## Basics

The job of the operating system is to create **good abstractions** and then implement and **manage** the abstract objects. The Operating System is as a **Resource Manager.**

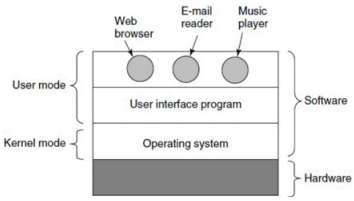
There are many OS variants: Mainframe Server, Multiprocessor, Personal Computer, Embedded, Sensor Node, etc.
Interacting with an OS: Terminal or Graphical User Interface

### Operating System vs Distributions

**Operating System** = Kernel, Core Component (engine) of an OS.

**Distribution** = Specific version of the Kernel, e.g. Linux Distributions. OS includes user interface, system utilities, drivers, etc.

### Kernel and User Mode




OS offers **access to HW** via **System Calls.**
→ **trap** command (system call) goes from user to kerne, mode.

```
strace # trace system calls and signals
```

**Signals**
Well defined signals (e.g. SIGTERM) allow to alert/inform a running process.

```
kill -9 PID # temrinate a running process
```

### OS Concepts

**Core:** Individual processing unit inside a CPU. Executes instructions. CPUs can have multiple cores, allowing for multithreading.

**Program:** A compiled executable (binary including a set of CPU instructions) for a target platform. File made of CPU-instructions (memory map)

**Process (task):** Program instance, loaded into memory selected for execution by Scheduler, executed by the CPU. Run in **user mode** but may **trap into kernel mode** for OS services (= system call). Each process resides witih an **environment** with its on virtual memory. Has an **owner with priviliges.**

**Scheduling:** Deciding which process to execute (CPU allocation)

**Multi-Tasking:** More than one process in execution

**User Mode (User Space):** read/write variables in RAM (Application Logic and data manipulation)

**System Mode (Kernel Space):** system and hardware management Instructions (I/O)

**Mode Switch:** Trap into kernel mode, e.g. when a user process calls a system call or an ISR (Interrupt Service Routine) occurs. Same process, therefore no scheduler involved.

**Context-Switch:** Change of process, invoked by scheduler, going from one process execution to another. CPU needs to remember where it was.

### Modules

A modular OS systems allow to extend the functionality of an OS by modules (Microkernels). Linux is a modular OS, not a monolith.

```
lsmod  # list all modules
insmod # insert module into running kernel
```

## Booting

When a OS is booted, all services (keyboard, screen, Memory, etc.) need to be started first.

### Step 1: BIOS (Basic Input/Output System)

BIOS is a type of firmware that initializes and tests hardware components when a computer is powered on, and it loads the operating system from storage into memory. The kernel (a computer program at the core of the OS) then takes over and becomes the manager of all things.

1. **Device discovery:** Scanning PCI buses for deviced to initialize
2. **Bootloader:** A boot device (usb, rom, disk) from list in CMOS is chosen, read into memory and executed. The **MBR** code **(Master Boot Record)** selects and loads a Bootloader (from a boot device) to be executed.

**Limitations:** Operates in 16-bit model, not designed for extendability, security issues (rootkit and bootkit attacks).
**UEFI** (and GUID) has replaced BIOS (and MBR) on most modern systems

### Step 2: Bootloader

The Bootloader, e.g. GRUB (GRand Unified Bootloader) accesses the location of the OS (boot partition), loads it into memory and executes it (sets registers and jumps to first instruction).

Problem: The not yet loaded OS needs access to the hard drive and its file system. The software needed to read the disk is stored on that disk itself.

Solution:

1. Bootloader (e.g. GRUB) loads the kernel into RAM.
2. Bootloader (GRUB) loads the **Initial RAM Disk Image** (RAMDISK, a **temporary root filesystem**) which contains necessary drivers and tools for the kernel.
3. Bootloader sets registers and jumps to first instruction (entry point of the kernel) and passes thereby control to the kernel. The CPU begins executing the kernel's code directly (there is no process yet).
4. The kernel mounts the RAMDISK as a file system and then mounts the real file systen in place of the initial RAM disk image.

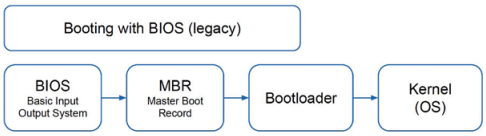
### Step 3: OS and Environment Initialization

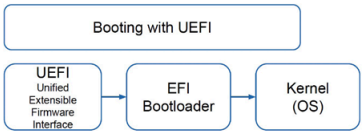
1. The OS queries the BIOS/bus system to get HW information and **initializes drivers** for each device.
2. Initializes **OS Management Structures** (e.g. process table, Interrupt/system-call tables, virtual memory, resource contorl) and creates **System Services** ( systemd, ssh, etc., so that users can interact)
3. Spawns a **(User) Interface** (textual/graphical login)

### UEFI (Unified Extensible Firmware Interface)

Replaces **BIOS**. UEFI is modular.
Replaces **MBR** with GPT (GUID Partitioning Table): Solves arbitrary number of partitions. Avoids collisions. Solves Security issues.

UEFI uses an architecture-independent virtual machine. Allows to execute any application or special binary files.




### System V (Legacy System Service)

At any moment the system is in predetermined states, called runlevels.

0 Halt: Shuts down the system
1 Single-user mode: Mode for administrative tasks
2 Multi-user mode: Does not configure network interfaces nor export networks services
3 Multi-user mode with networking: Starts the system normally
4 Not used/user-definable: For special purposes
5 Start the system normally with display manager (with GUI): Runlevel 3 + display manager
6 Reboot: Reboots the system

```
runlevel # Print previous and current SysV runlevel
```

**Limitations:** No knowledge of other system services and no dependency management. Very complex shell scripts to start system services. Replaced by **Systemd.**

### Systemd (System Service)

A **system and service (daemons) manager** for Linux OS.
It is responsible for initializing, managing, and maintaining system processes, services and dependencies after the Linux kernel has booted.
Replaces **System V** and manages the entire system, not just services.
**`systemd`** is the init system that runs in the background to manage services, logs, and dependencies.

Runs automatically on boot to initialize the system (a **`systemd`** user instance is launched by default on most distributions).

**`systemd`** units have a state: active, inactive, activating/deactivating or failed.

**Dependency System and Unit Files (Units)**

Dependencies are configured in service **unit configuration files.** There are many **Unit Types:** system service, socket, target, device, mount, timer, etc.

**Targets (.target):** Define a desired state. Replaces traditional runlevels. e.g:
network.target: Network is up and usable,
graphical.target: Multi-user mode with a GUI (like runlevel 5), etc.

**Service Unit (.service):** They start and control daemons and the processes they run while respecting **dependencies** with chronological order.

