**Big Data Batch Curriculum** -By: Peter Alagna

# **UNIX**

## **UNIX Family**

## **Open Source**

## **Linux File System**

### **EXT**

Before introducing UNIX famous **ext**ended file system, it would be great to mention a few of the ones that we already know:

* FAT32 (File Allocation Table ~ 4GB maximum file size ~ 2^32)
* NTFS (New Technology File System ~ Windows main file system)
* APFS (Apple’s file system)
* XFS (vs ext4 is a nice to know)

The purpose of a filesystem is to provide, at least, the following features:

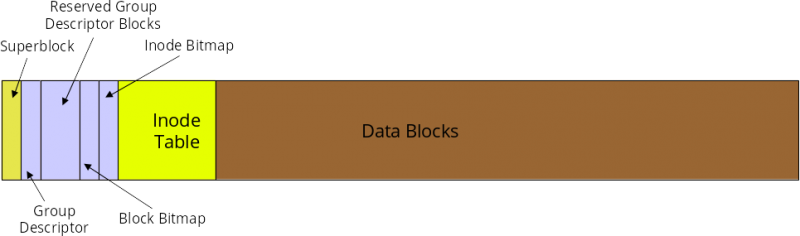
1. **Storage**. Of course, you want to store/retrieve files into/from the hard drive.
2. **Namespace**. Which maintains the metadata for all the files and directories in the tree (user who owns, permissions, file type, disk-blocks -> kept in **inode**)
3. **Security**. A way to provide access rights.
4. **API**. Ways to call the file system and manipulate data as files and directories.
5. **Implementation**. The software that implements the above (the different types of file systems).

However, the file system that UNIX uses is the **ext**. First written by **Remy Card** with Linux in 1992, it overcame the limitations of the **Minix** file system (written for educational purposes, **Linus Torvalds** didn’t want to write a file system so he leveraged this as the primitive version, written by **Andrew Tanenbaum**).

The first successful version was **ext2** which was used for many years in many Linux distributions. Major issue with this version was the fsck (file system check) that runs after a system failure during a file-update could take forever to run.

Version **ext3**, improved on this matter, using the same file structure as its predecessor, but adding a **file system journal**, which records in advance the changes that will be performed to the file system to improve recovery from failures.

(Optional) *Cylinder Structure, Journaling Types.*

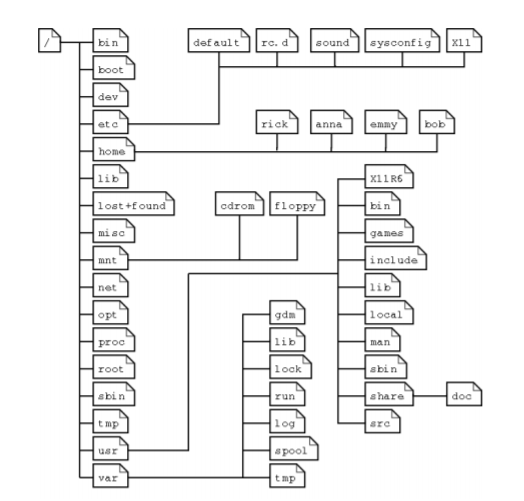


However, the current version is **ext4** which improves almost every detail of its predecessor like unlimited directory amount, bigger file size support, journal check-summing to avoid issues with devices which have their own cache, and more.

To create a file system, look into the ***mkfs*** command.

### **Directory Structure**

It’s very important to understand the standard directory structure of a UNIX based system, and most importantly, understand the difference between **root** (/), **home** (~) and other different folders.



* **/ :** Is where the file system begins and where all important operating system directory structure is located (like C:/)
* **~** : It’s a shortcut to specifically go to the **current logged user** home. Which is equivalent to /home/username or /root if you are logged in as root (like C:/Users/username)
* **boot**: contains all grub configurations for booting the operating system, kernels, and more.
* **bin:** holds binary files (executable commands) for both users (root privileged, and not) and system boot commands (like System32).
* **sbin:** system essential binaries, like fsck, mkfs, and more.
* **var:** contains variable files, like logs, emails and more. (/var/log \*\*)
* **usr:** stands for UNIX system resources, it contains majority of utilities and user programs (like Program Files)
* **tmp:** files that are not meant to be permanent (you can put installers here, or other files that you want to delete after using). The operating system also puts temporary files in this folder. Files here are usually not kept after system reboot.
* *Investigate the rest.*

### **Manual**

Before starting with Linux commands, it’s important to highlight to know of the existence of the **man** command. This is technically your best friend, since it usually contains all needed documentation for any command.

Running **--help** on almost any command also gives you a brief explanation of the usage. To search through manuals, use **man -k** <*term>*or while within a manual, hit **h** to show all available options (/pattern is how you search)

### **File and Directories Commands**

* **pwd** - prints current directory
* **cd** <*dirPath>* - change to directory (. , .., /, ~ or nothing, directoryPath)
* **ls** <*dirPath>* - list all files and directories (.files are hidden and can be display with -a flag)
* **mkdir** <*dirPath>* - create a directory
* **touch** <filename> - create blank file
* **rm** <*dirOrFilePath>* - remove files or directories (-R for directories, *be careful*)
* **cp** <*dirOrFilePath*> <*dirOrFilePathDest*> - copy file or directories (-R for directories)
* **mv** <*dirOrFilePath*> <*dirOrFilePathDest*> - move file or directories
* **du** -h <dirPath> - show directory size in human readable format

### **Query Commands**

* **cat** - print contents of a file
* **head** - print first 10 lines of a file (-n for different count)
* **tail** - print last 10 lines of a file (-n for different count)
* **sort** - sort lines in ascending order (-r for reverse)
* **tac** - prints contents of a file in reverse order
* **wc** - prints the amount of lines, words and bytes of file(s) in that order

### **Expression Query Commands**

* **grep (G**lobally search a **R**egular expression and **P**rint)
  + Very important command to perform queries on files.
  + One of the best tools to troubleshoot in UNIX.
  + You can also pipe the output of one command to grep.

