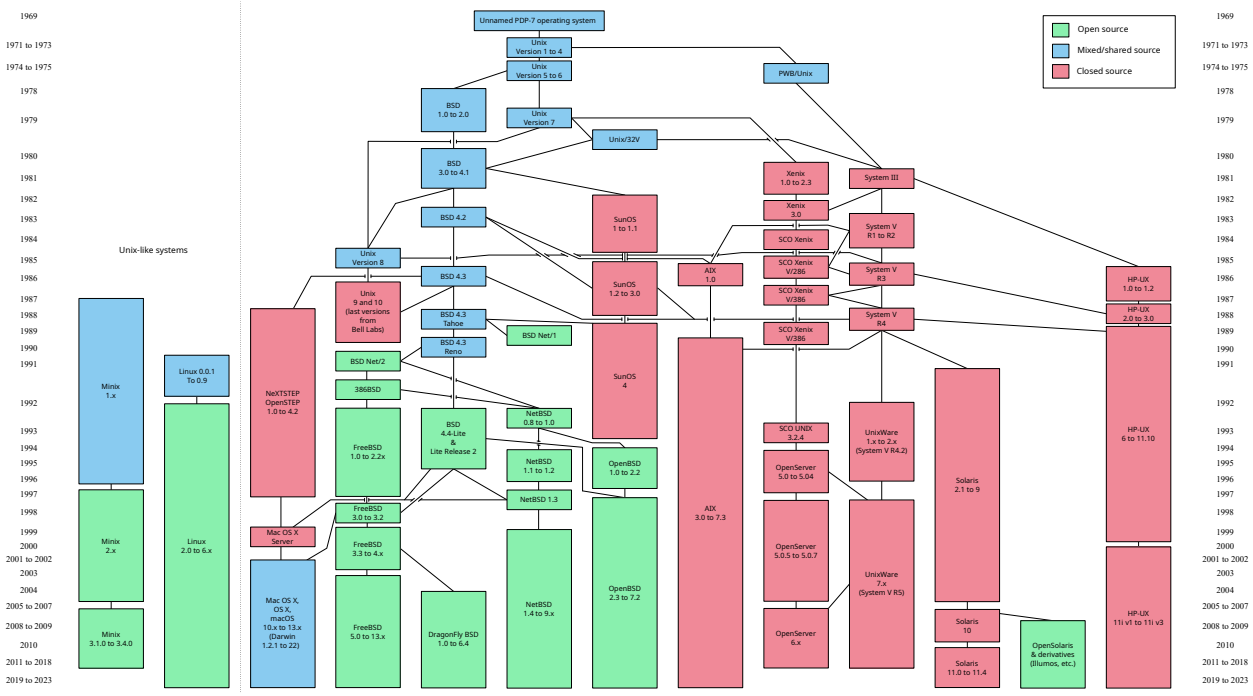
# Introduction to Unix/Linux shell

## What is (was) Unix

- Multi-user, multi-tasking Operating System
- Developed in the late 1960s at AT&T's Bell Labs
- Family or lineage of operating systems




## Unix - key aspects

- Multi-User & Multi-Tasking
- Hierarchical File System (root directory and subdirectories)
- "Everything is a File" Philosophy (system resources & hardware devices)
- Command-Line Interface (CLI): **shell** interpreter
- Pipes and Redirection (chain single-purpose commands & resources/devices)

## Jupyter & Unix shell

There are several ways to run shell commands directly within a Jupyter Notebook:

- Using the Exclamation Mark ( `!` )
- Using Cell Magics ( `%%bash` , `%%sh` , `%%script` )
- Using Python's `subprocess` Module

## Using the Exclamation Mark ( `!` )

Inside a code cell, prefix the shell command you want to run with an exclamation mark:

In [1]:
```
!pwd # Print working directory
```

```
/home/jupyter-mpenagaricano/Programming-for-AI
```

In [2]:
```
!ls # List all files
```

```
0_Index.ipynb             html         README.md
1b_unix_shell.ipynb       img          tmp
1c_git_and_github.ipynb   myutils.py   Untitled.ipynb
2a_NumPy.ipynb            pdf          WirelessPresenter.ipynb
2b_Pandas.ipynb           pdf_slides
2c_Matplotlib.ipynb       __pycache__
```

In [3]:
```
!for x in {1..5}; do echo $x ; done # print numbers from 1 to 5
```

```
1
2
3
4
5
```