**Depenencies and Ordering:** Positive requirement: Requires=
Negative (optional) requirement: Wants=
Ordering dependencies: After= and Before=

```
# /lib/systemd/system/my_service.service
[Unit]
Description=My Sample Application Service
Documentation=https://example.com/docs/myapp
Requires=network.target
After=network.target
[Service]
Type=simple # forking, oneshot, notify, exec, ...
User=myuser
Group=mygroup
ExecStart=/bin/bash -c 'echo "Service started at $(date)" >> /home/ubuntu/service_started'
# Restart in combination with RestartSec, StartLimitInterval and StartLimitBurst
Restart=on-failure # no, always, on-success, on-abnormal, on-abort, on-watchdog
RestartSec=5
Environment=MYAPP_ENV=production
WorkingDirectory=/var/lib/myapp
StandardOutput=journal
StandardError=journal
[Install]
WantedBy=multi-user.target # Allows the service to start on boot
# RequiredBy=
```

Installation:

```
# 1. Create a new service file
sudo nano /lib/systemd/system/my_service.service
# 2. Enable the Service (Auto-start on Login)
systemctl --user daemon-reload
# 3. (optional) Enable the service so it starts automatically at login
systemctl --user enable my_service.service
# 4. Start the service manually
systemctl --user start my_service.service
# 5. Check its status
systemctl --user status my_service.service
# (optional) By default, user services only start when the user logs in.
# Ensure the User Services Run at Boot (even when user is not logged in)
loginctl enable-linger ubuntu
```

Units have a state: active, inactive, activating/deactivatingor failed.

The default configuration of systemd can be found in systemd configuration file at **`/etc/systemd/system.conf`**

Unit files are installed in **`/lib/systemd/system`** and overridden by system administrators in **`/etc/systemd/system`**

```
man systemd.service # Service unit configuration
```

**systemctl** **# Controls the systemd system and service manager**
**`systemctl`** is a command-line tool used to communicate with systemd and control services. A command-line tool to control the systemd system and service manager. Provides an interface to interact with **`systemd`**.

```
systemctl status <unit-name | PID>
systemctl status -all
systemctl --user enable <unit> # autostart a unit on user login
systemctl list-dependencies # list dependencies of unit
systemctl list-units # shows systemd-managed units
systemctl get-default # default target for the local operating system
graphical.target # -> the system boots into a GUI environment.

# By default, user services only start when the user logs in.
# Ensure a User Service run at boot, even when the user is not logged in.
loginctl enable-linger <service>
```

**journalctl** **# Print log entries from the systemd journal**
A command-line tool used to query, filter and view logs managed by systemd-journald, the logging component of **`systemd`**.

```
journalctl -u <unit-name | PID>
```

## Processes and Threads

### Process Creation

OS loads the executable (binary) into RAM and sets: Memory Map, Scheduler, Process Table, etc., Program Counter and Stack Pointer.

Physical memory is mapped to virtual memory.

**fork()**

A system call that copies the current process and creates a child process (with pid = 0) which runs concurrently with the caller. Both processes will execute the next instruction after `fork()`.

The child process **copies the resources (memory map)** with either **Copy-on-Write** or **Distinct Address Space** of the parent but uses the same pc (program counter), same CPU registers, and same open files. Results in **Process Hierarchy.**

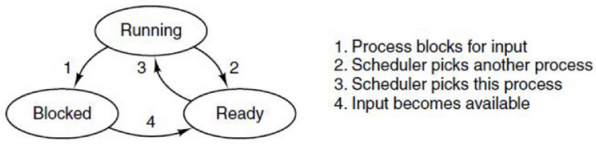
**Copy-on-Write:** Memory Map of Parent is shared as long as no changes are committed. A change request (Parent or Child) creates a copy.

### Process Termination

- Voluntary (return 0)
- Error, voluntary exit (e.g. return 1)
- Error, involuntary exit (e.g. segmentation fault / division by 0)
- Killed by request (kill -9 <PID>)

### Process Lifecycle

**R** Running. Process is actively running on the CPU or is ready to run.
**S** Sleeping. Process is waiting for an event (I/O) and can be woken.
**D** Uninterruptible. Process is waiting and and cannot be interrupted (I/O).
**Z** Zombie. Process has completed, but the parent hasn't read its exit status.
**T** Stopped or Traced. Process is stopped (SIGSTOP or being debugged).
**X** Dead. Process is dead (rare, should not appear).
**I** Idle. Only on multi-threaded systems; idle kernel thread




Reasons for state change: Mode (user/kernel space) or context switch, Interrupt, page fault (no more data in memory)

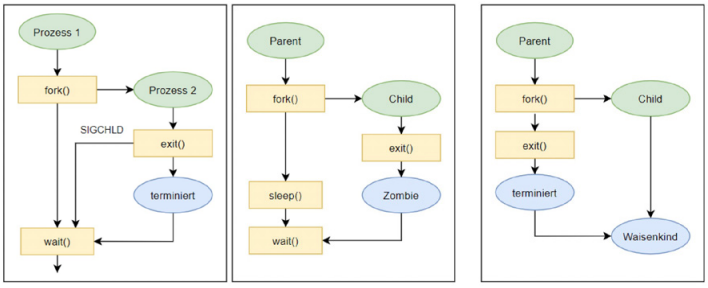
Each running process (including kernel threads) has a file **`/proc/<PID>/status`** where CPU information is available.

### Zombie, Orphan and Init Process

**Zombie Process:** Child dies first, but parent does has not called `wait()` or `waitpid()` yet (which is is supposed to do). `wait()` removes the child process from memory by the OS.

The child process can send a signal to the parent and the parent can terminate the child process immediately.

**Orphan Process:** Parent dies first, leaving the child process without a parent. **Init Process (systemd, PID 1)** automatically adopts orphans and cleans up.




### Thread

Created and owned by a process. Existence only within a process.
Each thread has its own stack and share same reseources.
Lightweight switching (OS doesn't have to allocate a new memory space).
Each thread has its own pc (program counter) and registers (working variables). Scheduler has no idea of the threads, only of the process.

In Linux, a thread is a standard **process** (task) created with **`clone3()`**, that shares a **configurable set of resources** with other processes (address space, file descriptors, etc.).

Kernel thread (kthreadd) has PID 2.

### Priority and Niceness

**PR (Priority)** shows how much priority the process has for CPU scheduling. The lower the value, the higher the priority. **PR = 20 + NI**

```
sudo chrt -r 80 ./myprogram  # 1 = lowest, 99 = highest
```

**NI (Nice value)** shows the *niceness* of the process, a value that can be adjusted to influence the priority of the process. Ranges from -20 (highest priority) to +19 (lowest priority). A higher nice value means the process is less important. Directly impacts the PR column, with more negative values increasing the priority of the process. Users can only modify nice-ness of their own processes.

```
nice -n 10 ./myprogram  # -20 = highest, 19 = lowest
renice <nice_value> -p <PID>
```

### Controllers and Control Groups (cgroups)

```
tmb@linux:/sys/fs/cgroup/pids> tree
.
|-- cgroup.clone_children
|-- cgroup.procs
|-- cgroup.sane_behavior
|-- init.scope
|   |-- cgroup.clone_children
|   |-- cgroup.procs
|   |-- notify_on_release
|   |-- pids.current
|   |-- pids.events
|   |-- pids.max
|   `-- tasks
|-- notify_on_release
|-- release_agent
|-- system.slice
|   |-- -.mount
|   |   |-- cgroup.clone_children
|   |   |-- cgroup.procs
|   |   |-- notify_on_release
|   |   |-- pids.current
|   |   |-- pids.events
|   |   |-- pids.max
|   |   `-- tasks
```

Allow tasks (processes and threads) to be organized into hierarchical groups whose usage of various types of resources can then be strictly controlled.
A cgroup is a collection of processes (tasks) that are bound to a set of limits. Cgroups enforce access and usage criteria. Cgroups is a Linux kernel feature.