*From Ryan Lessley:*

|  |  |
| --- | --- |
| **OPTIONS** | |
| -i | ignore case |
| -v | Return lines that don’t match…. !grep |
| -w | only whole word matches |
| -c | Count of matching lines |
| -l | List names of files which contain matches |
| -n | Show line number of each matching line |
| **REGULAR EXPRESSIONS** | |
| . (period) | Single character |
| [abc] | Any ONE of these characters |
| [^abc] | NOT one of the characters within |
| (abc) | Group characters and remember for later |
| \n | Where ‘n’ is a number, recall characters matched in that set of brackets |
| | | Logical or |
| \ | Escape character |
| **MULTIPLIERS** | |
| ? | Preceding item is matched 0 or 1 times (IE optional for ONE character) |
| \* | Preceding item will be matched 0 or MORE times |
| + | Preceding item will be matched ONE or MORE times |
| {n} | Preceding item matched n times specifically |
| {n,} | Preceding item matched n or more times |
| {n,m} | Preceding item matched n to m times. |
| **ANCHORS** | |
| ^ | Beginning of line |
| $ | End of the line |
| \< | Beginning of word |
| \> | End of the word |
| \b | Match either beginning or end of word |
| **QUICK N DIRTY EXAMPLES** | |
| egrep 'mellon' myfile.txt print every line in myfile contaiing the string mellon | |
| egrep '(.)bb\1' myfile.txt Fine every line with 2 b’s and the same character both before and after those b’s | |
| egrep -l '[0-9]{8,}' /files/projectx/\*  |print each file in projectx directory with number of 8 digits or more | |
| egrep '\b[a-z0-9.\_%+-][+@[a-z0-9.-]+\.[a-z]{2,4}\b' myfile.txt](mailto:+@[a-z0-9.-]+\.%5ba-z%5d%7b2,4%7d\b'%20myfile.txt)  |Find emails in file | |

* **egrep**
  + Uses **extended regular expression.**
    - For example, in **grep** the | character it’s a regular character, in egrep it’s an OR operand.
    - You can produce the same outcome with **grep** by escaping the character with **\**.
  + Deprecated to use separately, use **grep -E** instead.
* **fgrep**
  + Regular expressions are not used, therefore they are considered regular characters.
  + This can come in handy when using a file as a word bank (-f) and it contains characters that are considered expressions for **grep** or **egrep**.
  + It is also deprecated to use separately, use **grep -F** instead.

**Commands Alteration**

* **>** - redirect stdout to file
* **>>** - append stdout to file
* **2>** - append stderr to file
* **&** - run command as background process
  + *sudo du -h /\* >> du.txt 2> error.txt &*
* **\*** -wildcard, if used with directories it means *all.*
* **|** - redirect output of command of the right to the left \*\*

## **Package Management**

There are different ways to install software in UNIX environments, it all depends on which distribution.

### **rpm**

This package manager is widely available in majority of Linux distributions, and most importantly in RedHat (the acronym means RedHat Package Manager).

This is used to automatically install .rpm files in the operating system, which usually affects /usr/\* folders.

* To install a package run **rpm -ivh packageLocation**.
* To check the package content (where are files installed) run **rpm -ql packageName**.

### **yum**

Yellowdog Updater, Modified, it’s a package manager available in many Linux distributions. It leverages rpm to automatically install/update applications.

This is what you would normally use since it abstracts the usage of rpm and some automatic configurations.

The OS needs to be configured and plugged in into a yum repository, which would contain all .rpm files for different applications. Each operating system has their own yum repositories.

* To install an application run **yum install applicationName**
* To update an application run **yum update applicationName**
* To update all installed applications run **yum update**

### **apt-get**

Similar to yum, Advanced Packaging Tool is another package manager. It is not available in all Linux distributions, but it’s available in *Ubuntu* (the most widely used Linux distribution) for example.

* To install an application run **apt-get install applicationName**
* To update an application run **apt-get update applicationName**
* To update all installed applications run **apt-get update**

## **Environment Variables**

As a system administrator it’s important to know how to manage system environment variables. This way, applications or uses can programmatically access this variables, which normally contain folder locations. If in the future that variable changes, all applications that were accessing it don’t need to be updated.

* In majority of Linux distributions, environment variables are available in **~/.bashrc**, which is a text file.
* To add an environment variable, write **export VARIABLE\_NAME=**
* After the changes are saved, you can refresh the variables with **source .bashrc**
* To access an environment you can use **$**VARIABLE\_NAME from anywhere in the terminal.
  + It also supports **auto-complete** features, so if you **tab**, it completes the variable name (this also gives you a hint that you set it up properly).

