** the **`/dev`** folder contains a number of files. Count how many file names contain the string “tty”
- **Solution:** **`$ ls -1 /dev | grep tty | wc -l`**
- What’s happening here?
  - **`ls -1 /dev`** lists all files in /dev in a single column of text
  - **`grep tty`** lists all the lines that contain the string “tty”
  - **`wc -l`** counts the number of lines in the output

What's this "|" business?

- The "**|**" symbol represents a pipe operand
- An expression of the form "**`cmd1 | cmd2`**" means:
  - Execute the **`cmd1`** program
  - Take its output
  - Feed it as input to the **`cmd2`** program
  - **Let's get to the bottom of this**

Standard what?

A primer on UNIX file access

- In a UNIX-like system (like Linux), programs can **open**, **read**, and **write** files
  - **Duh!**
- They can also open **things that look like files**, but are not files
- The most common examples of this concept are:
  - Standard input (**stdin**)
  - Standard output (**stdout**)
  - Standard error (**stderr**)

Say what?

- In a UNIX-like system, whenever a program starts it automatically opens **three file-like object**
- **Standard output:** an output stream which by is printed on the terminal
  - By convention, used for **regular output**
- **Standard error:** like standard input, but used for error messages
- **Standard input:** an input stream which receives input from the terminal

What does this mean in practice

- Suppose you write a C program. When you run it, **you already have three “files” opened** that you can use to **receive input** and **emit output**
  - Those are not really files though - the **inputs** and **outputs** happen through **the terminal**

Who remembers how to write to a file in C?

Who remembers how to write to a file in C?

lorenzo — lorenzo@ensf461: ~/class/ensf461F23/lecture02 — ssh 192.168.64.3 — 19

```
WRITE(2)                                                    Linux Programmer's Manual

NAME
       write - write to a file descriptor

SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);

DESCRIPTION
       write() writes up to count bytes from the buffer starting at buf to the file referred to by the file descriptor fd.
```

Let’s see an example

Let’s see an example

```c
int main() {
    // Preparing the messages
    const char* out = "Hello, ENSF461!\n";
    const ssize_t out_l = strlen(out);
    const char* err = "Error, ENSF461!\n";
    const ssize_t err_l = strlen(err);

    // Let's write to the standard output
    write(1, out, out_l);

    // Let's write to the standard error
    write(2, err, err_l);

    return 0;
}
```

Let’s talk about what just happened

```c
int main() {
    // Preparing the messages
    const char* out = "Hello, ENSF461!\n";
    const ssize_t out_l = strlen(out);
    const char* err = "Error, ENSF461!\n";
    const ssize_t err_l = strlen(err);

    // Let's write to the standard output
    write(1, out, out_l);

    // Let's write to the standard error
    write(2, err, err_l);

    return 0;
}
```

Let’s talk about what just happened

```c
int main() {
    // Preparing the messages
    const char* out = "Hello, ENSF461!\n";
    const ssize_t out_l = strlen(out);
    const char* err = "Error, ENSF461!\n";
    const ssize_t err_l = strlen(err);

    // Let's write to the standard output
    write(1, out, out_l);

    // Let's write to the standard error
    write(2, err, err_l);

    return 0;
}
```

Initialize strings,
compute their length

Let’s talk about what just happened

```c
int main() {
    // Preparing the messages
    const char* out = "Hello, ENSF461!\n";
    const ssize_t out_l = strlen(out);
    const char* err = "Error, ENSF461!\n";
    const ssize_t err_l = strlen(err);

    // Let's write to the standard output
    write(1, out, out_l);

    // Let's write to the standard error
    write(2, err, err_l);

    return 0;
}
```

Initialize strings, compute their length

Write to file descriptor 1

Let’s talk about what just happened

```c
int main() {
    // Preparing the messages
    const char* out = "Hello, ENSF461!\n";
    const ssize_t out_l = strlen(out);
    const char* err = "Error, ENSF461!\n";
    const ssize_t err_l = strlen(err);

    // Let's write to the standard output
    write(1, out, out_l);

    // Let's write to the standard error
    write(2, err, err_l);

    return 0;
}
```

Initialize strings, compute their length

Write to file descriptor 1

Write to file descriptor 2

A few considerations

- The file descriptors 1 and 2 are **already defined at program startup**
  - No need to **open them**!
- There is **nothing special** about “output” or “error”
  - **By convention**, we send **regular output** to **stdout** and **errors** to **stderr**

What about standard input?