**Subsystems** (also known as **Resource Controllers**) are modules in the Linux kernel that apply resource limits or accounting rules to control groups (cgroups).

Controller types: **cpu** limits CPU usage, **memory** limits memory usage, **blkio** (v1)/**io** (v2) limits block devices bandwidth, **pids** limits number of processes, **devices** controls access to devices.
Each cgroup is represented by a **directory.**

Each cgroup directory a set of files which can be read or written to, reflecting resource limits of the mounted resource controllers.

A task cannot be a member of two different cgroup in the same hierarchy (controller)

#### v1 vs v2

In v1 each controller has its own directory nd requires at least one cgroup filesystem: `/sys/fs/cgroup/cpu/mycgroup/ # cpu is the controller`
A cgroup v1 filesystem needs to be mounted, so that we can interact with the controller's settings and hierarchy through a directory. In short: mounting exposes the cgroup APIs via the filesystem. In v2 there is only one unified cgroup hierarchy, already mounted.

```
systemd-cgls # display the cgroup hierarchy in a tree-like format
```

Example in cgroup v1

```
# Mount the controllers. Each controller has its own directory.
sudo mount -t cgroup -o cpu none /sys/fs/cgroup/cpu
sudo mount -t cgroup -o memory none /sys/fs/cgroup/memory
# Create cgroup directories
sudo mkdir /sys/fs/cgroup/cpu/mygroup
sudo mkdir /sys/fs/cgroup/memory/mygroup
# Limit to 50% of one CPU
echo 100000 | sudo tee /sys/fs/cgroup/cpu/mygroup/cpu.cfs_period_us
echo 50000  | sudo tee /sys/fs/cgroup/cpu/mygroup/cpu.cfs_quota_us
# Limit memory to 200 MB
echo $((200 * 1024 * 1024)) | sudo tee /sys/fs/cgroup/memory/mygroup/
memory.limit_in_bytes
# Assign a process to both cgroups
echo <PID> | sudo tee /sys/fs/cgroup/cpu/mygroup/tasks
echo <PID> | sudo tee /sys/fs/cgroup/memory/mygroup/tasks
```

Example in cgroup v2

```
# Create a new cgroup directory
# In v2 there is only one unified cgroup hierarchy, already mounted.
sudo mkdir /sys/fs/cgroup/mygroup
# Enable controllers
echo +cpu +memory | sudo tee cgroup.subtree_control
# Limit to 50% of one CPU
echo "50000 100000" | sudo tee mygroup/cpu.max
# Limit memory to 200 MB
echo $((200 * 1024 * 1024)) | sudo tee mygroup/memory.max
# Assign a process to the cgroup
echo <PID> | sudo tee mygroup/cgroup.procs
```

#### CPU Share

```
cat /sys/fs/cgroup/user.slice/cpu.max
20000 100000
```

20000 → This is the allowed CPU time in microseconds (µs) per period.
100000 → This is the total period length in microseconds (µs).

$$\text{CPUQuota} = \left(\frac{20000}{100000}\right) \times 100 = 20\%$$

#### systemd Slice

A slice in systemd is a hierarchical cgroup (control group) unit used to group multiple systemd units and manage processes under specific resource constraints (CPU, memory, I/O, etc.). It helps in organizing system resources efficiently. Slices create a structured hierarchy for processes.

| | |
|---|---|
| **system.slice** | For system services (e.g., cron.service, sshd.service) |
| **user.slice** | For user processes (e.g., shells, GUI apps) |
| **init.scope** | For PID 1 (systemd itself), managing the root of the cgroup tree |

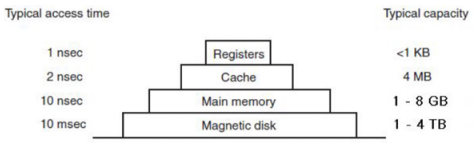
#### Helpfull Commands

```
# check the cgroup version
mount | grep cgroup

# information about the controllers that are compiled into the kernel
cat /proc/cgroups

# list cgroup control files (part of the cgroup hierarchy)
ls /sys/fs/cgroup
# cgroup.controllers – Lists available controllers (ecpu, memory, io)
# cgroup.max.depth – Defines the max depth of the cgroup hierarchy.
# cgroup.procs – Lists process IDs (PIDs) assigned to this cgroup.
# ...

# Check the root cgroup configuration
cat /sys/fs/cgroup/cgroup.controllers
# Show processes assigned to the root cgroup
cat /sys/fs/cgroup/cgroup.procs
# count how many processes are running in cgroup user.slice
wc -l /sys/fs/cgroup/user.slice/cgroup.procs

systemd-cgtop # monitor cgroup resource usage in real-time

# assign all user processes to a (mounted) cgroup
for pid in $(pgrep -u ubuntu); do
  echo "$pid" | sudo tee /sys/fs/cgroup/mygroup/cgroup.procs
done

# Set CPU limit for <user>
sudo systemctl set-property <user>.slice CPUQuota=50%
```

## Memory

Memory is a system resource and is therefore managed by the OS.

#### Memory Organisation




OS stores instruction in the cache.

### Fragmentation

Each process needs different amount of memory.

**Internal fragmentation:** Wasted space inside allocated memory blocks. Occurs when memory is allocated in fixed-size blocks and the process doesn't use the full block. Happens in **static memory division** (fixed Partitioning, static segmentation, Block-Based systems).

**External fragmentation:** Wasted space between allocated memory blocks. Occurs when memory is allocated and freed dynamically, leading to scattered free spaces that are too small to be useful. Happens in **Dynamic Memory Divison** (Variable Partitioning). Solutions: **Buddy Algrotihm** or **Paging**.

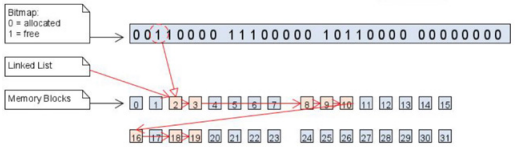
### Buddy Algorithm

Idea: Every request will find free space and divide it by 2 until best fit achieved.


Problem: Free Space Management

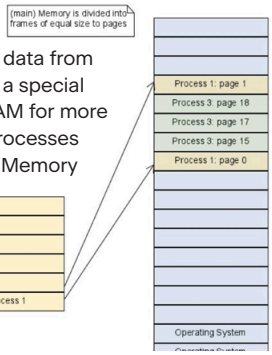
### Free Space Management

Must be found during next allocation. Implemented as Bitmap and Linked List.