## **File Editors**

As administrators, developers and/or scripting, knowing how to use Linux file editors is widely important. There is vim and there is nano (I call it the i-don’t-know-vi-editor).

### **vim**

The most famous Linux text editor, Visual Instrument iMproved, available in technically all Linux distributions, it’s a different kind of editor that is uniquely used in these environments. It posses three (3) modes: Command, Insert, and Line modes.

The improved version it’s a superset of its predecessor, which contains features like: syntax highlighting for popular programming languages (C++, Python, Java, and more), comparing two files with **vimdiff**, split files into multiple screens, multilevel undo/redo, and more.

Almost every distribution at this point has **vim** as the default editor, and even if you do **vi** it will still show you its improved version (or man vi). You don’t mind this, since vim supports everything vi does.

*From Ryan Lessley (enhanced):*

|  |  |
| --- | --- |
| **COMMAND MODE (hit escape)** | |
| k, up-arrow | Up a line |
| j, down-arrow | Down a line |
| h, left-arrow | Left a character |
| l, right-arrow | Right a character |
| w | Right a word |
| b | Left a word |
| ^ | Beginning of line |
| $ | End of a line |
| x | Delete a char |
| dw | Delete a word |
| ndw | Delete n words |
| dd | Delete a line |
| D | Delete everything at and after current position on current line. |
| r | Replace current character |
| cw | Change current word |
| cc | Change current line |
| c$ | Change text from current pos to end of line |
| C | Same as c$ |
| ~ | Reverse casing of character |
| yy | Yank (Copy) current line |
| y<pos> | Copy where pos = w for word or l for letter |
| p | Paste it |
| u | Undo |
| ctrl-R | Redo |
| /<pattern> | Find word past cursor |
| ?<pattern> | Find word before cursor |
| n | Find next occurrence of search word |
| N | Find previous occurrence of search word |
| ctrl-w -> hjkl or directional arrows | Move to a different split window |
| **INSERT MODE (hit any of below keys while in command mode)** | |
| i | Insert mode at position |
| I | Insert mode at beginning of line |
| a | Append after the cursor position |
| A | Append at the end of line |
| **LINE MODE (while in command mode, type ‘:’)** | |
| :w | Save file |
| :w! | Force file save |
| :q | Quit |
| :q! | Quit without saving |
| :wq! | Save and quit |
| :x | Same as :wq |
| :n | Go to line n |
| :$ | Go to EOF |
| :set nu | Turn on line numbers |
| :set nonu | Turn off line numbers |
| :help [command] | All available functionality in vim |
| :sp [pathToFile] | Split two file views horizontally |
| :vsp [pathToFile] | Split two file views vertically |

## **User Management**

There three important sections within user management in UNIX environments: Users, Groups, and Permissions.

### **Users**

Users can be people, programs or any entity that can use the operating system and its capabilities, following these properties:

* They may or may not be part of the **super users** (sudoers).
* They may or may not have **access** to certain files or directories.
* They must form part of a **group** (if they are not part of a group, they are a group themselves)

#### **Read**

There really isn’t any command to display users information in Linux. However, their information it’s located in **/etc/passwd**. This file content has data delimited with “:”, so we can use **grep** in combination with **cut** to perform queries.

The **passwd** file contains the following structure for users, separated by a colon:

1. USERNAME
2. A symbol representing that the password is encrypted (usually x or !)
3. UID (User ID)
4. GID (Group ID)
5. GECOS (Information of User: Full name, Building or room, Office number, Home number, Contact info [email for example])
   1. All this delimited by commas
6. Home directory (usually /home/username)
7. Login shell (usually /bin/bash)

To query all users from the **passwd** file:

1. Display all actual users (/home): **cat /etc/passwd | grep ‘/home’ | cut -d: -f1**
2. Display all users: **cat /etc/passwd | cut -d: -f1**
3. Display connected users: **who**
4. Display current user: **whoami**

As an additional, user passwords are stored salted and encrypted in **/etc/shadow**. The encryption algorithm and some configuration can be altered, but this changes between different distributions, so it’s better to leave it as default.

#### **Create**

1. To create the user: **useradd <username> -c <fullName>**
   1. You can use the same flags as **usermod** to add everything at once.
   2. Users created without a password are disabled by default.
2. To add a passwd to the user: **passwd <username>**
   1. Force them to change password on login: **chage -d 0 <username>**
3. To switch to a different user (substitute user), if run with no arguments, we are root: **su <username>**
4. Execute a command as a different user (with no flags it’s root): **sudo -u <username> <command>**

#### **Update**

1. To update his comment (full name): **usermod -c <fullName> <username>**
2. To modify the home directory: **usermod -d <dirPath> <username>**
3. To change the primary group: **usermod -g <groupName> <username>**
   1. If added to **wheel** (masquerade standard for super users in a lot of UNIX systems) it will become a **sudoer** (super user).
   2. If added to **himself**, it will become a regular user.
4. To lock an account (prevents user from logging in): **usermod -L <username>**
5. To unlock an account: **usermod -U <username>**
6. To remove password from user: **passwd -d <username>**

#### **Delete**

1. To remove a user and its home directory: **userdel -r <username>**
   1. If that user is a super user, user will be deleted but not the group.

### **Groups**

Users in UNIX systems can be part of groups. They can have a **primary group** and a set of **secondary groups**. Group information is stored in **/etc/group**.