What do you think I should do to access it?

What about standard input?

What do you think I should do to access it?

```c
#include <unistd.h>
#include <string.h>

int main() {
    char inputbuf[1024];
    ssize_t bytesread;

    bytesread = read(0, inputbuf, sizeof(inputbuf));
    if ( bytesread < 1 )
        return -1;

    write(1, inputbuf, strlen(inputbuf)-1);
    return 0;
}
```

What about standard input?

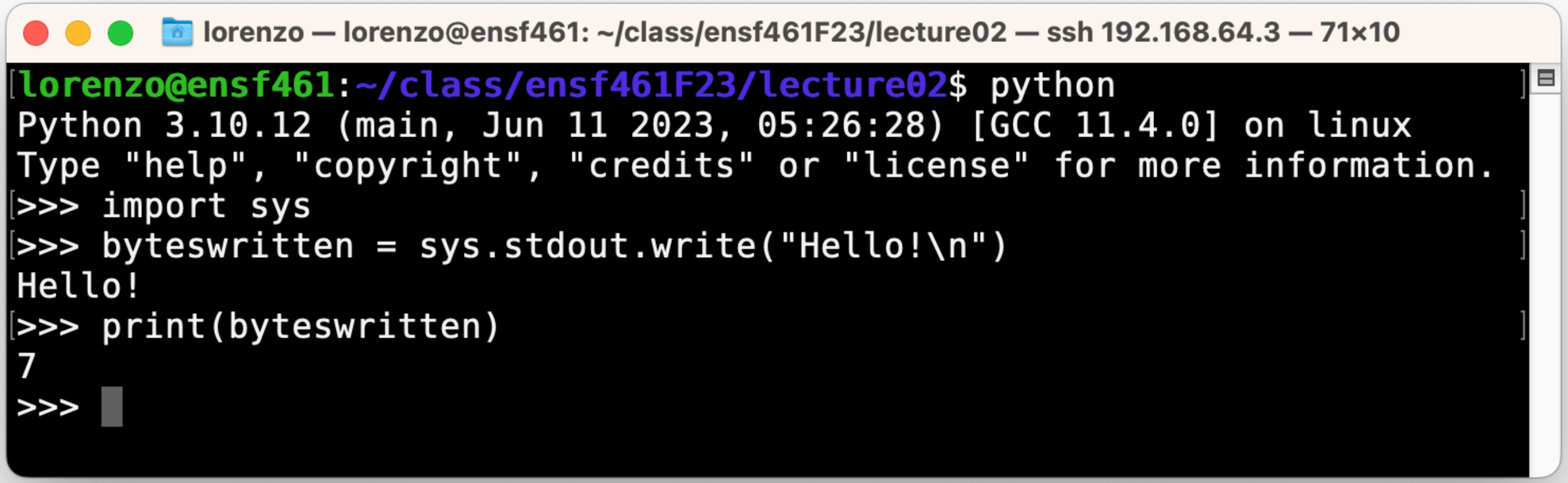
What do you think I should do to access it?

```c
#include <unistd.h>
#include <string.h>

int main() {
    char inputbuf[1024];
    ssize_t bytesread;

    bytesread = read(0, inputbuf, sizeof(inputbuf));
    if ( bytesread < 1 )
        return -1;

    write(1, inputbuf, strlen(inputbuf)-1);
    return 0;
}
```

Reads at most `sizeof(inputbuf)` bytes

Aside: works in Python too!