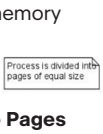
### Swapping

Solution when main memory runs (RAM) out.




Swapping is when the operating system moves data from RAM (physical memory) to hard disk (usually to a special space called swap space) in order to free up RAM for more urgent tasks. This helps the system run more processes than can physically fit in RAM at once → Virtual Memory

Problem: Need to partition memory so this is efficiently possible
Solution: **Paging,** Memory divided into **Frames** (e.g. 4kB) and processes into **Pages** of equal size (e.g. each 4 kB).




Fragmentation only happens in the last page for leftover bits/bytes.

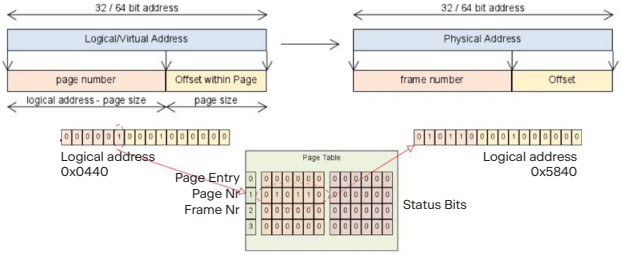
4 kB is quite small and modern programms have large variations in memory requirements → Pages Frames up to 2 GB in size. Divide **memory map** into different **Zones** and manage memory separately across each zone.

### Frames, Pages and Virtual Memory

Pages are managed through a page table.
Programmer sees unlimited memory (virtual memory).

#### Address Translation

The **Memory Management Unit (MMU)** handles the address translation.

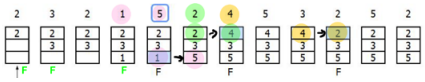



Problem: Page Tables can get very large.
Solution: Translation Lookaside Buffer (TLB), cache for MMU for recently accessed page tables entries.

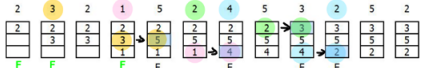
### Page Replacement

A page fault occurs when a program tries to access a page not in RAM (but located on disk). The OS then finds the page on disk and loading it into RAM. If RAM is full → Page Replacement (swap page in RAM with the one on disk).

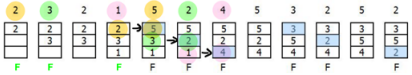
**Optimal:** Replaces page which is used the latest in the future



**LRU:** Least recently used (Linux approach)



**FIFO:** First In first Out



F: Page fault = Page not in RAM → fetch from disk → Page Replacement

### Shared Libraries / Dynamically Linked Libraries (DDL)

Solution to wasted memory when multiple processes execute the same library function. Libraries have **position-independent code.** Code is then dynamically linked with a process. Library must be then provided when running.

### Pinning and Locking Memory

**Locking:** Ensure that the memory stays in RAM and is not swapped out. Means you're preventing a region of memory from being swapped out to disk by the kernel. Usually requested in **user-space** application, directly via system calls (`mlock, mlockall`).

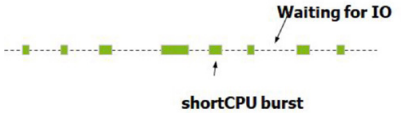
**Pinning:** Prevent page faults and ensure memory stays in a specific location. Means not only locking it into RAM (`mlock`), but also preventing the memory pages from being moved or relocated by the kernel. Usually requested in **kernel-space** (kernel itself or device drivers). All pinned memory is effectively locked, but not all locked memory is pinned.

#### Mental Model

**Locking** is like telling the OS: *«Hey, keep this page in memory, I don't want delays.»*
**Pinning** is like telling the OS: *«This page must stay exactly here in physical memory because a piece of hardware depends on it.»*

## Scheduling

Scheduling is a resource-time management activity.




**Burst Time** is an expression for the **runtime of a task** representing the **computing time** and not including any wait times for I/O which may increase the actual completion time of the task.

### Basics

When to schedule: Process is created/terminated, task blocks on I/O, interrupt (I/O) occurs

1. **Scheduler** sorts a **queue** according to some policy.
2. **Dispatcher** moves the task from the head of the ready queue to execution on CPU (performs the context switch)

**Preemptive:** Scheduler interrupts a task
**Cooperative:** Non-Preemptivescheduler
**Starvation:** Task does not get any CPU-time

### Policies

**Fairness:** Giving each process a fair share on the CPU while keeping all parts of the system busy.
**Throughput:** Maximizing jobs per hour
**Turnaround time:** Minimize time between submission and termination
**CPU utilization:** Keep the CPU busy at all times
**Response time:** Respons time to requests (Interactive Systems)

### FIFO (First-In-First-Out)

Run to completion. No awareness of nature of jobs. Non-preemptive.

### Round Robin

Single queue, preemptive. No starvation.
Arriving task has priority. Tasks run for a predefined timeslice called quantum.

Problems:
- Fair but no priorities and wastes quantum when a task stalls on I/O
- Not realtime: Real-time systems need responsiveness to I/O (e.g. keybard)

| Task | Arrival t | Burst t |
|---|---|---|
| T1 | 0 | 10 |
| T2 | 3 | 6 |
| T3 | 7 | 1 |
| T4 | 8 | 3 |

Quantum: 2

| Task | Time | Queue (low to high priority)<br>T1:10 (task:remaining time) |
|---|---|---|
| T1 | 0 – 2 | T1:8 |
| T1 | 2 – 4 | T1:6, T2:6 (T2 arrived) |
| T2 | 4 – 6 | T2:4, T1:6 |
| T1 | 6 – 8 | T1:4, T2:4, T3:1, T4:3 (T3 and T4 arrived) |
| T4 | 8 – 10 | T4:1, T1:4, T2:4, T3:1 |
| T3 | 10 – 11 | T4:1, T1:4, T2:4 |
| T2 | 11 – 13 | T2:2, T4:1, T1:4 |
| T1 | 13 – 15 | T1:2, T2:2, T4:1 |
| T4 | 15 – 16 | T1:2, T2:2 |
| T2 | 16 – 18 | T1:2 |
| T1 | 18 – 20 | |

### Rate Monotonic Scheduler

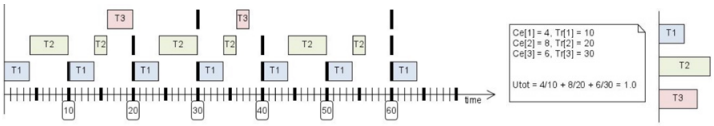
Real time scheduler. Tasks with **shorter period (T)** (= higher repetition rate) get **higher priority.** Tasks are **ordered by priority** and executed sequentially, each executed for its own **WCET (C)** time. WCET (Worst Case Execution Time) represents the maximum possible runtime of a piece of code or a process/thread independently of the runtime environment and any scheduling factors. WCET includes wait times for I/O.