The **group** file contains the following structure, separated by a colon:

1. Group Name
2. A symbol representing that the password is encrypted (usually x or !)
   1. Groups can have passwords, but they are not widely or at all used in UNIX systems (this bring security issues, everyone has to know one password).
   2. To add a password to a group you can use gpasswd.
3. GID (Group ID)
4. Group members separated by commas.
   1. This field is not used most of the times and it’s empty.
   2. If it’s empty, it doesn’t mean that the group has no members. To check members of a group, use the lid command, which will check the GID field on /etc/passwd.

Groups are used to categorize users in different sections, so it is easier for administrators to know what a specific user should be doing, or in which part of the organization is that user at. Also, and most importantly, when we get to permissions, files and directories can have specific permissions for a group.

#### **Read**

1. To display all users in a group: **lid -g <groupName>**
2. To display in which group is a user at: **lid <username>**
3. To display all groups: **cat /etc/group | cut -d: -f1**
   1. Sometimes members are attached

#### **Create**

1. To create a group: **groupadd <groupName>**
   1. Users with this group as their primary group, will allow them to have the file or directory privileges. If they create a file or directory, the group that owns the file will be the primary group of this user.

#### **Update**

1. To modify a group name: **groupmod -n <newName> <groupName>**
   1. This will alter everything about this group and will not create issues.

#### **Delete**

1. To delete a group: **groupdel <groupName>**

### **Permissions**

#### **chmod**

Used to manage permissions of files or directories. It uses a particular structure that follows the following structure: **user**, **group**, and **others** permissions (ugo), and the ability to the fiend read (r), write (w), and/or execute (x) capabilities to each section.

The structure of the command follows: **chmod <permissions> [options] <file or directory>**. You can recursively change for all files within a directory with **-R**.

The permissions have the following structure, representing binary numbers of three digits (with three binaries with can make till 7 combinations):

* 4 for **read** (100)
* 2 for **write** (010)
* 1 for **execute** (001)
* For example, read and execute (r\_x) would be (101 -> 5)

This permissions has to be set three times, to specify the permissions for each section, for example:

* **chmod 750 file** -> Users can do everything, group members can only read and execute, and others can’t do anything.
* **chmod 777 file** -> The file is technically publicly accessible to anyone that has access to the machine.

It is important to notice that **sudoers** don’t mind this permissions, that’s why it is important to know what you are doing as a system administrator.

#### **chown**

In UNIX systems, files and/or directories are owned by users and groups. Ownership of files is very important, since it’s related to the permissions available in chmod (ugo).

By default, if you create a file with a specific user, the ownership into the new file or directory is inherited from the user (e.g. peter.alagna from the revature group, root from the root group [for sudo this happens]).

To change the ownership of a file or directory:

1. chown username:groupName fileOrDir
2. chown username fileOrDir
3. chown :groupName fileOrDir
4. Recursively for all files within a directory: -R

# **Processors**

The central processing unit (CPU) is the place where all the magic happens in the operating system, and is the most valuable resource used by **processes**.

### **Kernel**

The interface between the software and the hardware. It controls all of the resources available in the operating system and assigns them to **processes** when requested. Examples of resources are: **CPU**, **Memory, Hard Disks, External Peripherals, etc.**

Technically when we execute user commands, we are consulting the Kernel about hardware information, but we are never interacting with the hardware directly.

### **Execution Workflow**

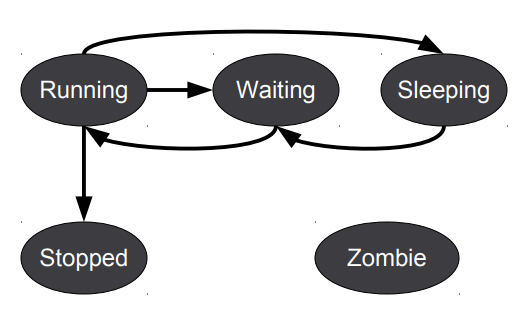
Processes exist in the operating system as long as they still have **instructions** that need to be executed.

In a **single** **processor** machine, current operating systems are able to **multitask** with certain mechanisms. In a **multi-processor** machine, the operating systems are actually able to run processes in parallel. This cannot be perceived by the human eye, since the execution of instructions happens in the order of **nanoseconds**.

#### **Control Block**

Each process contains its own PCB, which is information about the process like: the current **program counter**, the **stack pointer**, the **process state**, and more.

#### **States**



1. **Running**. The CPU is currently executing instructions of a process.
2. **Waiting**. A process is ready for execution.
3. **Sleeping**. A process is ready for execution, but has logic that is making him wait for a certain amount of time or it’s waiting for another process to awaken it.
4. **Stopped**. A process has ended its execution.
5. **Zombie**. A process has ended its execution, but still had instructions that needed to be run. You can’t kill a **Zombie** process, since it’s already dead.

#### **PC**

One of the **CPU Registers** stored in the PCB, contains the location in memory of the next instruction that needs to be executed. Once it is executed, the CPU increments it’s amount **by 1.**

#### **SP**

Marks where a process *has been*. It is used to achieve sub-routine calls and/or recursive calls in processes. Some processes will push/pop values to this stack, which will alter the current **PC**.

### **Scheduling**

The scheduling information of a process it’s also stored in the PCB. The Scheduler itself it’s a process, and it interrupts processes that are currently executing instructions in the processor depending on a specific strategy.

