Introduction to Unix

University of Massachusetts Medical School

October, 2015

CAN'T YOU DO ANYTHING RIGHT?
GLASBERGEN

DISCLAIMER

For the sake of clarity, the concepts mentioned in these slides have been simplified significantly.

Most of the command line tools and programs we explain have many more features that can't be found here. Unix command line tools are very well documented and their manual pages are freely available.

These slides are prepared for the users of MGHPCC. MGHPCC system has features that don't exist in standard Unix systems such as `loading modules`.

A rigorous treatment of topics on Unix and Bash can be found in various books in several levels.

Some References

Unix:
Unix in a Nutshell , Arnold Robbins, ISBN-13: 978-0596100292
Learning the bash Shell: Unix Shell Programming, Cameron Newham, ISBN-13: 978-0596009656

Umass High Perfotmance Computing:
http://wiki.umassrc.org/

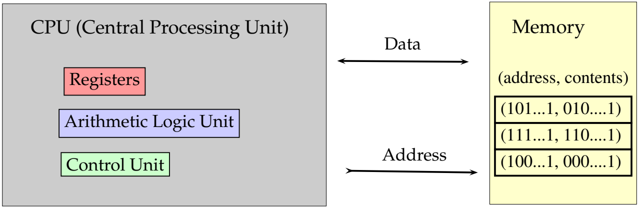
Bioinformatics:
UCSC genome bioinformatics: https://genome.ucsc.edu/
Sam file format: samtools.github.io/hts-specs/SAMv1.pdf
Samtools: http://samtools.sourceforge.net/
Bedtools: http://bedtools.readthedocs.org/en/latest/
bowtie2: http://bowtie-bio.sourceforge.net/bowtie2/manual.shtml
tophat: http://ccb.jhu.edu/software/tophat/index.shtml

Basic Components of a Computer

- **Input devices:** Keyboard, mouse, scanner, *terminal*,...
- **Output devices:** Screen, speaker, printer, *terminal*,...
- **Processors:** CPU (e.g., intel i7), GPU (GeForce), ...
- **Memory:** RAM, Level 3 Cache, ...
- **Storage Devices:** Flash Drives, Magnetic Drives, Optical Disks,...

Data and software are stored in the *storage devices* via *input devices*. Software and data are loaded into the *memory* and processed by the *processors* and we get the results via the *output devices*.

Computer Architecture


CPU (Central Processing Unit)

Registers

Arithmetic Logic Unit

Control Unit

Data

Address

Memory

(address, contents)

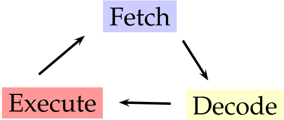
(101...1, 010....1)

(111...1, 110....1)

(100...1, 000....1)

This is called the *Von Neumann architecture*.

A cpu works in cycle to carry out instructions.


Fetch

Decode

Execute

Consequently, a processor of clock rate 4.5 GHz performs, in general, better than a processor (**in the same family**) of clock rate 4 GHz.

Fetch-Decode-Execute

Fetch: CPU has a special register that holds the memory location of the next instruction. It sets the address bus accordingly and gets the the next instruction.

Decode: Once it gets the instruction, it can figure out the memory location of the next instruction. It *understands* the instruction.

Execute: It performs the instruction.

Note

In today’s computers, many improvements have been made to optimize the cost and the performance.

Examples: Multiple cores, several levels of caches on the CPU, pipelining, ...

Wall Clock Time and CPU Time

We can talk about two different time measures when we run a process.

Wall Clock Time: The time difference between the start and end of the execution of a process.

CPU Clock Time: The actual time CPU spent in the fetch-decode-execute cycle to run the process.

Example

A simple experiment can show us that, even two different processes take almost the same time to finish, their CPU time usage can be different: First, create a dummy file of 1 GB.

```
$ dd if=/dev/zero of=dummy.img bs=1024 count=1024000
```

Then compute the hash sum of this file and time it

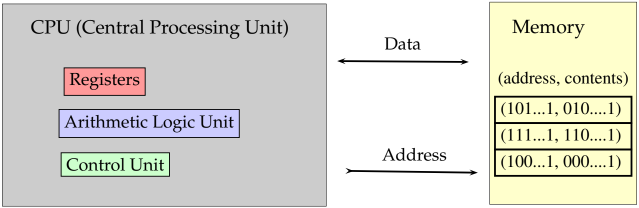
```
$ time md5sum dummy.img
```

Now run sleep for the same amount of seconds

```
$ time sleep 10
```

Notice the difference in *user + sys* between two runs.