…you just need to know the syntax



Piping, again

- UNIX-like operating systems (Linux, MacOS) come with a large number of standard **command-line utilities** preinstalled
- We have already seen some examples:
  - **`grep`**: search for a string in each line of input
  - **`wc`**: counts the number of characters/words/lines in the input
- Unless specified, these **read from standard input** and **output to standard output**

Piping, again /2

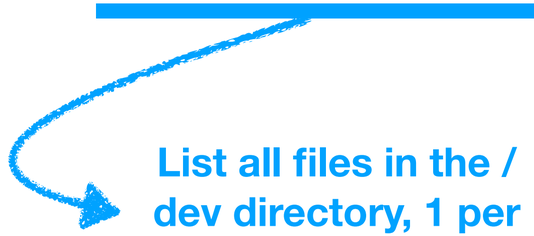
Let’s break down the example we saw earlier

```
$ ls -1 /dev | grep tty | wc -l
```

Piping, again /2

Let's break down the example we saw earlier

```
$ ls -1 /dev | grep tty | wc -l
```

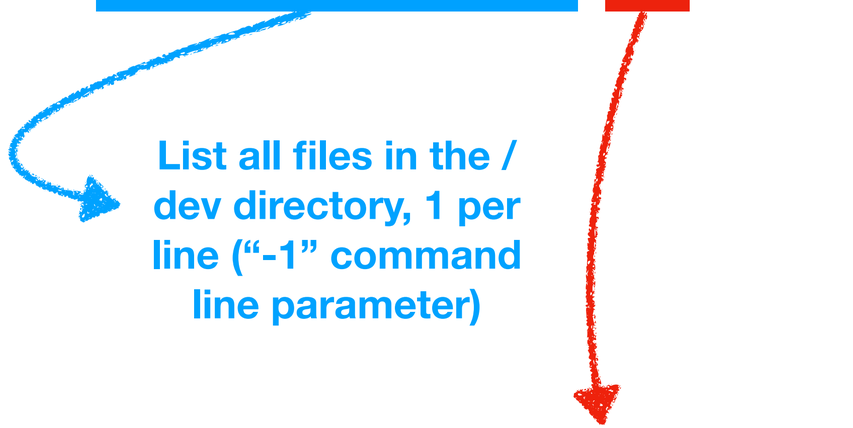


List all files in the /dev directory, 1 per line ("-1" command line parameter)

