R5CYBgos - Introduction to computer systems - 1

Samuel Pélissier (samuel.pelissier@centralesupelec.fr)

2025

Goals

- High level understanding of the why and how of operating systems
- Linux and Windows CLI should not scare you
- Manage containers
- Improve debug methods
- Prepare for following classes

Planning

- CM1 : OS basics and boot
- CM2 : files and file system
- CM3 : softwares and CLIs
- TD 1 et 2 : Linux CLI
- TD 3 et 4 : Windows CLI
- TP 1 et 2 : OS install
- CM 4 : Containers
- CM 5 : Containers
- TP 3 et 4 : Containers

Computer and Software

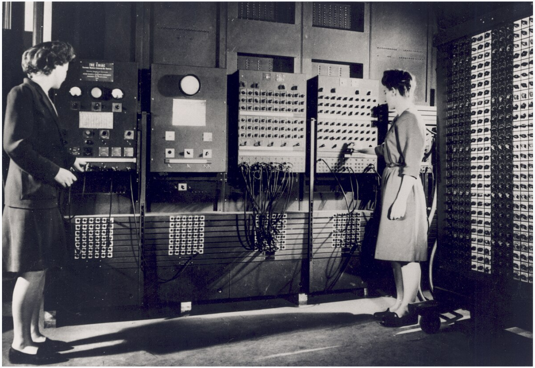
Digital technologies are based on the interaction between:

- Software, describing how to process information (intangible)
- Computers, capable of executing these programs (tangible)

Computers

A tangible history

- 1946: ENIAC electronic calculator
  - Architecture based on lamps and vacuum tubes: 30 tons, 170 m^2 floor space, 5,000 additions per second
  - 0.005 MIPS (Millions of Instructions Per Second)
  - **Programming required rewiring**.



Programmers Betty Jean Jennings (left) and Fran Bilas (right) operating ENIAC (Wikimedia).

Computers

A tangible history

- 1947: Invention of the transistor
- 1958: Invention of the silicon integrated circuit
  - Multiple transistors arranged on the same substrate

A replica of the first working transistor, a point-contact transistor invented in 1947 (Wikimedia).

Computers

A tangible history

20 µm – 1968
10 µm – 1971
6 µm – 1974
3 µm – 1977
1.5 µm – 1981
1 µm – 1984
800 nm – 1987
600 nm – 1990
350 nm – 1993
250 nm – 1996
180 nm – 1999
130 nm – 2001
90 nm – 2003
65 nm – 2005
45 nm – 2007
32 nm – 2009
28 nm – 2010
22 nm – 2012
14 nm – 2014
10 nm – 2016
7 nm – 2018
5 nm – 2020
3 nm – 2022

- 1971: Intel 4004 processor
  - 2,300 transistors in a single integrated circuit
  - 740 kHz, 0.092 MIPS

Fast forward 40 years...

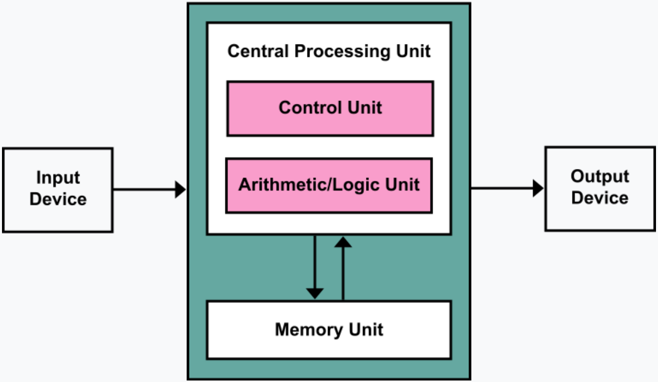
- 2011: Intel Core i7 2600K
  - 1.4 billion transitors
  - 4 cores, 8 threads
  - 3.4 GHz, 128,300 MIPS

Fast forward another 14 years...