Scheduling is guaranteed if **Utilisation *U***

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

*C* is runtime (WCET)
*T* is period (1 / repetition rate)
*n* Number of tasks

The **smaller the period**, the **higher the priority.**




### Earliest Deadline First Scheduler

Like Rate Monotonic, but dispatches the task with the next deadline first. Schedule is achievable when Utilisation *U* <= 100%.

### Linux Schedulers

Different tasks can be scheduled with different schedulers. Two policies:
Real-Time Policy: Deadline, FIFO, Round Robin
Normal Policy: Batch, Idle

## Input/Output (IO)

OS abstracts the heterogeneity and complexity of devices and interfaces (e.g. USB) by providing a unified approach to work with devices (peripherals).

Device Categories: Block-level (block of bits) and Character-level (bit-stream)

**Blocking/non-blocking vs Async/Sync**
Async vs Sync is choice of the consumer (process).
Blocking vs Non-Blocking is a feature of the provider (IO system).

### Block Device

A block device is used to store data in fixed-size blocks (sectors), typically with a block size of 512 bytes or 4 KB. It provides random access to data.
A block device is mountable to the filesystem to store data.
Example: Hard drive

### Character Device

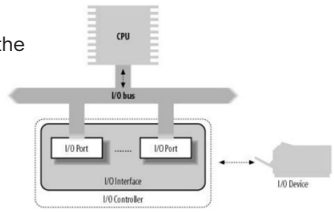
A character device provides sequential access to data. Example: **`/dev/ttyS0`**, represents a serial port for communication. These devices are typically used for continuous data streams or low-level communication with hardware.
Examples: **`/dev/zero`** provides an infinite stream of zero bytes.
**`/dev/random`** generates random data.

### I/O Controller

I/O Controller (electronic interface and part of device) needs to be able to talk to other CPUs. OS installs drivers for that device (OS needs to know about the HW). Device drivers can be pre-installed, embedded in HW or downloaded seperatly. Driver = usually a programm written in Assembly.
I/O always runs in kernel mode. Only the kernel can talk to devices directly.

**I/O-Ports:** Address that points to controller registers which enables the CPU to interact with the controller.

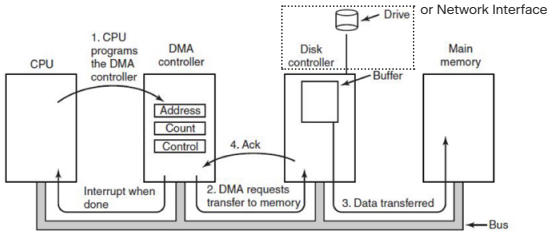
```
# list ports
less /proc/ioports
0040-0043 : timer0
0060-0060 : keyboard
...
```




### Direct Memory Access Controller (DMA)

Proxy/middlewear, bypasses the CPU for **data transfer tasks.**
Takes some of the load of the CPU.

1. CPU programs the DMA controller
2. DMA requests transfer to memory from Disk controller (or network interface)
3. Data transferred from Buffer in Disk controller to Main memory
4. Disk controller sends Ack ro DMA controller.




### Interrupt Handling

When an interrupt occurs, the current flow of execution is suspended and a special routine called Interrupt Handler is executed. After the handler completes the previous execution flow is resumed. Interrupts are maskable (can be ignored). Types of interrupts:

- **Synchronous:** Generated by executing an instruction (software)
- **Asynchronous:** Generated by an external event (IO device)

### Shared Access

As soon as access to IO is shared, performance becomes a real issue.
There are several tools to analyze IO performance, like `iostat` and `iotop`.

### Linux Device Model (/sys) and UDEV (/dev)

Linux maintains a representation of devices in user space mounted at **`/sys`** (virtual filesystem): **block** (block devices, e.g. disks, partitions), **bus** (types of bus to connect physical devices, e.g. usb), **firmware, fs** (information about mounted file systems), **kernel** (kernel status information), **module** (list of modules/drivers currently loaded), **power** (power management subsystem)

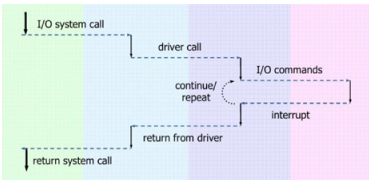
**UDEV** (Userspace **`/dev`**) is the device manager for the Linux kernel and part of systemd. Manages device nodes in /dev and handles hotplug events (e.g., USB inserted). Runs in userspace.

The **/dev** directory contains special files, each representing an IO device. These files allow to talk to devices via standard IO operations (i.e. r/w on character- or block-level). Program can accesses them. Device files behave almost exactly like ordinary files, yet the kernel passes file operations on those special files to device drivers.

### Linux Device Access

```
fd = open(/home/tmb/textfile.txt);
read(fd, buffer, 50);
```

open() → I/O system → driver call




### Custom Kernel

Steps:

1. Choose and download kernel source code at https://www.kernel.org/
   `wget https://cdn.kernel.org/pub/linux/kernel/v6.x/linux-6.12.26.tar.xz`
2. Get an existing kernel config:
   `ls /boot | grep config`
   `cp /boot/config-6.8.0-45-generic .config`
3. Defining the configuration file:
   `make menuconfig # or # make oldconfig`
   Select features and modules and save to `.config` file.
4. Build the kernel from binary by execute a Makefile:
   `make -j $(nproc) bindeb-pkg`
   `# -j $(nproc) uses all available CPU cores for faster compilation`
5. Install the new kernel:
   `sudo dpkg -i ../linux-*.deb`
6. Reboot the system to apply the changes:
   `sudo reboot`
   `uname -a # check if new kernel is installed`

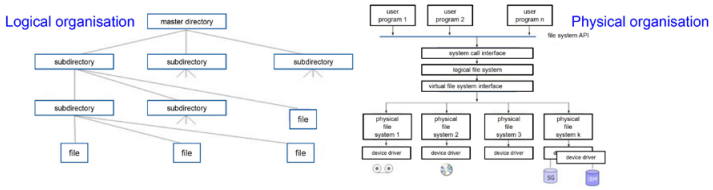
## Filesystem

Requirements:
- Store very large amounts of information
- Information must survive termination
- Able to access information concurrently
- Support multiple storage media

### Basics

A file is an abstraction, a logical unit of information. Files are managed by the file system, a part of the OS.