**NOTE:** Depending on the host OS, the shell will be different. Our host is running Ubuntu 22.04, a linux distribution:

In [4]:
```
!cat /etc/os-release
```

```
PRETTY_NAME="Ubuntu 22.04.5 LTS"
NAME="Ubuntu"
VERSION_ID="22.04"
VERSION="22.04.5 LTS (Jammy Jellyfish)"
VERSION_CODENAME=jammy
ID=ubuntu
ID_LIKE=debian
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"
UBUNTU_CODENAME=jammy
```

### Capturing shell output into a Python Variable

You can assign the standard output of a shell command to a Python variable using the `variable = !command` syntax:

In [5]:
```
files = !ls
print(type(files))
```

```
<class 'IPython.utils.text.SList'>
```

In [6]:
```
import IPython
#help(IPython.utils.text.SList)
```

In [7]:
```
files = !ls
for x in files :
    print(type(x),x)
```

```
<class 'str'> 0_Index.ipynb
<class 'str'> 1b_unix_shell.ipynb
<class 'str'> 1c_git_and_github.ipynb
<class 'str'> 2a_NumPy.ipynb
<class 'str'> 2b_Pandas.ipynb
<class 'str'> 2c_Matplotlib.ipynb
<class 'str'> html
<class 'str'> img
<class 'str'> myutils.py
<class 'str'> pdf
<class 'str'> pdf_slides
<class 'str'> __pycache__
<class 'str'> README.md
<class 'str'> tmp
<class 'str'> Untitled.ipynb
<class 'str'> WirelessPresenter.ipynb
```

### Using Python Variables in Shell Commands

You can pass Python variables into your shell commands by enclosing the variable name in curly braces `{}` :

In [8]:
```
dirname = "img"
!ls {dirname}
```

```
array_vs_list.png     git-branches.png       split-apply-combine.png
broadcasting.png      git_branch_merge.png   Unix_history-simple.svg
centralized_vcs.webp  git_commands2.webp     VCS_Diff-768x314.png
distributed_vcs.webp  git_commands.webp
```

## Using Cell Magics ( `%%bash` , `%%sh` , `%%script` )

With Cell Magics, you can run multiple lines of shell script within a single cell:

In [9]:
```
%%bash
echo "STARTING THE SCRIPT (dir: $(pwd))"
for f in * /dev/null ; do
    if [ -d ${f} ] ; then
        echo "  ${f}: directory"
    elif [ -f ${f} ] ; then
        echo "  ${f}: regular file"
    else
        echo "  ${f}: not regular file"
    fi
done
echo "SCRIPT FINISHED!!"
```

```
STARTING THE SCRIPT (dir: /home/jupyter-mpenagaricano/Programming-for-AI)
  0_Index.ipynb: regular file
  1b_unix_shell.ipynb: regular file
  1c_git_and_github.ipynb: regular file
  2a_NumPy.ipynb: regular file
  2b_Pandas.ipynb: regular file
  2c_Matplotlib.ipynb: regular file
  html: directory
  img: directory
  myutils.py: regular file
  pdf: directory
  pdf_slides: directory
  __pycache__: directory
  README.md: regular file
  tmp: directory
  Untitled.ipynb: regular file
  WirelessPresenter.ipynb: regular file
  /dev/null: not regular file
SCRIPT FINISHED!!
```

### `%%sh` vs `%%bash`

- `sh` (Bourne Shell): Developed at Bell Labs in the late 1970s. It was the original standard Unix shell.
- `bash` (Bourne-Again Shell): Created for the GNU Project in the late 1980s as a free software replacement and enhancement for `sh`.
  - Includes almost all features of `sh` but adds many modern conveniences:
  - Command History, Tab Completion, Arrays, Brace Expansion, Extended Globbing, Richer Arithmetic, Process Substitution...

### Using `%%script <interpreter>`

You can run the cell content with a specific interpreter (e.g., perl, tcl, java...).

In [10]:
```
%%script perl
print("Hello World!!\n");
```

```
Hello World!!
```

In [11]:
```
%%script tclsh
puts "Hello World!!"
```

```
Hello World!!
```

In [12]:
```
%%script javash
System.out.println("Hello World!!")
```

```
Couldn't find program: 'javash'
```

## Using Python's `subprocess` Module

Python's built-in `subprocess` module allows more fine-grained control, error handling, capturing stderr separately, or handling complex interactions. This is the standard Python way to run external commands from Python scripts.