Programming



From a technical point of view, we can define programing as generating the memory content in such a way that when the CPU processes the contents of the memory, we'll achieve a given objective.

Programming Languages

Today, programing is very rarely done by manually generating the contents of the memory.
We use *programming languages* instead (most of the time).

Why?

Programming Languages

Today, programing is very rarely done by manually generating the contents of the memory.
We use *programming languages* instead (most of the time).

Why?

Consider:
Binary Code:
`1011000011.....1101`

Programming Languages

Today, programing is very rarely done by manually generating the contents of the memory.
We use *programming languages* instead (most of the time).

Why?

Consider:
Binary Code:

```
1011000011.....1101
```

R Code:

```
for (gene_expression in RNA_Seq_Data){
 print(mean(gene_expression))
}
```

Programming Languages

A programming language is a formal definition of a system that translates human instructions to Binary (Machine) code. This translation is done by a piece of software which can be a *compiler* or an *interpreter*.

```
for (gene_expression in RNA_Seq_Data){
 print(mean(gene_expression))
}
```

$\xrightarrow{\text{interpreter eventually executes}}$ 0101...011111

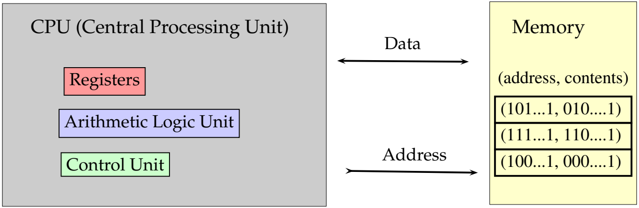
Processes & Threads

A **process** can be defined as the running instance of some software where the instructions are sequentially executed by the CPU.

A **thread** is the smallest unit of a process that can be run independently.

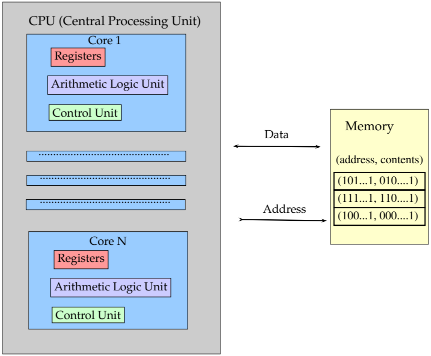
Both threads and processes can run in parallel. The main difference between them is that different threads of a process can share the same system resources (memory space, files and etc) whereas processes can **not**.

Processes & Threads



The Von Neumann architecture allows one process at a time to run. So how can multiple processes / threads can run in parallel?

Processes & Threads



Modern processors have more than one core.
So, if multi-cores are available and your software can take advantage of this, the run time of your process will be shorter.
It is advantageous to take this into consideration when submitting jobs and setting multithreading parameters of your bioinformatics tools. Examples to come in the following classes.

Exercise

You are using an short-read aligner software. Assume that its speed is linearly proportional to the number of threads it is working with. You have run this software using 3 threads and it took 90 minutes to align your RNA-Seq data. How much time, roughly, do you expect to wait if you run the same software, on the same data and, on the same computing environment, using 5 threads?

Exercise

You are using an short-read aligner software. Assume that its speed is linearly proportional to the number of threads it is working with. You have run this software using 3 threads and it took 90 minutes to align your RNA-Seq data. How much time, roughly, do you expect to wait if you run the same software, on the same data and, on the same computing environment, using 5 threads?

$$90 \times \frac{3}{5} = 54 \text{ minutes}$$

If we run it using a single thread, it works 3 times slower , compared to 3 threads, so the expected running time is $90 \times 3 = 270$ minutes. If we use 5 threads, it will be 5 times faster, compared to single thread, so 270 / 5 = 54 minutes.