**Logical Organisation (User Perspective) vs Physical Organisation**

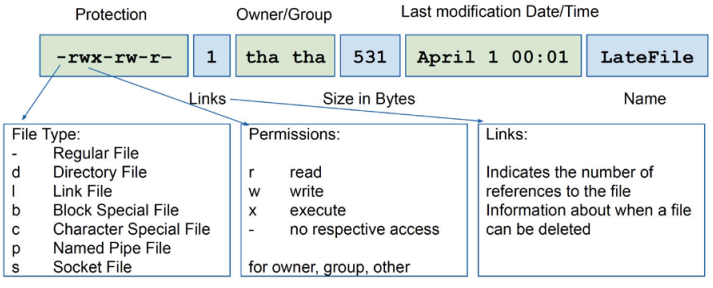



**File Types**
- **Regular (Ordinary) Files:** User files
- **Directory Files:** System Files with information about the file system
- **Character and Block Files:** Represent I/O devices
- **Special Files:** Links, Sockets (/var/run/mysocket.sock), Named Pipe Files

### File Permissions (Access Rights) and Attributes

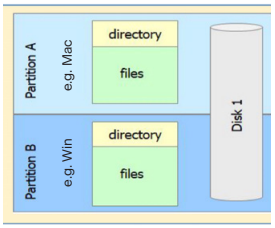
```
755 -rwxr-xr-x user group filename      rwx = 421(7)    r-x = 401(5)
user, group, others   - No access, r Read(4), w Write(2), x Execute(1)
File Type: - Regular File, d Directory, l Simbolic link, etc.

--x on a dir:  permission to traverse the dir (cd) but not list (ls).
To ls (list) I need at least r-x permission.
```

To delete or rename a file: The user needs **`wx`** rights on the directory.

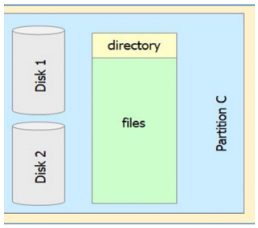


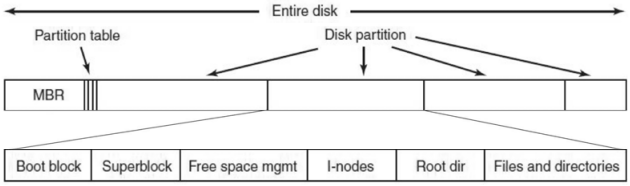
## Disk Organisation

Physical Volume




Logical Volume




```
sudo gdisk /dev/vdb # create partions in a filesystem

sudo mkfs.ext4 -b 1024 /dev/vdb1 # format the partitions
sudo mkfs.ext4 -b 4096 /dev/vdb2

sudo mkdir -p /mnt/vdb1
sudo mount /dev/vdb1 /mnt/vdb1 # mount a partition vdb1
```

## Disk Space Management

### Continuous Allocation

Challenge: Choosing the right Block size. The larger the block size, the higher the data rate but bigger fragmentation:

- Too small: Many blocks per file → small internal fragment. but slow access
- Too big: few blocks per file → higher data rate but big internal fragentation

### Linked-List Allocation

No external Fragmentation but access slow and problem if pointer is corrpted.

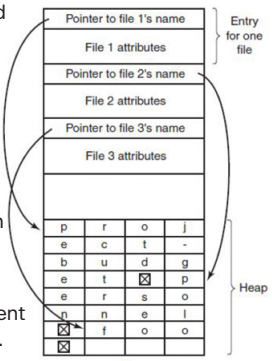
### Indexed Allocation (I-nodes)

Each file has an associated data structure called **index node (I-node). Verwaltungseinheit** eines Files, eine Tabelle für File-Metadaten und wird vom Kernel verwaltet.

An index node lists file **attributes** (owner, size, access timestamp) and **block addresses** (physikalischer Ort des Files auf der disk).

Ein Filename gehört nicht zum File, es ist nur ein Mittel, um ein File dem User zu präsentieren.

The i-node **number (ino)** is **unique to each filesystem,** so when the file is moved to a different filesystem (e.g. a tmpfs), it gets a new i-node Nr.



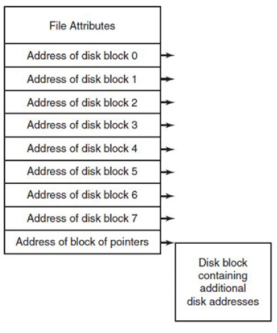

```
stat file # view i-node metadata
ls -i # List the i-node numbers
```

Advantages

- Fast address lookup and random access: Loaded into memory only when needed (on demand)
- No external fragmentation

Drawbacks

- I-nodes need disk space
- Additional structure to maintain




## File System Consistency

Idea: Keep a log of what a file system is going to do before it does it.
Example deleting a file requires 3 steps:

1. Remove the file from its directory
2. Release the i-node to the pool of free i-nodes
3. Return all the disk blocks to the pool of free disk blocks

## Virtual Filesystem (VFS)

A virtual filesystem doesn't store data on a physical disk. They are loaded into RAM and automatically mounted on bootup. It is a way for the kernel to expose system information or interfaces using a file/directory layout, even though no real files exist on disk.

| | |
|---|---|
| /proc | Process and system info |
| /sys | Kernel objects and hardware |
| /sys/fs/cgroup | Control groups interface |
| /dev | Devices as files |

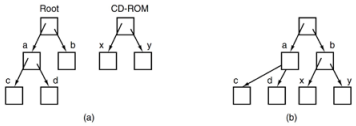
## Device

In Linux, a device refers to any hardware component that interacts with the operating system, such as storage drives, keyboards, mice, network interfaces, and more. Devices are represented as special files in the /dev/ directory, known as device files or device nodes. Examples:

/dev/tty (teletypewriter = terminal)
/dev/random (random number generator)

## Mounting

Mounting is the process of **making a filesystem** (e.g., a hard drive, USB drive, or network storage) **accessible in the directory structure.** Without mounting, a storage device cannot be used by the system.




(a) Before mounting, the files on the CD-ROM are not accessible.
(b) After mounting, they are part of the file hierarchy.

```
mount <device> <mount_point>
unmount <mount_point>
```

`<device>` The storage device or partition (e.g. USB device partition /dev/sdb1)
`<mount_point>` The directory in the filesystem (e.g. /mnt/usb)

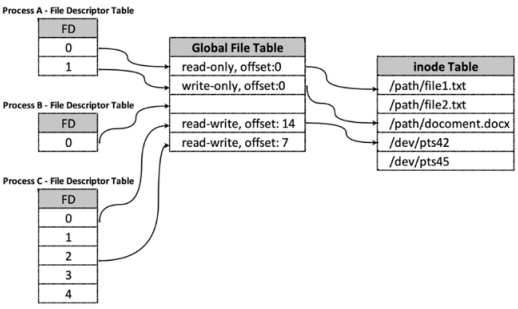
To list all currently mounted devices, use: **`mount`** or **`findmnt`**

## File Descriptor

Every file is made of:

- **Inode:** Metadata + a pointer to the data blocks.
- **Filename (directory entry):** Just a reference to the inode.

Filedescriptor = Integer, dass eine Nummer einem File zuweist. OS weist jedem File eine systemweite eindeutige Nummer im Descriptor Table zu. Jedes öffnen eines Files, kreiert ein Eintrag im Open File Table. Der Kernel verwaltet geöffnete Files anhand einer Integer-ID, das ist der File Deskriptor (fd). System Calls teilen diese File Deskriptoren mit dem User Programmen, um mit tatsächlichen Files zu verknüpfen.