In [13]:
```
import subprocess

# Run a command and capture output.
#  capture_output=True captures stdout and stderr
#  check=True raises error if command fails
#  text=true decodes stdin, stdout and stderr using the given/default encoding
result = subprocess.run(['ls','img'], capture_output=True, text=True, check=True)
```

```
print("Return Code:", result.returncode)
print("Stdout:")
print(" ",result.stdout)
print("Stderr:")
print(" ",result.stderr)
```

```
Return Code: 0
Stdout:
  array_vs_list.png
broadcasting.png
centralized_vcs.webp
distributed_vcs.webp
git-branches.png
git_branch_merge.png
git_commands2.webp
git_commands.webp
split-apply-combine.png
Unix_history-simple.svg
VCS_Diff-768x314.png

Stderr:
```

# Unix-like terminal commands

1. Getting Help
2. Navigation & Directory Listing
3. File & Directory Manipulation
4. Viewing File Contents
5. Searching, filtering and sorting
6. System Information & Monitoring
7. User & Permissions
8. Networking
9. Process Management
10. Archiving & Compressing

## Unix commands - Getting Help

### `man` - an interface to the system reference manuals

- `man command_name` : Display the manual page for a program, utility or function. Press `q` to quit.

In [14]:
```
#!man man
```

In [15]:
```
#!man 1 printf
```

In [16]:
```
#!man 3 printf
```

### `command_name --help` - try to get help from a command

Many commands have the option `--help` or `-h`

In [17]:
```
#!man --help
```

## Unix commands - Navigation & Directory Listing

## `ls` - list directory contents

- `ls` : List current directory contents.
- `ls img` : List the contents of the directory `img`.
- `ls -l` : List in long format (permissions, owner, size, date).
- `ls -a` : List all files, including hidden ones (starting with `.`).

In [18]:
```
!ls img/*.svg
#!ls -l ../Programming-for-AI/img/*.svg
#!ls -l /home/jupyter-mpenagaricano/Programming-for-AI/img/*.svg
```

```
img/Unix_history-simple.svg
```

Bash performs filename expansion (a process known as **globbing**) on unquoted command-line arguments:

- `*` → any character sequence (could be empty)
- `?` → any single character
- `^` → negating the sense of a match
- `[xy]` → single character from range `x` to `y`
- `{pattern1,pattern2,pattern3}` → any of the three patterns

For example:

- `ae*` : any filename starting with `ae`
- `[ab]*` : any filename starting with `a` or `b`
- `[a-k]*` : any filename starting with letters from `a` to `k`
- `[^ab]*` : any filename NOT starting with `a` or `b`
- `[^A-Z]?` : any filename of length two, not starting with uppercase

In [19]:
```
%%bash
ls img/[^a-z]*
```

```
img/Unix_history-simple.svg
img/VCS_Diff-768x314.png
```

## `pwd` - print name of current/working directory

## `cd` - change directory

- `cd dirname` : Change to the directory `dirname`.
- `cd ..` : Go up one directory level.
- `cd ~` or `cd` : Go to your home directory.
- `cd -` : Go to the previous directory you were in.

In [20]:
```
%%bash
pwd
cd /tmp
pwd
cd
pwd
```

```
/home/jupyter-mpenagaricano/Programming-for-AI
/tmp
/home/jupyter-mpenagaricano
```

## Unix commands - File & Directory Manipulation

### `cp` - copy files and directories

- `cp src_file dst_file` : Copy file.
- `cp src_file dst_dir/` : Copy file to destination dir.
- `cp -r src_dir dst_dir/` : Recursively copy source dir to destination dir.

There are many options to control metadata and symbolic links

In [21]:
```
#!man cp
```

### `mv` - move (rename) files

- `mv old_filename new_filename` : Rename file.
- `mv src_file dst_dir/` : Move file.

### `rm` - remove files or directories

- `rm filename` : Remove file.
- `rm -r dirname` : Recursively remove a directory (**USE WITH CAUTION!**).

### `mkdir` - make directories

### `rmdir` - remove empty directories

## Unix commands - Viewing File Contents

### `cat` - concatenate files and print on the standard output

### `more` & `less` - View file content page by page (scrolling & searching)

### `head` - output the first part of files

- `head filename` : Output the first 10 lines of the file.
- `head -n 20 filename` : Output the first 20 lines of the file.
- `head -n -20 filename` : Output all the lines of the file except the last 20.
- `head -c 123 filename` : Output the first 123 bytes (characters?) of the file.