- 2025: Apple M3 Ultra
  - 186 billion transistors

Architectures: the Von Neumann architecture (1945)

- Programs are stored in memory
- Data and program instructions are accessed from the same memory unit



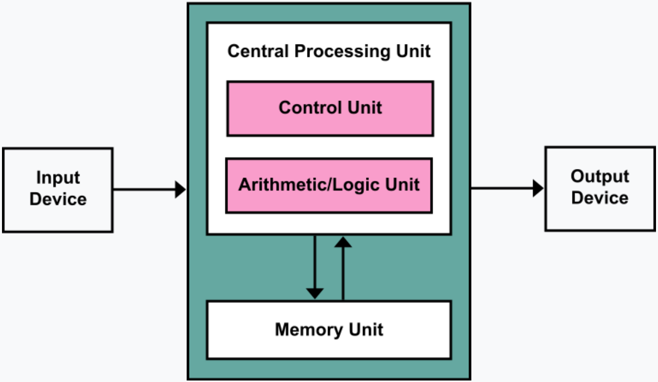
Architectures: the Von Neumann architecture (1945)

Components:

- *Control unit*: sequence operations (read instruction and act upon it)
- *Arithmetic unit*: arithmetic operations
- *Memory*: stores data and instructions
- *Outside recording medium*: somewhere to store input/output
- *Input / output mechanisms*: transfer data between memory and some outside recording medium

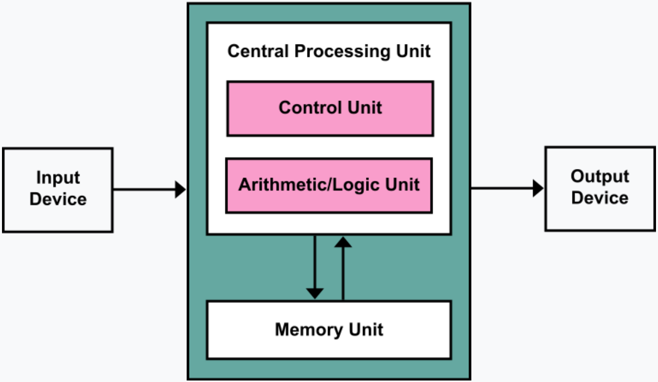
Architectures: the Von Neumann architecture (1945)

Where is the bottleneck?



Architectures: the Von Neumann architecture (1945)

Where is the bottleneck?



Shared memory and single bus for both data and instructions.
This can cause the **Von Neumann bottleneck**, where the CPU is stalled waiting for memory access

Architectures: alternatives and optimizations

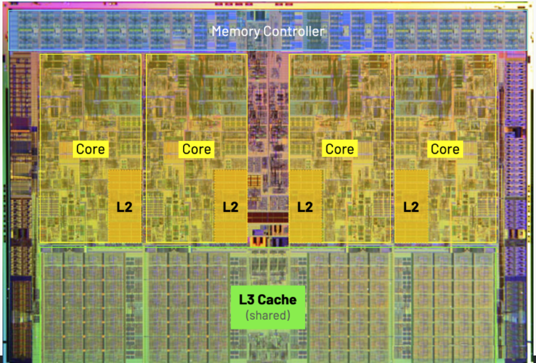
Harvard Architecture: separate memory and buses for instructions and data, *allowing simultaneous access*

Modern CPUs draw from Von Neumann and Harvard architectures: separate instruction/data caches (Harvard style) but shared main memory (Von Neumann style)

Architectures: alternatives and optimizations

Additional optimizations include:

- **Cache:** frequently used data and instructions stored close to the CPU to reduce memory delays
- **Parallelization:** multiple cores, pipelines, and SIMD units allow concurrent processing



Nehalem microarchitecture, used by the Intel Core i5 and i7 processors

Architectures: where's the rest?

So far, we've focused on the *processing unit*.

What building blocks do we need for a computer?

Architectures: where's the rest?

So far, we've focused on the *processing unit*.