## Hard and Soft Links

**Hard Link:** *Same physical file.* Maps a file with same **i-node** of an existing file. Cannot be created for directories. The actual file data on disk is only deleted when all hard links to the file have been removed and no process is keeping the file open (in memory).

The i-node number is **unique to each filesystem,** so when the hard-link file is moved to a different filesystem (e.g. a tmpfs), it gets a new i-node number and does not link to the original file anymore.

**Symbolic (or Soft) Link (symlink):** Ein File dessen Inhalt ein **Pfad** zu einem anderen File/Directory ist. File that points to a file or directory in the system, but doesn't mirror the other file's data. This file gets its own i-node.

Link across file systems: If you want to link files across the file systems, you can only use symlinks/soft links. But if we change the name of the original file or even move it, then all the soft links for that file become **dangling:**

If the target file is moved or deleted → the symlink breaks.
If the symlink itself is moved → might break, depending on the type of path it stores.

```
# ln – make links between files
ln original.txt hardlink.txt     # create hard link
ln -s original.txt softlink.txt  # soft link, stores the realtive path
ln -s $(realpath original.txt) softlink.txt  # store absolute path

# readlink – print resolved symbolic links or canonical file names
readlink softlink.txt  # prints the symlink stores
realpath softlink.txt  # prints the path realpath of the file
vim $(realpath softlink.txt)
```

### Temporary Filesystem (tmpfs)

A tmpfs (temporary filesystem) is a special type of in-memory filesystem that stores files in RAM (= virtual memory) instead of on disk. This makes it extremely fast but also volatile, meaning everything stored in tmpfs is lost when the system reboots.

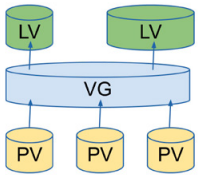
```
sudo mkdir /mnt/mymnt
sudo mount -t tmpfs -o size=2G tmpfs /mnt/mymnt
```

### Logical Volume Management (LVM)

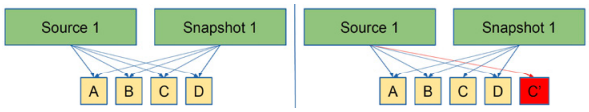
**Logical Volumes (LV):** Named partitions containing a file system and spanning across multiple disks without being physically contiguous.

**Volume Group (VG):** Named collections of physical and logical volumes.

**Physical Volumes (PV):** Disks providing block space to store logical volumes.




**Snapshots:** Freeze an existing LV at a particular point in time using CoW. Allows the full backup of a system without disabling write access.
**Copy on Write (COW):** Efficiently duplicate or copy a resource upon modification (only if that resource is shared).
**Balancing:** Spread allocation evenly on all available physical disks.




Create a Logical Volume Group from a Physical Device:

```
pvcreate /dev/vdb /dev/vdc # Initialize Physical Volumes (PV)
vgcreate myVG /dev/vdb /dev/vdc # Create a VG and attach PVs
lvcreate -L 1G -n myLV myVG # Create Logical Volume on VG
mkfs.ext4 /dev/myVG/myLV # Format the Logical Volumes (LV)
mkdir /mnt/myDir
mount /dev/myVG/myLV /mnt/myDir # mount LV
lvextend --resizefs -L 8000M /dev/myVG/myLV # Resize a logical volume
lvcreate --size 50M --snapshot --name myLVSnpsht /dev/myVG/myLV # snapshot
```

### fsync

The `fsync()` system call is used to flush data to disk, ensuring that any changes made to a file are permanently written to disk.

When you modify a file (like appending to a log or writing data), the data isn't immediately written to disk. Instead:
- It's stored in the filesystem's cache (in memory).
- The OS will write it to disk later (during periodic syncing).

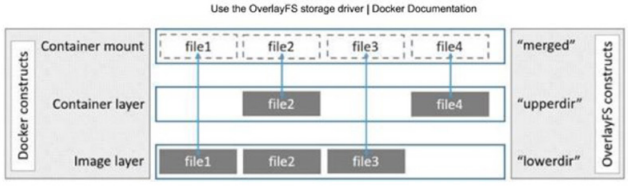
### Linux File System Disk Layout

Linux supports different format versions: **ext2, ext3, ext4** and **BtrFS.** ext3 and successot ext4 also support Journaling.

BtrFS is a B-tree filesystem (mkfs.btrfs).

**Another Union File System (AuFS) / Overlayfs**
Overlayfs is a layered filesystems. It overlays an existing filesystem transparently on top of another one. This allows changes to be made without modifying the original (lower) files. Perfect for sandboxing, containers (Docker, Podman) and versioning.

**Merged** Final view the system sees (combined result)
**Upperdir** Read-write layer (e.g., user changes)
**Lowerdir** Read-only base directory (e.g., a system image)




```
mkdir lower upper work merged
mount -t overlay overlay -o \
  lowerdir=/lower,upperdir=/upper,workdir=/work /merged
```

You only read/write to the merged/ directory. All changes are automatically reflected in the upperdir. The lowerdir remains untouched.

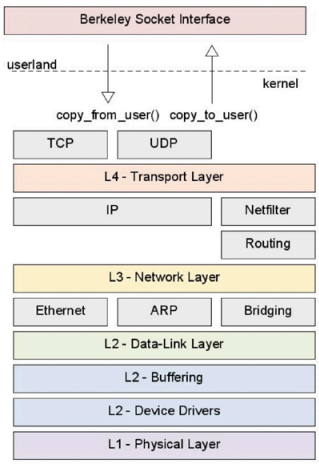
```
echo "test" > merged/notes.txt
# Overlayfs takes care of it and creates notes.txt in upperdir/
# merged/notes.txt now maps to upperdir/notes.txt
```

OverlayFS does not physically delete the file from /lowerdir when removing files through /merged.

## Networking

OS requirements: Support for different network interfaces (ethernet, wireless).

### OSI Model




L5–7: Application Layer
L4: TCP/UDP, Firewall (nftables)
L3: Routing, IP
L2: Briding, Device Drivers, Buffering, Bridge/Switch, Ethernet-Frame, MAC, ARP
L1: Repeater, 100BASE-TX, etc.

### L1 – Physical Layer (PHY)

A physical Network Interface Card (NIC) is a kernel space driver and is connected via the system-bus to the rest of the computer system.
It has two integrated circuits: **Physical Layer Device** (L1) and the **Media Access Controller (MAC)** for L2.

- Rx: Decodes and de-serializes incoming signals to data streams.
- Tx: Encodes and serialises outgoing bytes to serial format.

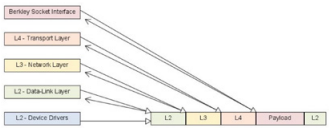
### L2 – Ethernet Media Access Controller (MAC)

Rx: MAC receives byte stream from PHY, checks incoming frames and places data into a buffer in main memory. Activates the NIC driver via an interrupt.

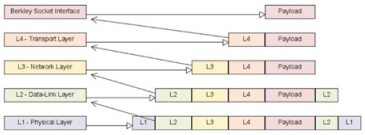
Tx: Notified by NIC driver that frame is ready for transmission and forwards frame to PHY. Data is transfered by **Direct Memory Access Controller (DMA)**.