### `tail` - output the last part of files

## Unix commands - Searching, sorting, filtering and transforming

### `grep pattern filename` - print lines that match patterns

### `find` - search recursively for files and directories (based on name, size, ...)

- `find . -name "*.txt"` : Find files ending in `.txt` within the current directory (recursively)

- `find /home/user -type d -name "my_project"` : Find a directory named `my_project` within `/home/user`

### `sort` - sort lines of text files

### `uniq` - report or omit repeated lines

### `tr` - translate, squeeze or delete characters

### `cut` - remove sections from each line

## Unix commands - Archiving & Compressing

### `tar` - an archiving utility (Create or extract archive files)

- `tar -cf archive.tar dirname` : Create an uncompressed archive from directory.
- `tar -xf archive.tar` : Extract an uncompressed archive in the current directory.
- `tar -czvf archive.tgz dirname` : Create a (GNU) zip archive from directory.
- `tar -xzf archive.tgz` : Extract a (GNU) zip archive in the current directory.

### `gzip` , `gunzip` and `zcat` - compress or expand files

- `gzip *` : Compress all files in current directory (rename to *.gz)
- `gunzip *.gz` : Uncompress all compressed files in current directory
- `zcat *.gz` : Print to the standard output the uncompressed content of files

## Unix commands - Networking

### `ping` - check network connectivity to a host

### `ssh username@host` - connect to a remote machine via Secure Shell

### `scp` - copy files between hosts over SSH

### `wget url` or `curl -O url` - download files from the web

## Unix commands - System Information & Monitoring

### `uname` - print system information

### `df` - report file system disk space usage

### `du` - estimate file space usage

### `ps` - snapshot of running processes

### `top` - display running processes (pid, user, CPU/memory usage)

## Unix commands - User & Permissions

### `su` - switch user

### `sudo` - execute a command as another user

### `whoami` - show the current effective username.

### `chmod` - change file mode (permissions - read, write, execute)

- `chmod +x script.sh` : Make `script.sh` executable.

### `chown` - change file owner and group

- `sudo chown user:group filename`

## Unix commands - Process Management

### `kill` - send a signal to a process

- `kill pid` or `kill -15 pid` : Send SIGTERM signal (*graceful* termination)
- `kill -9 pid` : Send SIGKILL signal (*ungraceful* termination)

### `killall` - kill processes by name

- `sudo killall -9 -u bob` : Terminate all processes owned by bob

# Command chaining

Commands can be chained. Every command will be executed based on the success or failure of the preceding command.

- When a command finishes executing, it returns an integer (*exit status*).
  - 0 → the command executed successfully
  - [1-255] → command failed
  - The variable `$?` contains the exit status of the last command

In [22]:
```
%%bash
ls 1b_unix_shell.ipynb
#ls this_file_does_not_exist
echo $?
```

```
1b_unix_shell.ipynb
0
```

## Logical AND ( `&&` )

- Syntax: `command1 && command2 && command3 && ...`

- Behaviour:
  - The shell executes `command1`
  - If `command1` succeeds, then the shell executes `command2`
  - If `command2` succeeds, then the shell executes `command3`
  - ...
  - If any command fails, none of the following ones will be executed

In [23]:
```
# Update package lists and then upgrade packages
#!sudo apt update && sudo apt upgrade -y
```

## Logical OR ( `||` )

- Syntax: `command1 || command2 || command3 && ...`
- Behaviour:
  - The shell executes `command1`
  - If `command1` fails, then the shell executes `command2`
  - If `command2` fails, then the shell executes `command3`
  - ...
  - If any command succeeds, none of the following ones will be executed

In [24]:
```
# Check if a file contains a specific pattern; if not, print a message
#!grep "ERROR" system.log || echo "No errors found in system.log"
```

## Combining `&&` and `||`

- Both can be chained together
- Precedence rules: *usually* `&&` tighter than `||`
- Use parentheses `( ... )` for clarity

In [25]:
```
# Check if a file contains a specific pattern; if not, print a message
#!make clean && make && ./run_tests || echo "Build or test process failed!"
#!((make clean && make) && ./run_tests) || echo "Build or test process failed!"
```

## ( `&&` and `||` ) vs `;`

- `command1 ; command2`
- Executes `command1` and then **always** executes `command2`

# Redirection and Pipes

Every command has three standard communication channels associated with it:

- **Standard Input** (stdin): This is where the command reads its input from. By default, it's connected to the keyboard.