What building blocks do we need for a computer?

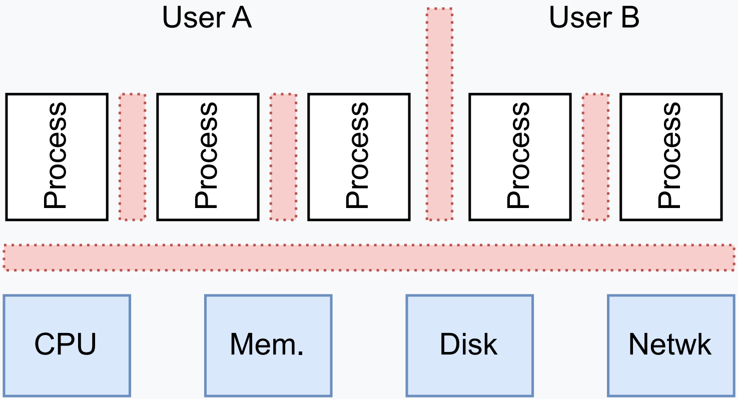
CPU, Memory (RAM), Storage, I/O devices, Networking...

Operating Systems

The operating system (OS) == an abstraction layer

- **Manages and allocate resources**
  - Memory, CPUs, devices
  - Processes and threads
  - Scheduling
- **Controls & protects**
  - Interrupts and exceptions (e.g., divide by zero, I/O completion)
  - Access rights and security policies
  - Memory protection and isolation between processes

Operating Systems



Operating Systems

The OS manages hardware specifics:
developers can use **uniform APIs**

Reading a file from disk without an OS

- Understand the physical disk layout (sectors, tracks, blocks)
- Send low-level commands to the disk controller
- Manually manage timing, interrupts, and I/O completion
- Handle errors (bad sectors or device busy states)
- Translate raw bytes into structured data

With an OS

- Call `open()`
- Call `read()`

Operating Systems

There are dozens of operating systems...

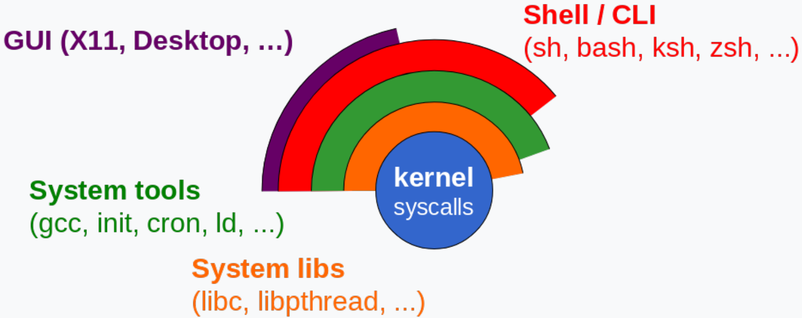
- Windows
  - XP, 7, 10, 11...
  - Windows Phone
- Unix and Unix-like
  - GNU/Linux (distributions: Ubuntu, Debian, SUSE, Arch, RedHat...)
  - Android (based on the Linux kernel)
  - BSD family (FreeBSD, OpenBSD, NetBSD, ...)
  - Bonus certified UNIX: AIX, Solaris, HP-UX
- Apple systems
  - macOS (formerly OS X)
  - iOS (and derivatives: iPadOS, watchOS, tvOS)

... with differences in architectures, implementations, features

OS & kernel

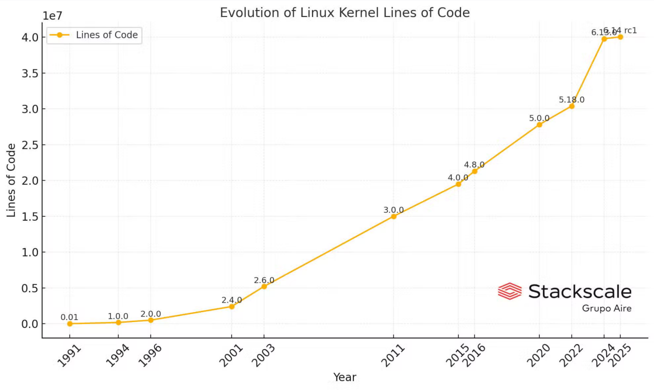
OS =/= kernel (most critical part of an OS)