Piping, again /2

Let's break down the example we saw earlier

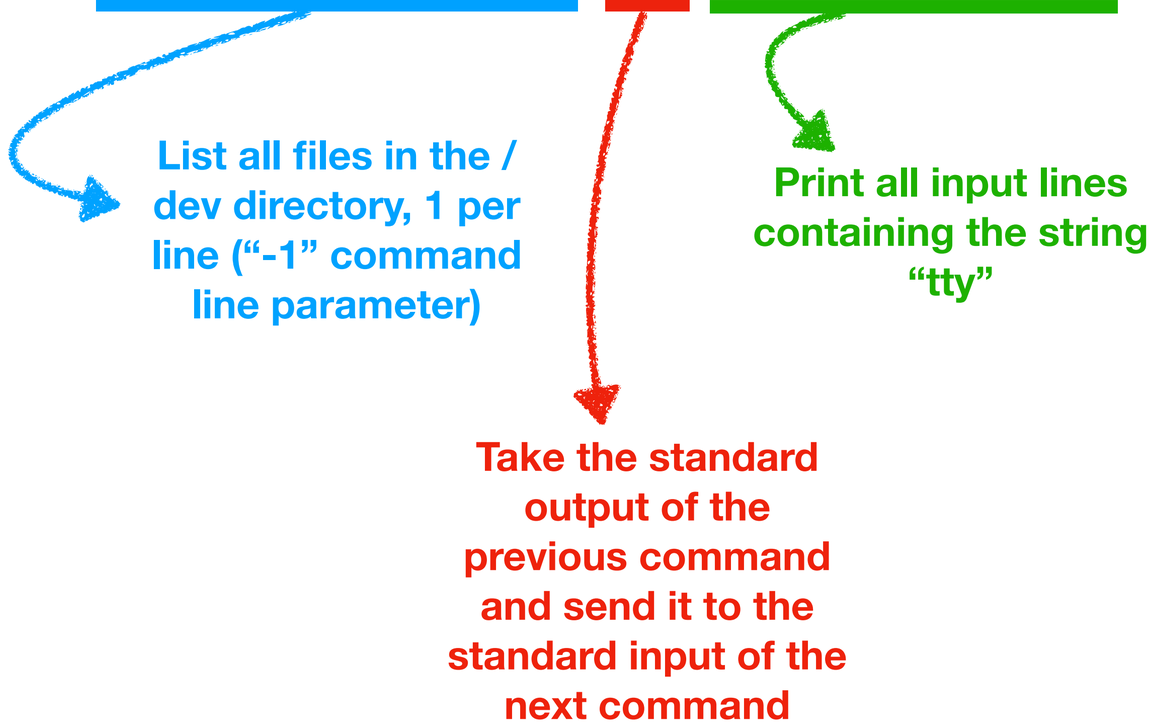
```
$ ls -1 /dev | grep tty | wc -l
```