- **Standard Output** (stdout): This is where the command writes its normal output. By default, it's connected to the terminal screen.

- **Standard Error** (stderr): This is where the command writes its error messages. By default, it's also connected to the terminal screen (but it's a separate stream from stdout).

Redirection is about **changing where these streams point to or from**, typically involving files (which could refer to devices).

## Output Redirection ( `>` and `>>` )

- `command > filename` : Redirects stdout of `command` to `filename` .
  - Overwrites: If `filename` exists, its contents are deleted and replaced.
  - Creates: If `filename` does not exist, it is created.
  - `cat file1 file2 > both_files`

- `command >> filename` : Redirects stdout of `command` to `filename` .
  - Appends: If `filename` exists, the output is added to the end of the file.
  - Creates: If `filename` does not exist, it is created.
  - `grep -B 1 -A 2 "^ERROR:" today_log >> all_errors`

## Input Redirection ( `<` )

- `command < filename` : Redirects stdin of `command` to come from `filename` .
  - `grep pattern < /dev/some_device`

## Error Redirection ( `2>` and `2>>` )

- `command 2> error_log.txt` : Redirects stderr to `error_log.txt` . Overwrites the file.
- `command 2>> error_log.txt` : Redirects stderr to `error_log.txt` . Appends to the file.
  - `find / -name secret.txt 2> errors.log` : Finds a file by name, putting permission errors etc. into `errors.log`

## Redirecting Both stdout and stderr

- `command > output.log 2>&1` : Redirects stdout to `output.log` , then redirects stderr ( `2>` ) to the current location of stdout ( `&1` ). Order matters.
- `command &> output.log` or `command >& output.log` : Bash shortcut to redirect both stdout and stderr. Overwrites.
- `command &>> output.log` : Bash shortcut to append both stdout and stderr.

## Pipes ( `|` )

- A pipe (pipeline) connects the stdout of one command with the stdin of another command.
- `command1 | command2` : stdout of `command1` is connected to stdin of `command2` .
  - Both commands run in parallel
  - Data doesn't need to be written to disk and read back in; it is buffered in memory between processes.
- `cmd1 | cmd2 | cmd3 | ...` : You can chain multiple commands together:

**Exercise:** Find the 15 most common words in the Bible and their number of occurrences.

1 - Download the bible from gutenberg.org (https://www.gutenberg.org/cache/epub/10/pg10.txt)

In [26]:
```
#!curl https://www.gutenberg.org/cache/epub/10/pg10.txt -o bible.txt
#!head -30 bible.txt
```

2 - Convert to lowercase and all non alphabet characters to whitespaces

In [27]:
```
#!cat bible.txt | tr [:upper:] [:lower:] | tr -c '[a-z] \n' ' ' | head
```

3 - Replace all whitespaces by newlines (one word per line)

In [28]:
```
#!cat bible.txt | tr [:upper:] [:lower:] | tr -c '[:alpha:]\n' ' ' | tr ' ' '\n' | head
```

4 - Sort the lines (words) alphabetically

In [29]:
```
#!cat bible.txt | tr [:upper:] [:lower:] | tr -c '[:alpha:]\n' ' ' | tr ' ' '\n' | sort | hea
#!cat bible.txt | tr [:upper:] [:lower:] | tr -c '[:alpha:]\n' ' ' | tr ' ' '\n' | sort | tai
```

5 - Retain a single occurrence and count them

In [30]:
```
#!cat bible.txt | tr [:upper:] [:lower:] | tr -c '[:alpha:]\n' ' ' | tr ' ' '\n' | \
#    sort | uniq -c | head
```

6 - Reverse sort the lines (words) numerically

In [31]:
```
#!cat bible.txt | tr [:upper:] [:lower:] | tr -c '[:alpha:]\n' ' ' | tr ' ' '\n' | \
#    sort | uniq -c | sort -n | tail
```

Single command line solution:

In [32]:
```
%%bash
curl https://www.gutenberg.org/cache/epub/10/pg10.txt 2> /dev/null | \
    tr [:upper:] [:lower:] | tr -c '[:alpha:]\n' ' ' | tr ' ' '\n' | \
    sort | uniq -c | sort -nr | \
    head -16 | tail -15
```

```
 64309 the
 51762 and
 34846 of
 13680 to
 12927 that
 12727 in
 10422 he
  9840 shall
  8997 unto
  8997 for
  8854 i
  8473 his
  8235 a
  7964 lord
  7378 they
```