Example for Linux:



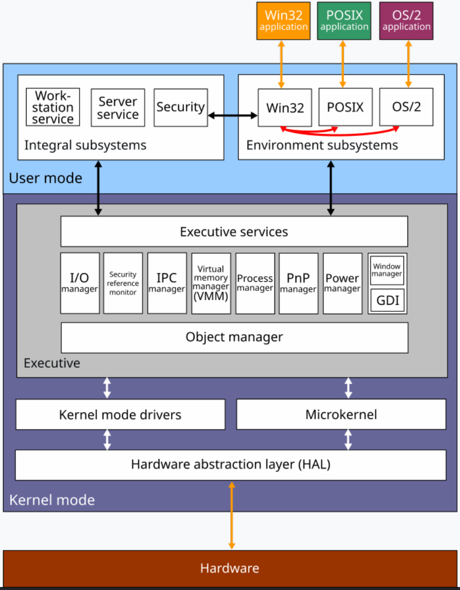
OS & kernel

The kernel itself can be complex: 40M lines of code (v6.14)

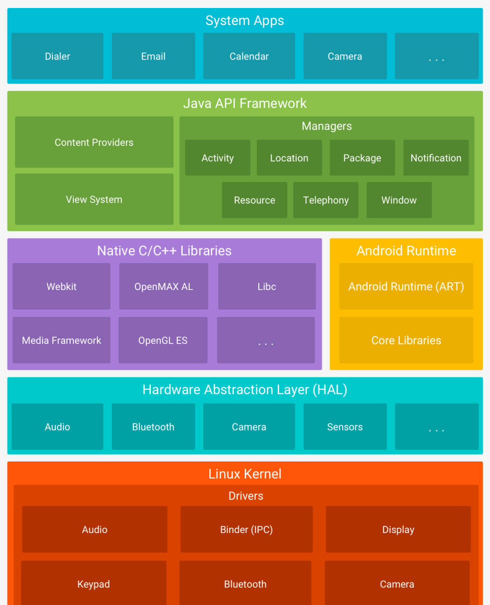


Not everything is included when compiled: a majority corresponds to **drivers** (softwares handling specific hardware components) or **dynamically loaded modules**

Another OS example: Microsoft Windows



Yet another OS example: Android



Boot Process: From Power On to Login Screen

Why is the boot process important?

From an usability standpoint:

- Debugging becomes infinitely easier once the basic steps are known

From a security standpoint:

- Anything that runs before the OS is invisible to the OS
- Modern firmware has numerous capabilities (e.g., send network requests)
- **Can't trust the OS if the boot chain is not trusted**

Boot TL;DR

Power On ⚡ → **Firmware initialization** → **Boot loader** → **Kernel** → **Rest of operating system**

Notes:

- This is a simplified overview of the process
- The following slides are also a simplification
- It should be enough to understand what is going on when powering on a computer

Step 1: Power On

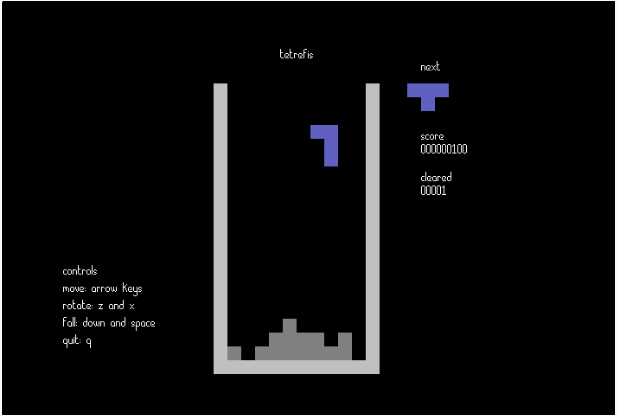
- Power button pressed $\rightarrow$ Power Supply Unit converts AC to low-voltage DC
- CPU jumps to a pre-defined address
- CPU fetches the first instruction from **firmware** (**UEFI**)
- **POST** (Power-On Self-Test): checks CPU, RAM, keyboard, storage, GPU
- Errors reported via beep codes (platform-dependant)