Every time a process is **interrupted**, there is a **Context Switch**. This is why we need a **PCB**, in order to be able to know what to execute next.

Some of the different approaches used for scheduling are: FIFO (no scheduling), Priority Queues (covered later), and Quantum Time (mostly used).

The **Quantum Time**, also known as **time slice**, it’s calculated by the **Scheduler** and stored in the PCB. This time is used by the **Scheduler** to decide which process should execute next, based on its remaining executing time. Processes are interrupted when their quantum time is done, to allow the **Kernel** to switch between processes. E.g.: If a process has a lot of remaining time, this process will have priority on the process queue (let’s get rid of the hardest job first).

### **Priority**

Process priorities are normally controlled by the operating system. It can also be controlled by specific programs (not recommended since the OS comes first).

Usually, priority of processes is decided based on their QT (round-robin time slice) by default or they will all have the same priority (normal time schedule). However, OS specific process will always have priority over regular processes, and are usually the ones that constantly interrupt them (e.g.: keyboard stroke, mouse click, mouse move, etc.).

You can manually change the priority of processes, however, this is not recommended because it might create one of the renowned issues of processing, known as Starvation.

* **SCHED\_FIFO\***: first in, first out processes
* **SCHED\_RR\***: round-robin processes. These represent the **QT** strategy
* **SCHED\_OTHER\***: normal time schedule (equitative), a lot of processes use this
* **SCHED\_BATCH**: similar to **3**, but this process consumes more CPU, not available in all distributions

### **Issues**

#### **Deadlock**

Process A is holding a resource that Process B needs but won’t release it until Process B releases the resource it’s holding that Process A needs.

Deadlocks have no solution as of now, process permanently stay in the **Sleeping** state,and the only way of solving it is to kill one of the processes, or both.

#### **Starvation**

Usually occurs when priority strategies are used poorly. This is why we want to let the operating system decide when to execute a process by its own means.

### **Commands**

#### **top**

Displays current processes which are executing *live****.***

#### **lscpu**

Displays detailed information about the processor.

#### **chrt**

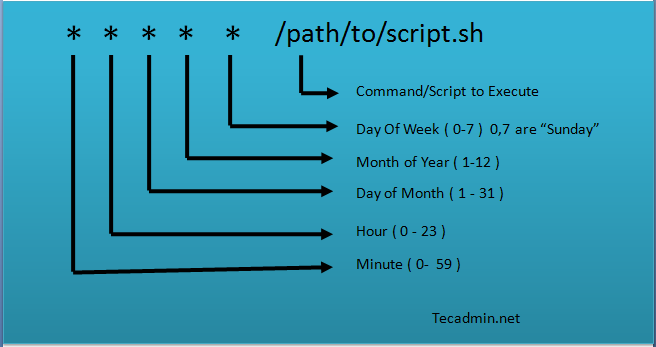
Displays real time information about the priority of a specific process. This is the information that usually displays as specified in the Priority section.

* **SCHED\_FIFO\***: **chrt -f -p 1-99 PID**
* **SCHED\_RR\***: **chrt -r -p 1-99 PID**
* **SCHED\_OTHER\***: **chrt -o -p 0 PID**

To display the range number of priorities: **chrt -m**

#### **crontab**

Use to schedule process based on a specified **Cron Time**.



1. To check the current crontab for current user: **crontab -l** or **crontab -u username –l**
2. To edit the crontab with vim: **crontab -e**

Each line of the crontab file will represent a different job, as soon as the file is changed, they will be available immediately.

#### **watch**

Execute and watch the output of a process every certain amount of time. Good usage for **monitoring**.

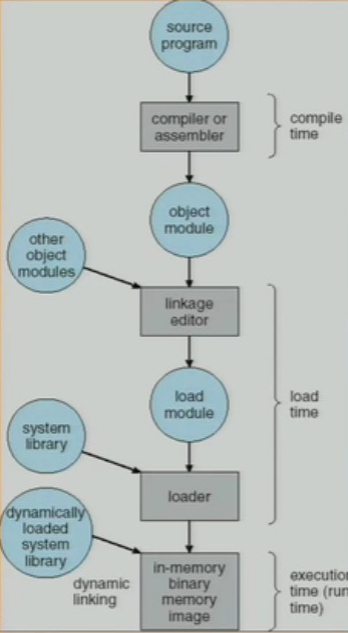
#### **kill**

<PID> kill a process. -9 obliterates it (forceful quit).

# **RAM**

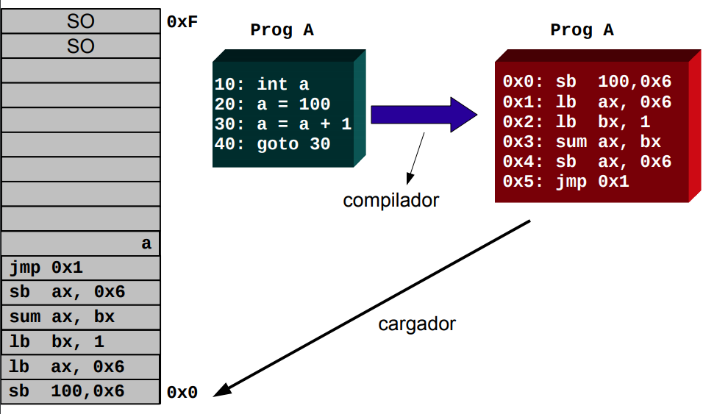
Random Access Memory, is almost as important as the Processor, since without it, it couldn’t do anything. All processes use memory, some more than others, and running out of memory could be a grave issue for any operating system.