Piping, again /2

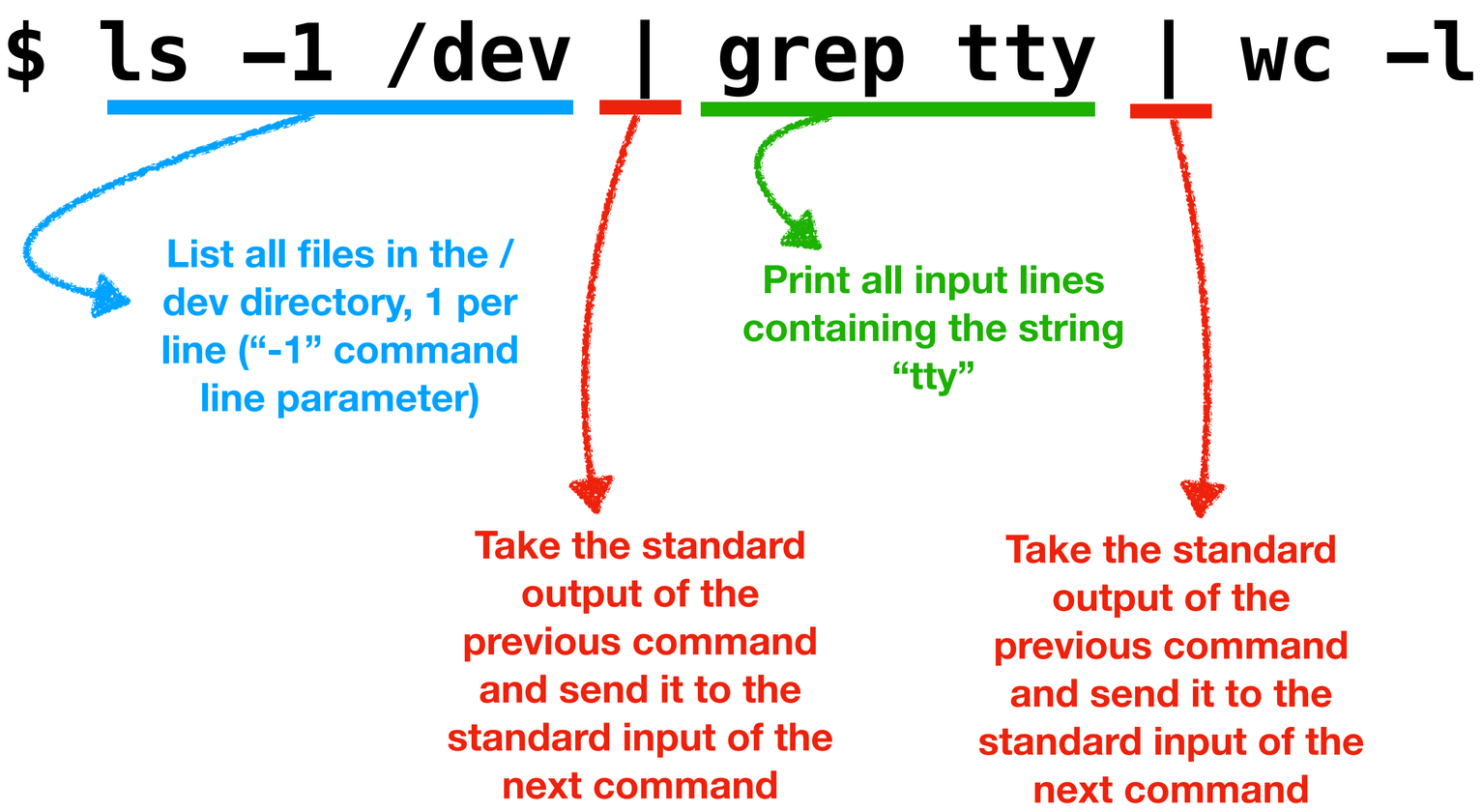
Let’s break down the example we saw earlier

```
$ ls -1 /dev | grep tty | wc -l
```