Step 2: Firmware Initialization

On top of POST, UEFI (Unified Extensible Firmware Interface):

- Initializes the hardware required for booting (disk, keyboard controllers... may also include network interface, GPU)
- **Detects available boot options** (drives, USB devices, network, etc.)
- Supports execution of EFI applications (e.g., **bootloaders**, shell and diagnostic tools, firmware updaters, networking stack...)
- Provides a user-accessible configuration interface (**boot order**, **secure boot**, its own settings)

Step 2: Firmware Initialization



Tetris as an EFI application

What are the implications for security?

Step 2: Firmware Initialization

How does UEFI work?

- UEFI firmware reads **boot entries stored in NVRAM**
- Each boot entry points to:
  - A **disk and partition** (usually an ESP)
  - The path to an **EFI executable** (e.g., `/EFI/ubuntu/grubx64.efi`)
- **EFI System Partition (ESP)**:
  - Special partition (FAT32, required on GPT disks)
  - Stores bootloaders, EFI applications, and (rarely) drivers
  - Provides a standardized fallback path: `/EFI/boot/bootx64.efi`
  - Systems can use one (recommended/standard) or multiple ESP

Step 2: Firmware Initialization

Illustration / demo

```
~ ❯ lsblk -f
NAME                                          FSTYPE      FSVER LABEL       UUID                                 FSAVAIL FSUSE% MOUNTPOINTS
nvme0n1
├─nvme0n1p1                                   vfat        FAT32             CCAE-BFAE                            1021,5M     0% /boot/efi
└─nvme0n1p2                                   crypto_LUKS 1                 9c382968-a086-4fda-9fb1-d2c00f2ddd27
  └─luks-9c382968-a086-4fda-9fb1-d2c00f2ddd27 ext4        1.0   endeavouros 06223aa6-a501-47fd-8c11-18e8faccb8d6  186,9G    75% /
~ ❯ sudo tree /boot/efi/EFI/
/boot/efi/EFI/
├── boot
│   └── bootx64.efi
└── endeavouros
    └── grubx64.efi
```

Step 2: Firmware Initialization

How does multibooting work?

- The **ESP contains subdirectories for each OS**/vendor (e.g. `/EFI/Microsoft`, `/EFI/ubuntu`)
- Each subdirectory stores that OS's bootloader (EFI application) and related support files
- UEFI firmware reads boot entries from NVRAM:
  - Each entry specifies a **disk**, a **partition** (often the ESP), and the **path to a bootloader** (e.g. `/EFI/ubuntu/grubx64.efi`)
  - We get a cool **menu with boot entries** (not to be confused with the following bootloader's menu, e.g. GRUB's)

Step 2: Firmware Initialization

What about BIOS?

- **Legacy**
- Term used by default (**all modern PCs actually use UEFI**)
- Relies on boot sectors
  - Selects a device
  - Reads the first 512 bytes corresponding to the Master Boot Record (MBR)
  - MBR contains the bootloader information (448 bytes, used to load the actual bootloader) and partition table (64 bytes)
- **Less flexible**
  - Max 2TB
  - Max 4 primary partitions
  - Simplified: does not support secure boot (required for modern Windows)

Step 2: Firmware Initialization

Power On ⚡ → **Firmware initialization** UEFI (prev. BIOS) → **Boot loader** → **Kernel** → **Rest of operating system**