### **Storage Workflow**



1. The **source program** (.java, .py, .cpp, etc.) is compiled.
2. The compiler generates an **object module** (executable file).
3. The executable file is **linked** to external or system libraries.
4. The program is **loaded** into memory.
5. Update **PC** of new process with the first line of code.
6. The process is ready to **run**.

In the following image, the simplified process can be seen.

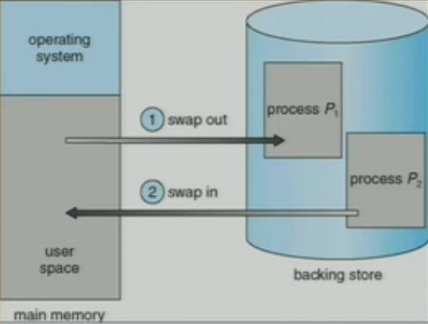


### **Swap**

Since memory is a very valuable resource, we need to leverage different strategies if we run out of it. **Swapping** is one of them.

When properly configured, the operating system is capable of storing a process memory usage into a specific place in the **hard drive**. When the process is ready to run again, its context will go back to **memory**.

This process my slow-down the machine performance, but it will keep it running without fatal errors.



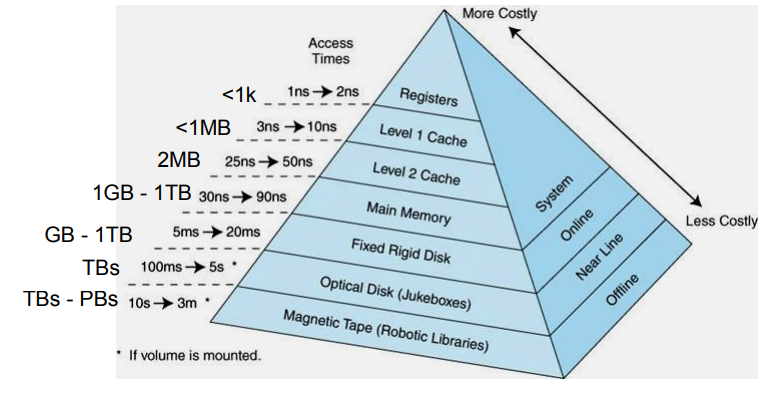
### **Cache**

Similar to memory, it is used to speed up the process of retrieving data. We are going to talk about **physical** **caches**, since some software and operating system use **virtual caches** (similar concepts apply nonetheless).

The size of these memories is way lower than actual memory, but the access time is way faster. The processor is capable of levering these smaller memories to access data that *is being used frequently*.

This rises to major concepts:

1. **Temporal Locality**: most recent used data has high chances of being used again.
2. **Spatial Locality**: adjacent pieces of a memory element have a high chance of being used (bring everything).



What happens if data stays to long in caches? That’s why there has to be different **page** **replacement algorithms**. In simple words, these are the available algorithms:

1. **FIFO\***: The page that has been in memory the longest is replaced.
2. **MIN**: The page that will not be used in the most time is replaced.
   1. Wait! We can’t see the future.
   2. It’s used to compare it to other algorithms since this one has the best performance.
3. **RANDOM**: Random pages are removed. Produces unpredictable times.
4. **LRU\***: Least Recently Used pages are removed. It needs a lot of instructions to implement. *Read about LRU Approximation*, which is the usual implementation.

### **Commands**

#### **free**

Simplest output of memory information.

#### **vmstat**

Displays more detailed information about memory usage, IO, processes and CPU. By default, it displays values as follows (for a more verbose output use –s):

1. Procs – r: Total number of processes waiting to run
2. Procs – b: Total number of busy processes
3. Memory – swpd: Swapped virtual memory
4. Memory – free: Free virtual memory
5. Memory – buff: Memory used as buffers
6. Memory – cache: Memory used as cache (virtual cache)
7. Swap – si: Memory swapped from disk (for every second)
8. Swap – so: Memory swapped to disk (for every second)
9. IO – bi: Blocks in. i.e blocks received from device (for every second)
10. IO – bo: Blocks out. i.e blocks sent to the device (for every second)
11. System – in: Interrupts per second
12. System – cs: Context switches
13. CPU – us, sy, id, wa, st: CPU user time, system time, idle time, wait time

#### **fallocate**

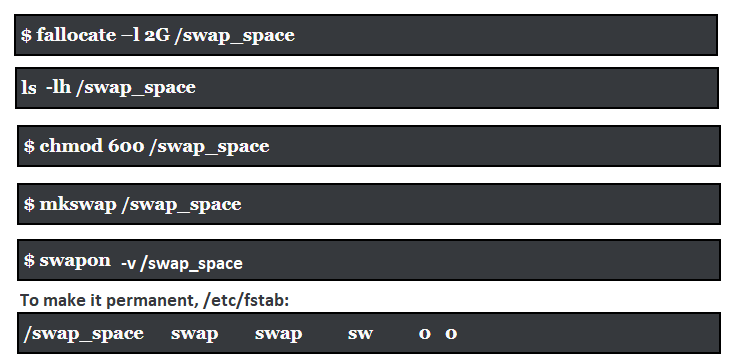
Pre-allocate space to a file, this can be used to create swap areas.