Process Management

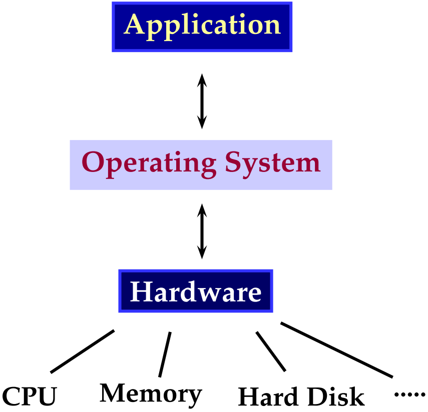
- What if you want to run 40 processes *simultaneously* using a 4 core CPU? (Process management)
- How can you make sure that a process doesn't read / write the memory contents of another process? (Memory management)
- What if you want to make a single computing environment available to many users and make sure that system resources are safely and fairly shared? (File permissions and etc.)

This is orchestrated by a group of software called *Operating System*.

What is an Operating System?

An operating system is a collection of software that manages system resources (such as memory, cpu(s), display and etc.) and provide applications a simpler interface to the system hardware.

Operating System



Application

Operating System

Hardware

CPU

Memory

Hard Disk

.....

Unix

Developed in the 1960's by Ken Thompson, Dennis Ritchie. It became very popular in the 70's and 80's.

"Contrary to popular belief, Unix is user friendly. It just happens to be very selective about who it decides to make friends with."
–anonymous

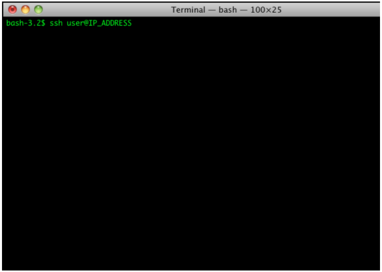
Unix-like operating systems

Unix-like operating systems *"behave"* like the original Unix operating system and comply (at least partially) with POSIX (portable operating system interface) standards.

Examples: Linux, OS X, Free BSD, Solaris, Hurd.

Unix $\equiv$ Unix-like $\equiv$ Unix-clone

What is a Terminal?



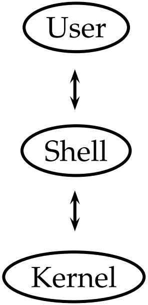
```
bash-3.2$ ssh user@IP_ADDRESS
```

A terminal is a software that emulates a teletype writer terminal used in the early days of Unix.

Unix Shell

A shell is a software that runs inside a terminal and interprets and executes user commands.

One of the most popular shells being used today is called **BASH** (Bourne Again Shell). It is also the default shell in the UMass HPC cluster, OS X and many Linux distributions.



Convention

$ denotes the Bash command line prompt. It is not meant to be typed. All Bash commands will appear in a gray box.

```
$ bashcommand
```

Hello Bash

We connect to the MGHPCC server.

```
$ ssh username@ghpcc06.umassrc.org
```

Hello Bash

We connect to the MGHPCC server.

```
$ ssh username@ghpcc06.umassrc.org
```

Can’t see your password being typed on the screen?

That is a good sign!
For security reasons, you shouldn’t see your password on the screen when you type it!

Hello Bash

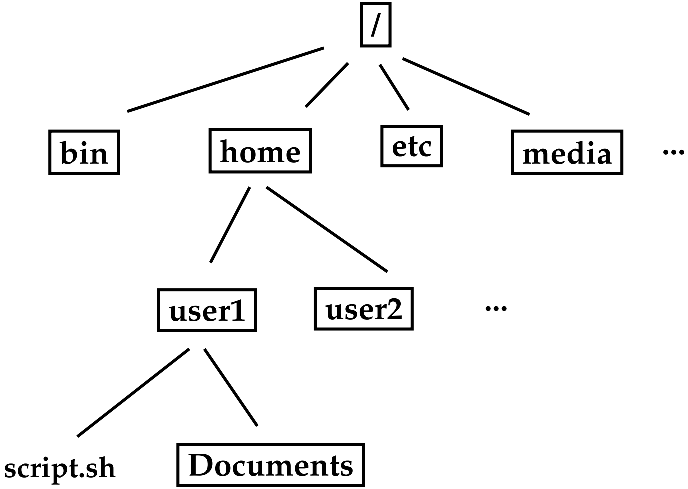
Let’s verify that we are at the right place.

```
$ hostname
```

To print something on the screen, we use echo

```
$ echo Hello World
```

Navigating the Unix File System



Exploring the Directories

Bash always has a working directory. To see the working directory,

```
$ pwd
```