- Hardware check (POST)
- Bootable devices (USB, HDD...) detection and selection
- Start bootloader from ESP
⚒️ *- May load temporary (UEFI only) drivers*

Step 3: Bootloader

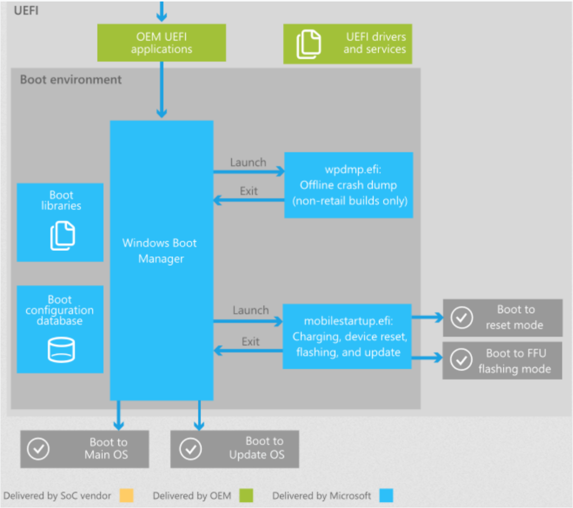
- The bootloader (EFI application) is loaded (e.g., **GRUB** or systemd-boot on Linux, **Windows Boot Manager**)
- May show OS selection menu (**boot with specific options**, e.g. ...)
- **Loads kernel**
- That's it.

Step 3: Bootloader and kernel (Linux)

GRUB:

- Loads kernel (usually `vmlinuz-X`) with wanted parameters
- Then, the kernel unpacks the **initramfs** in RAM
  - Provides a **minimal environment** with basic tools and **drivers**
  - Exists because **the kernel cannot** (and should not) **statically include every possible driver**
  - Commonly **includes drivers for storage** (SATA, NVMe, USB, etc.), for **encryption** (LUKS), scripts to **mount the real root filesystem**

Step 3: Bootloader and kernel (Windows)



Windows Boot Manager (`BOOTMGFW.efi`)

- Reads boot options from the Boot Configuration Data (BCD) file and selects the Windows installation to load
- Can also launch EFI application (e.g., crash dumps)

Step 3: Bootloader and kernel (Windows)

- **Windows Boot Loader** (`WINLOAD.efi`) loads the **kernel** (`ntoskrnl.exe`), which then initializes the **hardware abstraction layer** (`hal.dll`), and boot-critical drivers into memory
- Boot-critical drivers include storage drivers (SATA/NVMe/RAID) and, if the system volume is encrypted, BitLocker support (`fvevol.sys`)
- The kernel also sets up the **Windows Registry**
  - Hierarchical key:value database
  - Information, settings, options for everything running on Windows
  - Available via `REGEDIT` (GUI) and `reg.exe` (CLI)

Step 3: Bootloader and kernel (Windows)

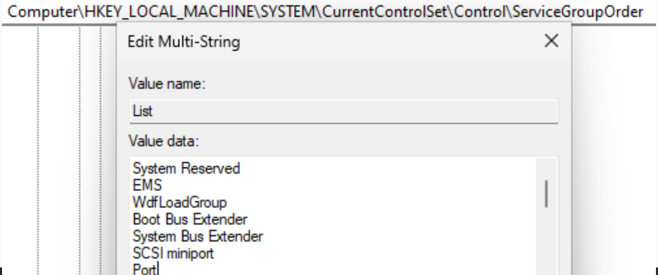
Illustration / demo

```
-a----          9/9/2025  10:44 AM          28672 BCD
-a----         9/13/2023  11:22 AM          10895 boot.stl
-a----         9/13/2023  11:22 AM        2577376 bootmgfw.efi
-a----         9/13/2023  11:22 AM        2560480 bootmgr.efi
-a----         9/13/2023  11:23 AM          54608 kdnet_uart16550.dll
-a----         9/13/2023  11:22 AM          87528 kdstub.dll
-a----         9/13/2023  11:23 AM          71008 kd_02_10df.dll
-a----         9/13/2023  11:23 AM         443744 kd_02_10ec.dll
-a----         9/13/2023  11:23 AM          70992 kd_02_1137.dll
-a----         9/13/2023  11:23 AM         279904 kd_02_14e4.dll
-a----         9/13/2023  11:23 AM          91488 kd_02_15b3.dll
-a----         9/13/2023  11:23 AM          83280 kd_02_1969.dll
-a----         9/13/2023  11:23 AM          71008 kd_02_19a2.dll
-a----         9/13/2023  11:23 AM          62800 kd_02_1af4.dll
-a----         9/13/2023  11:23 AM         333136 kd_02_8086.dll
-a----         9/13/2023  11:23 AM          54608 kd_07_1415.dll
-a----         9/13/2023  11:23 AM          87392 kd_0C_8086.dll
-a----         9/13/2023  11:22 AM        2340848 memtest.efi
-a----         9/13/2023  11:22 AM          10341 winsipolicy.p7b
```