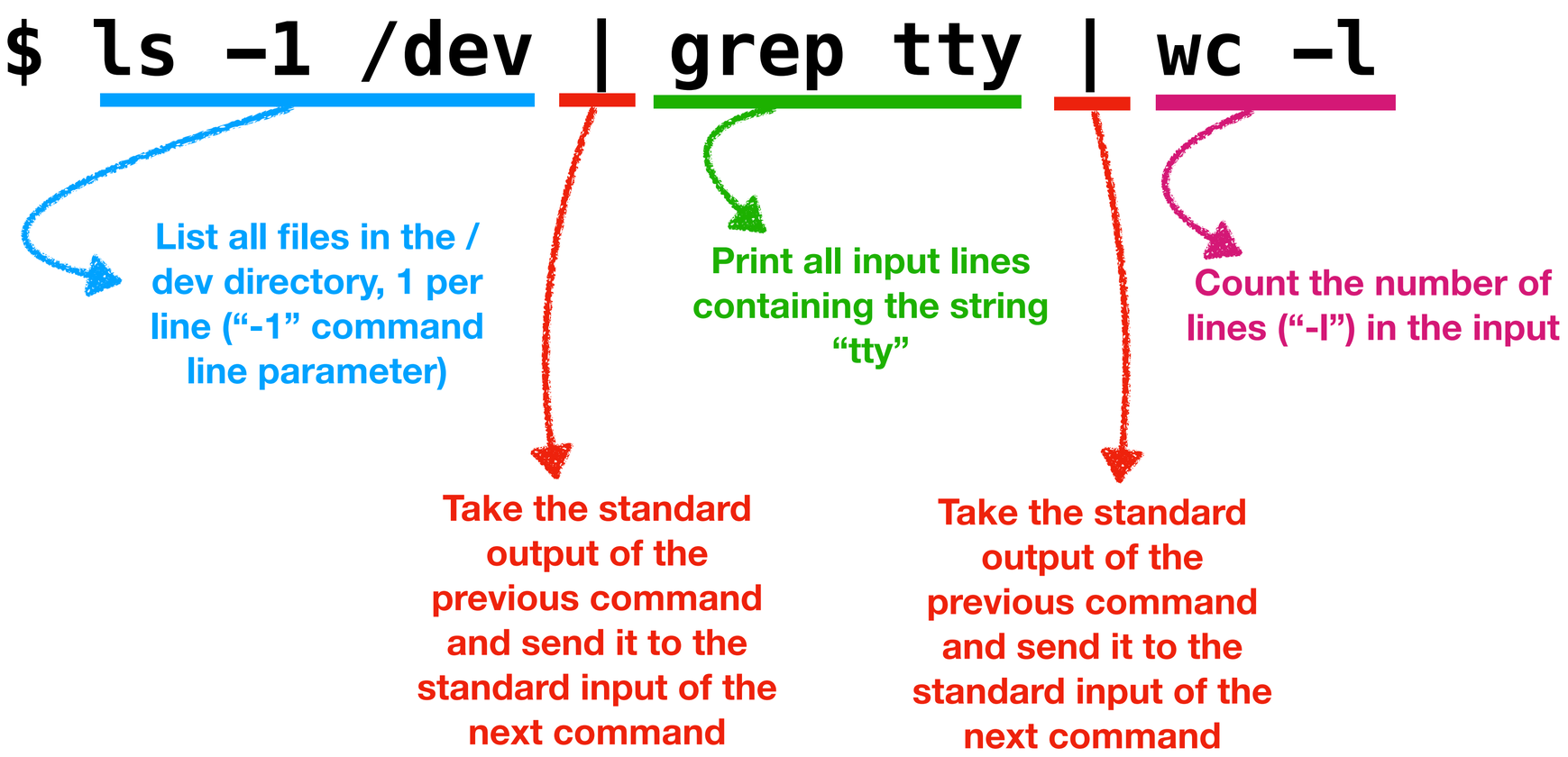
Piping, again /2

Let's break down the example we saw earlier



Piping, again /2

Let's break down the example we saw earlier



Redirecting standard input/error

- The shell can also **redirect** the **standard output/error** of a command to **file**
- …or even **redirect the standard output to error** (and **vice versa)**
- Useful to **save the output** of a chain of commands to a **file**
- **Let’s see how to do it**

Redirecting to file

- **`command > filename`**: save standard output to a new file **`filename`**
- **`command 2> filename`**: save standard error to a new file **`filename`**
  - If **`filename`** already exists, the commands above delete its entire content
- To avoid, use “**`>>`**” and “**`2>>`**” (append)
  - Example: **`command >> filename 2>> filename`**

Be careful!

- Something like **`command > outfile 2> outfile`** most likely **does not do what you want**

Be careful!

- Something like **`command > outfile 2> outfile`** most likely **does not do what you want**
- **Can you tell me what is the issue?**

Be careful!

- Something like **`command > outfile 2> outfile`** most likely **does not do what you want**
- **Can you tell me what is the issue?**
- If command generates both standard output and error, the first to be generated will **erase the other one**

More redirection

- You can **redirect** the **standard error** to the **standard output** with “**2>&1**”
- You can **redirect** the **standard output** to the **standard error** with “**1>&2**”
- Remember, **stdout** is **file descriptor 1**, and **stderr** is **file descriptor 2**

More redirection

- You can **redirect** the **standard error** to the **standard output** with “**2>&1**”
- You can **redirect** the **standard output** to the **standard error** with “**1>&2**”
- Remember, **stdout** is **file descriptor 1**, and **stderr** is **file descriptor 2**
- **Who can tell me what this does?**

```
command 2>&1 > output.txt
```

More redirection

- You can **redirect** the **standard error** to the **standard output** with “**2>&1**”
- You can **redirect** the **standard output** to the **standard error** with “**1>&2**”
- Remember, **stdout** is **file descriptor 1**, and **stderr** is **file descriptor 2**
- **Who can tell me what this does?**
  ```
  command 2>&1 > output.txt
  ```
- (Redirects the **standard error** to the **standard output**, then redirects the **standard output** to a **file** named “output.txt”)

That’s all for today!