Exploring the Directories

Bash always has a working directory. To see the working directory,

```
$ pwd
```

To see the contents of the working directory,

```
$ ls
```

Exploring the Directories

Bash always has a working directory. To see the working directory,

```
$ pwd
```

To see the contents of the working directory,

```
$ ls
```

To see the contents of a particular directory, provide the path of the directory.

```
$ ls /bin
```

displays th contents of the directory `/bin`

Command Line Arguments

A program's execution and / or output format can be modified, according to our needs, by providing command line arguments.

$$\$\quad \underbrace{\texttt{ls}}_{\text{executable}}\ \underbrace{\texttt{-lh}}_{\text{argument 1}}\ \underbrace{\texttt{/bin}}_{\text{argument 2}}$$

More on ls

We can get more information about the files or directories by running `ls` with additional arguments.

Compare

```
$ ls
```

with

```
$ ls -l
```

A closer look at ls

`ls` lists the given directory contents.
To see the complete list of options,

```
$ man ls
```

This will display the manual page of `ls`.

A closer look at ls

`ls` lists the given directory contents.
To see the complete list of options,

```
$ man ls
```

This will display the manual page of `ls`.

Some commonly used options are:

-l : list contents in more detail

-A : list all files in the direcory

-h : when used with the `-l` option,
prints file sizes in KB, MB, GB, and etc.
to reduce the number of digits on the screen.

These options can be grouped

```
$ ls -lh
```

Let’s first have some files and folders to experiment with.

```
$ /tmp/bootcamp.pl
```

Upon success, you will see
/home/user_name/bootcamp

Changing the working directory

First, we see the working directory

```
$ pwd
```

Now, lets change the directory to the `bootcamp` directory.

```
$ cd ~/bootcamp
```

We can verify the change by

```
$ pwd
```

Tab for Autocompletion

You can use the `tab` key for autocompletion. This can save you time and typing for long file / directory names.

```
$ touch this_is_a_somewhat_long_name.txt
```

To delete this file, after `rm`, type `this_` and hit the *tab* key.

```
$ rm this_
```

Creating Directories

```
$ mkdir directory_path
```

Note that

```
$ mkdir /home/username/bootcamp/my_new_directory
```

is the same as

```
$ mkdir ~/bootcamp/my_new_directory
```

Copying Files

```
$ cp sourcefile(s) destination_path
```

Let’s copy our sample files into the home folder.

```
$ cp ~/bootcamp/transcriptomics.tar.gz ~/transcriptomics.tar.gz
```

Copying Directories

```
$ cp source_directory destination_directory
```

will not work in general.
You need to provide the **-r** parameter.

```
$ cp -r source_directory destination_directory
```

Another useful option is `-v` which prints the files being copied.

```
$ cp -rv source_directory destination_directory
```

Moving Files

```
$ mv source_file destination_file
```

Deleting Files

`rm` : Remove comand

```
$ rm file_path
```

To remove non-empty directories, use rm with the `-r` option.

```
$ rm -r directory
```

rm is a dangerous command

There is no undoing of rm!
There is no command like `unrm my_file` or
bring_back my_file.
A file is a bunch of 0's and 1's in a storage device and the location of these 0's and 1's are stored in a table. Once you remove the file, it is very likely that those 0's and 1's will be overwritten and hence flipped. So it is very unlikely that you'll recover the file.

You may want to use it in the interactive mode not to delete files accidentally.

```
$ rm -i file
```

Back up Your Data!

It is good practice to backup your data regularly. You can use a remote machine or an external drive to take backups.

File Path

~	:	Home directory
.	:	current directory
..	:	parent directory

- If a file path begins with `/`, then it is an ***absolute path***. It doesn't depend on the current working directory.
- If a file path begins **WITHOUT** `/`, then it is a ***relative path***. The path is defined according to the current directory. So, the path **depends** on the current working directory.

Examples: Absolute vs Relative Path

`/tmp/bootcamp/bootcamp.pl` : Begins with `/` so, **absolute path**.

`bootcamp.pl` : This is the `botcamp.pl` file in the **working** directory. So, **relative path**

`../bootcamp.pl` : This is the `botcamp.pl` file in the **parent** directory of the working directory. So, **relative path**

`~/bootcamp.pl` : This is the file in the **home** directory of the current user. So, **relative path**.