#### **mkswap**

Format/setup a Linux swap area. This can be used after using **fallocate**.

#### **swapon, swapoff**

This can be used to turn on/off a specific swap area after formatting it with **mkswap**.



# **Storage**

A machine with a processor and memory might sound as more than enough. However, what happens if we turn it off? To make our changes persistent, we need storage.

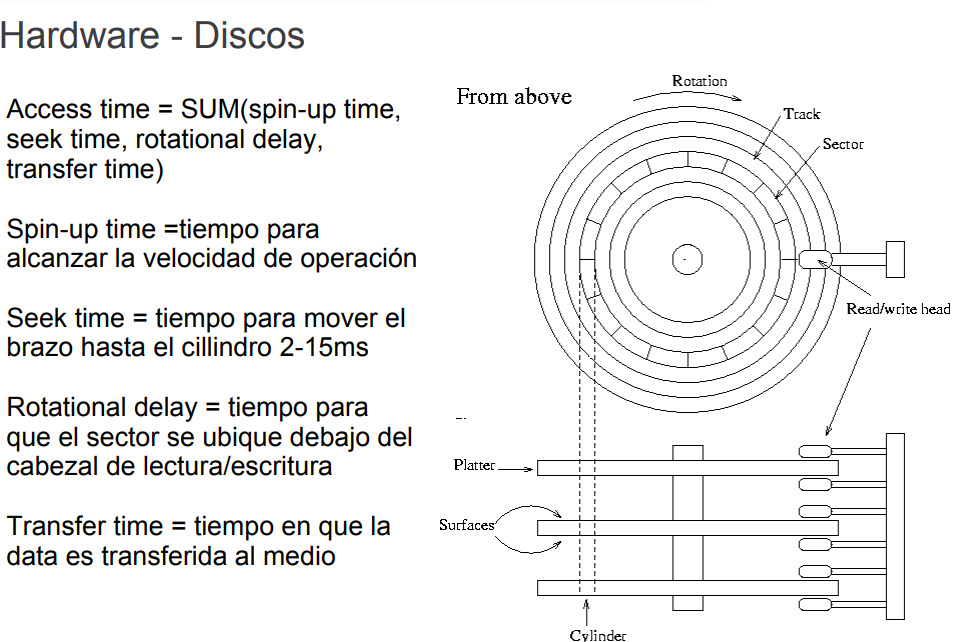
### **Types**

#### **Magnetic**

Also known as Hard Disk Drive (HDD), or **mechanical drive**, is one of the first ever built storage drives. The first ever hard drive was released by IBM in 1956, it was able to hold 5MB, it costed 50000$ and it was of the size of two refrigerators. Nowadays we are in the order of TB drives, and they don’t cost that much money at all, they are actually one of the cheapest parts of any machine.

It’s all about physics. The rotatory disks generate an **electric field** on top of the disk. Then there is a mechanical arm, which generates **magnetic** **fields** to direct the electric fields to a specific direction, *going down is* ***positive*** *(0), and going up is* ***negative*** *(1)*.

To measure HDD performance, the following criteria has to be followed:



Since the drive is mechanical, the access time is extremely lower than memory access or cache access.

* The average access time of modern HDD is between **9~15ms**
* Transfer rates of up to 120MB/s ~ 500MB/s depending on **rpm**, without counting mechanics, overhead, heat, and magnetism.
* Suffers from **Fragmentation** and **Magnetism.**

#### **Solid State**

Invented 40 years ago (whether you believe it or not), these drives came afterwards and were very expensive at its beginnings (like any tech, including HDDs). Technically, the follow the same concept as RAM, but the data actually stays after a **reboot**. That’s why it’s called solid state, since it doesn’t have any mechanical moving parts, it’s technically a **brick**.

Without getting too much into physics, DRAM transistors need to be refreshed every certain amount of time. SSD transistors keep their charge even after being powered off. *You can read more about the physics if you are into electronics.*

* 35 ~ 100 microseconds (0.035ms ~ 0.1 ms) access time. That means that in the worse case, SSDs are at least **100 times faster** in access times.
* Transfer rates 500MB/s ~ 800MB/s
* Doesn’t suffer from **Fragmentation** nor **Magnetism**.

### **Partitions**

Let’s say you don’t have access to get **multiple** **drives** or you are running a **low cost** server, but you still need to **separate** your **data** in certain ways depending on your business or needs, why not **partitioning** your disk?

We can create partitions to modularize our file system into more granular pieces, an example of some partitions in a hard drive: **boot** (OS), **swap** (optimization), **backup** (recovery), **applications**, **data**, **etc. *Partitions act as logical devices.***

Some benefits of using partitions:

* Of course, your hard drive is organizing data very well if partitioned properly. You are **modularizing concerns** and using pieces of your drive with a single **responsibility**.
* Easier to **recover from fatals**, if you have a **backup** partition. You didn’t have a backup? Your whole hard drive **didn’t get compromised**, so you still have some of the data left. The OS got **damaged**? You still have your **data**.
  + This is why usually UNIX distributions use partitions by default and encourage you to do more for different purposes.