#### Zero-Copy Stack Architecture

Only pointers to frames are copied.



Comparison: A naive implementation would copy payload 4 times.



Linux copies headers and pointer to payload.

### L3 – Network Layer

**Linux Netfilter System:**
- Packet selection and filtering
- Network Address Transaltion (NAT = Port Mapping)

### L7 – Berkeley Sockets

Sockets (a special file type, /var/run/mysocket.sock) connect a port and a network address with a protocol and a user-process. A socket is defined by:

- **IP address** (e.g., 192.168.1.10)
- **Port number** (e.g., 80 for HTTP)
- **Protocol: TCP** (IP_PROTOTCP) or **UDP** (IP_PROTOUDP)

**TCP** (IP_PROTOTCP) is a **stream**-type protocol (SOCK_STREAM):
Connection-oriented and Reliable: A connection must be established Data is sent as a continuous stream of bytes

**UDP** (IP_PROTOUDP) is a **datagram**-type protocol (SOCK_DGRAM):
- Connectionless and Unreliable: Each packet is sent independently
- Data is sent as discrete packets (datagrams) with defined boundaries.

Sockets have a specific type (i.e., communication semantics):
SOCK_STREAM, SOCK_DGRAM, SOCK_RAW.
Raw sockets (SOCK_RAW) can be used to get protocol-free access.

At the OS level, a socket is a file descriptor. Creating a socket in a program, the operating system returns a file descriptor. You can then use standard operations like read(), write(), select(), and close() on it. If my program connects to a port, the kernel checks if my process have rw permissions and returns a file descriptor (= key to the file).

Protocols operate in a domain:
- AF_INET / AF_INET6, AF_BLUETOOTH
- AF_UNIX, AF_NETLINK (kernel sockets)

## Helpfull Commands

### System Information

```
uname -a # print system information
Linux lab-files 6.8.0-45-generic #45-Ubuntu SMP PREEMPT_DYNAMIC Fri Aug 30 12:02:04 UTC 2024 x86_64 x86_64 x86_64 GNU/Linux
uname -r # check current running kernel
```

```
less /proc/cpuinfo
model name      : AMD EPYC 7713 64-Core Processor
cpu MHz         : 1996.249
...
```

```
lscpu # display information about the CPU architecture
Architecture:             x86_64
  CPU op-mode(s):         32-bit, 64-bit
  Address sizes:          48 bits physical, 48 bits virtual
...
```

```
nproc # get the number of CPU cores, or lscpu | grep "^CPU(s):"
```

```
lshw # display inof about hardware architecture
hwinfo # probe for hardware. 'apt install hwinfo'
lsmod # list all modules
insmod # insert module into running kernel
runlevel # Print previous and current SysV runlevel
free # Display amount of free and used memory in the system
```

### Process related Commands

```
ps # process status
ps aux # list all processes
ps -u $USER # list all processes started by the current user
ps xawf -e # display process information in a tree format
```

```
htop # interactive process viewer
top # display sorted information about processes
top -p 1750,1751
# PR (Priority): Default priority (higher values mean lower priority)
# NI (Nice value): Priority adjustments (negative values -> higher prio)
```

```
pstree # display a tree of processes
pidof # find the process ID of a running program
kill -9 <PID> # terminate a running process
env or printenv # run a program in a modified environment
size /bin/ls # list section sizes and total size of binary files
   text     data      bss      dec      hex  filename
 127793     4936     4760   137489    21911  /bin/ls
```

```
objdump # prints how memory is organized of a programm
objdump -dj .text /bin/ls # -d disassemble, -j section
```

```
# Start a job in the background
wc /dev/zero &
```

```
^Z # (Ctrl + Z) pause and move the current process to the background
bg # resume the process in the background
```

```
jobs -l # list all jobs
```

```
fg # resume a job in the foreground
fg %<JOB-ID>
```

### Input/Output

```
lsblk # list block devices
NAME    MAJ:MIN RM  SIZE RO TYPE MOUNTPOINTS
vda     253:0    0   80G  0 disk
├─vda1  253:1    0   79G  0 part /
└─vda16 259:0    0  913M  0 part /boot
vdb     253:16   0    1G  0 disk # <-- not mounted disk
```

```
# /dev/vda is the main system disk (vda = virtual disk)
# /dev/vda1 is the root partition
```

```
grep vda /proc/diskstats # show details of a specific device
cat /proc/diskstats | grep vda # same
```

```
iostat # CPU and IO statistics for devices and partitions
iostat -p vda # virtual disk device vda and its partitions
iostat -dm /dev/vda
```

```
uevent # infos about device events (added, removed or changed)
```

### Networking

```
ifconfig # configure a network interface. 'apt install net-tools'
ip # show/manipulate routing, network devices, interfaces and tunnels
dig <url> # DNS lookup utility
hostname # show or set the system's host name
tc # show / manipulate traffic control settings
lnstat # unified linux network statistics
bridge # show / manipulate bridge addresses and devices
```

### Various

```
tail -f /var/log/syslog # -f monitor a file for change
```

### Filesystem and Partitioning

```
tree # list contents of directories in a tree-like format
```

```
fdisk # manipulate disk partition table
sudo fdisk -l # list partition tables
```

```
mount # list all mounted filesystems
# or
df -h # report file system space usage, -h: human readable
Filesystem      Size  Used Avail Use% Mounted on
/dev/vda1        19G  2.2G   17G  12% /
tmpfs           392M   12K  392M   1% /run/user/1000
```

```
mkfs # build a Linux filesystem
```

```
mkfs.ext4 /dev/vdb # format the new disk
# mkfs - make filesystem. Creates a new filesystem on a storage device
#        (such as a hard drive, SSD, or partition).
# ext4 - the type of the filesystem to create.
mount /dev/vdb /mnt # mount
lsblk | grep vdb#
vdb     253:16   0    1G  0 disk /mnt # <-- mounted
```

```
stat <file> # view i-node metadata
fsck # check and repair a Linux filesystem
```

```
stat file # view i-node metadata
ls -i # List the i-node numbers
```

```
sudo gdisk /dev/vdb # create partions in a filesystem
sudo mkfs.ext4 -b 1024 /dev/vdb1 # format the partitions
sudo mkfs.ext4 -b 4096 /dev/vdb2
sudo mkdir -p /mnt/vdb1 # mount partition vdb1
sudo mount /dev/vdb1 /mnt/vdb1
```

```
# measure your disk write output
dd if=/dev/zero of=speedtest bs=10M count=100 conv=fdatasync
# if: input file, of: output file, bs: block size
# count: number of blocks to copy
# conv=fdatasync: ensures that the data is physically written to disk
# before the command completes. But slower than without syncing.
# When you modify a file (like appending to a log or writing data),
# the data isn't immediately written to disk. Instead:
# - It's stored in the filesystem's cache (in memory).
# - The OS will write it to disk later (during periodic syncing).
```