Files

Technically, every file is a binary file.

Yet, it is common practice to group the files into two:

`Text Files` : Contains only printable characters.
`Binary Files` : Files that are not text files.

Binary files are generally more compact and therefore smaller in size. They are read and processed by specific applications.

Viewing text files

Text files are of extreme importance in bioinformatics.
There are many tools to view and edit text files.
`less` is a very useful program.

```
$ less text_file
```

For details read the manual page

```
$ man less
```

Editing Text Files

For a very simple text editor:

```
$ nano
```

`vi` and **`emacs`** are very powerful text editors with sophisticated features.

File Permissions

Let’s try to run a bash script.

```
$ ./hello.sh
```

Let’s try another one

```
$ ./other_hello.sh
```

Why didn’t this work?

File Permissions

Let’s try to spy on Hakan and see what he is up to.

```
$ ls /home/ho86w
```

File Permissions

Let’s try to spy on Hakan and see what he is up to.

```
$ ls /home/ho86w
```

In Unix, users can only access (read, write or execute) files that they have permission to.

Seeing File Permissions

A typical output of

```
$ ls -lh
```

is

```
-rw-r–r–       1   ho86w   moore    11M   Nov 5    00:41   RPKM.csv
-rwxrwxrwx     1   ho86w   moore   143B   Oct 27   02:53   my_script.sh
drwxr-xr-x    21   ho86w   moore   714B   Jan 24   13:52   sam_files
```

r	:	read
w	:	write
x	:	execute

$$\underbrace{\text{rwx}}_{\text{user}}\ \underbrace{\text{r-x}}_{\text{group}}\ \underbrace{\text{r- -}}_{\text{others}}$$

User : Has read, write and execute access.

Group : Has read and execute access but can’t write to the file

Others : Has only read access.

Changing File Permissions

chmod: Change file mode bits.

```
$ chmod filemode file
```

To give only read access to other users,

```
$ chmod o=r myscript.sh
```

To give read and execution access together,

```
$ chmod o=rx myscript.sh
```

What is a Process?

A computer process is an instance of a program. It takes up space in the memory and its instructions are executed in the CPU.

Processes

You can list **your** running processes

```
$ ps -u username
```

where `username` is your username. You can see them in more detail with the `-F` option

```
$ ps -u username -F
```

Terminating a process

You can terminate a process by

```
$ kill -9 processID
```

where `processID` is the process id that can be obtained by the `ps` command.

Background Processes

To send a process to the background, first put the process on hold by pressing `ctrl` and z, and then send it to the background by

```
$ bg
```

You can check your processes by

```
$ jobs
```

To put a back to the foreground, first get the process id by

```
$ jobs
```

and then

```
$ fg % processNumber
```

To start a process on the background, put & at the end of your command

```
$ command &
```

Working with Compressed Files

A common type of binary files is zipped files. We can save space by zipping text files.
We use
`gzip` to compress / decompress files
`tar` to pack files into one file for archiving purposes.
To compress a file

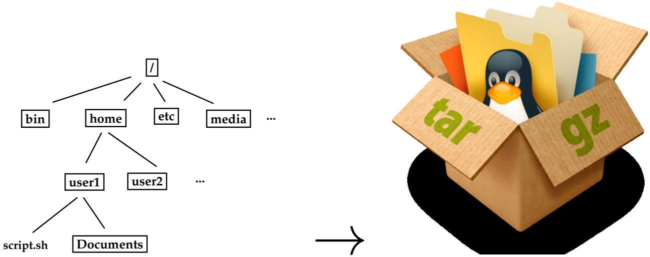
```
$ gzip file
```

To decompress, `reads.fastq.gz`,

```
$ gzip -d reads.fastq.gz
```

tar files

Say we collect our results on a project in one directory. We want to send this directory in an email attachment to a collaborator. How do we do that?



tar files

To pack a directory

```
$ tar -cvf arcive.tar directory
```

To pack a directory in a zipped tar file

```
$ tar -czvf archive.tar.gz directory
```

To get our directory back from the tar file

```
$ tar -xvf archive.tar
```

To get our directory back from the tar.gz file

```
$ tar -xvzf archive.tar.gz
```