**Partitions** are still considered physical (pieces of the drive in a logical way). **Logical** **Volumes** can be a confusing term, but it’s different. A LV needs formatting, a partition does not, which means LV is related to file systems.

### **Arrays**

The opposite of partitions. You have a lot of similar drives, you want to make them one with bigger storage and increase your performance (optimizing, again). That’s why they are called RAID (Redundant Array of Independent Disks).

The performance of arrays increases because reads and writes can be run in **parallel**, however, some of the different approaches have issues, but there are solutions.

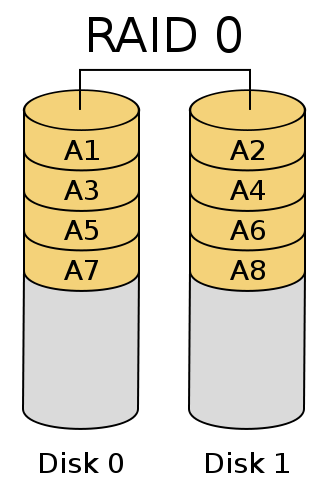
Before getting into the types of RAID, there are two very important concepts in these kinds of arrays, which are **Parity** and **Mirroring**.

Some RAID levels use **Parity** for fault tolerance. It’s technically a strategy to recover from write failures (in parallelism, fatals sometimes are unevatible). It increases **fault** **tolerance**, but it **decreases** performance.

Some other RAID levels use **Mirroring**, which is technically **replication**. This also increases **fault tolerance**, but increases the cost of your architecture (more hard drives are needed)

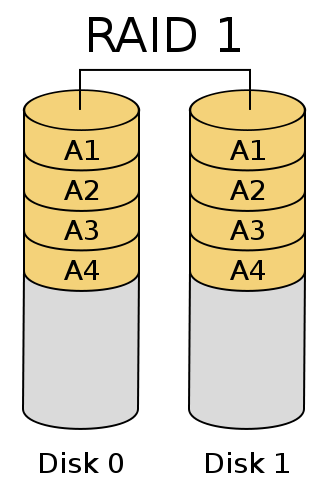
#### **RAID 0**

* The monster of RAIDs with no mirroring, nor parity.
* Highest performance, but has **zero** fault tolerance.
* You can use this when performance is your primary concern and lost of data is not an issue.



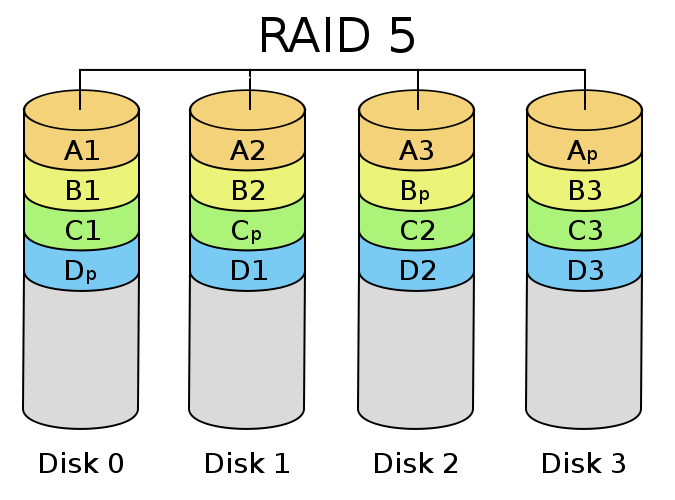
#### **RAID 1**

* No parity, full mirroring.
* Replicates every write of main disks.
* Leveraged to provide real-time backups for disks.



#### **RAID 5**

* Parity data distributed into all arrays, no mirroring.
* Provides the most fault tolerance, but needs more disks for the parity data.



#### **Combinations**

* Best performance and good safety, **RAID 0+1** (lacking fault tolerance)
* Safest and most fault tolerance solution, **RAID 5+1**.

### **Issues**

#### **Fragmentation**

For both Memory and Hard drives, fragmentation is an issue. *Read about details on different solutions for both sides*. For memory is not the biggest issue anymore as back in the days.

### **Commands**

#### **df**

Report filesystem disk space usage.

#### **fdisk, sfdisk, cfdisk**

Create partition tables in different particular ways. They can all produce similar outcomes but receive different parameters.

#### **lsblk**

List all mounted blocks in the machine. This will include swap areas or any temporary drives like a CD or a USB drive. It also includes the sizes in human readable format.

#### **mdadm**

Manage multiple drive (MD) devices. The Linux application to configure RAID.

* **-E** <drive> examine drive state (it can show if it’s already on a RAID)
* **-C** <array-name>
* **-l** <level>
* **--detail** <array-name>

# **Virtualization**

<From your head: HOST vs Guest>

### **Hypervisor**

Also known as Virtual Machine Monitor (VMM), it’s a process, firmware or hardware that creates, runs and manages virtual machines. It’s in charge of managing resources between the HOST and the GUEST.

There are two types:

* Type1: Known as **bare-metal** hypervisors, they run directly on the HOST hardware (the HOST OS is aware of the virtualization). Examples: **Xen**, **KVM**, **Microsoft Hyper-V**, and more.
* Type2: Known as **hosted** hypervisors, they run as a regular application, abstracting the virtualization from the operating system. Examples: **VirtualBox**, **VMWare,** and more.

*Xen vs KVM is important in 2018 for AWS.*