Step 3: Bootloader and kernel (differences)

Main difference between Linux and Windows:

- On Windows, boot-critical drivers are explicitely marked and enforced by the OS (groups defined in reg `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\ServiceGroupOrder`).
- On Linux, admins can include more or fewer drivers in the initramfs script; the OS does not care.



Step 3: Bootloader and kernel

Power On → **Firmware initialization** → **Boot loader** → **Kernel** → **Rest of operating system**

Firmware initialization
UEFI
(prev. BIOS)

- Hardware check (POST)
- Bootable devices (USB, HDD...) detection and selection
- Start bootloader from ESP
- May load temporary (UEFI only) drivers

Boot loader
GRUB
Windows Boot Manager + Windows Boot Loader

Load kernel with parameters

Step 3: Bootloader and kernel

Power On ⚡ → **Firmware initialization** → **Boot loader** → **Kernel** → **Rest of operating system**

Firmware initialization

UEFI
(prev. BIOS)

- Hardware check (POST)
- Bootable devices (USB, HDD...) detection and selection
- Start bootloader from ESP
⚒️ *- May load temporary (UEFI only) drivers*

Boot loader

GRUB
Windows Boot Manager + Windows Boot Loader

Load kernel with parameters

Kernel

vmlinuz-*
initramfs
ntoskrnl.exe
hal.dll

Load minimal drivers for storage / encryption
⚒️

Rest of operating system

Step 5: Init System & Services (Linux)

After the bootloader and kernel start up:

- ✅ Minimal drivers are loaded
- ✅ The root file system is mounted

Well then?

- **The kernel starts** `/sbin/init`
- This is actually the first *process* of the system (PID 1); it runs until system shutdown and is the parent of all subsequent processes
- In *most* modern Linux systems, `/sbin/init` corresponds to **systemd**

```
~ ❯ ls -lah /sbin/init                                                        20:02:49
lrwxrwxrwx 1 root root 22  4 sept. 06:50 /sbin/init -> ../lib/systemd/systemd*
~ ❯ ls -lah /usr/sbin/init                                                    20:02:50
lrwxrwxrwx 1 root root 22  4 sept. 06:50 /usr/sbin/init -> ../lib/systemd/systemd*
~ ❯                                                                           20:03:02
```

Step 5: Init System & Services (Linux)

systemd

- An **init system** and **service manager**
- A *service* is either:
  - Long-running background process, also called *daemon* (e.g., `systemd-resolved.service` for network name resolution)
  - One-time task (e.g. `systemd-modules-load.service` to load kernel modules)
- A service can start/stop/reload its configuration, it can *depend* on other services

Step 5: Init System & Services (Linux)

At boot, systemd:

- **Organizes and parallelizes service startup** according to their dependencies
- Coordinates drivers or **kernel modules loading** via `systemd-udevd` (e.g., `vboxdrv` for VirtualBox)
- **Mounts** the rest of the **file systems and partitions**
- Starts services: **networking** (NetworkManager, Bluetooth), **graphical interfaces**, etc.
- Starts the **user session**
- Optionally, bakes cookies

Step 5: Init System & Services (Linux)

Demo / illustration: `man bootup` & `systemd-analyze blame`

```
 19ms modprobe@fuse.service
 19ms sys-kernel-tracing.mount
 18ms bluetooth.service
 18ms kmod-static-nodes.service
 18ms systemd-rfkill.service
 17ms systemd-remount-fs.service
 16ms dbus-broker.service
 16ms cups.service
 15ms systemd-userdbd.service
 15ms modprobe@drm.service
 12ms dracut-pre-pivot.service
 11ms avahi-daemon.service
 10ms rtkit-daemon.service
  9ms lightdm.service
  9ms systemd-modules-load.service
  7ms systemd-udev-load-credentials.service
  6ms wpa_supplicant.service
  5ms dracut-shutdown.service
  5ms systemd-battery-check.service
```

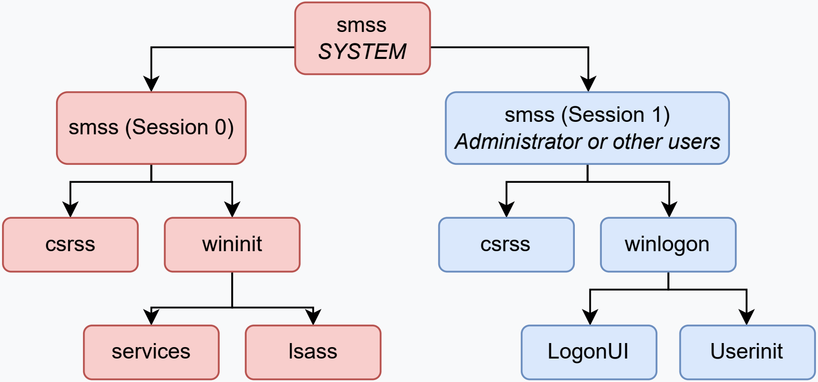
Step 5: Init System & Services

What about Windows? :)

Step 5: Init System & Services (Windows)

- ✅ `ntoskrnl.exe` (kernel) is running
- ✅ Boot-critical drivers are loaded

Well then?



Step 5: Init System & Services (Windows)

The kernel launches the Session Manager Subsystem (`smss.exe`)

- Loads environment variables (e.g., %PATH%, %TEMP%, etc.)
- Creates sessions

User subsystem in **Session 0**:

- **Highest privileges** (`SYSTEM`)
- Starts the Win32 subsystem via smss
- Through `wininit.exe`:
  - **Services Control Manager** (`services.exe`): **high privilege background processes** (e.g., networking, event logging), somewhat analogous to `systemd`
  - **Local Security Authority Subsystem** (`lsass.exe`): **user authentication, local security policy**, access tokens.

Step 5: Init System & Services (Windows)

User subsystem in **Session 1**:

- Lower privileges (administrator or other users)
- Prompts for credentials (`logonui.exe`)
- Spawns custom environment for the user (`userinit.exe`) and graphical environment (`explorer.exe`)

Bonus: each session also launches `csrss.exe` (Client/Server Runtime Subsystem) console, process, and thread management

Step 5: Init System & Services (Windows)

Illustration/demo:

Name	PID	Status	User name	CPU	Memory (a...	Archite...	Description
System interrupts	-	Running	SYSTEM	00	0 K		Deferred procedure calls and interrupt servic...
System Idle Process	0	Running	SYSTEM	89	8 K		Percentage of time the processor is idle
System	4	Running	SYSTEM	00	16 K		NT Kernel & System
Registry	160	Running	SYSTEM	00	8,576 K	x64	NT Kernel & System
sppsvc.exe	348	Running	NETWORK...	00	5,180 K	x64	Microsoft Software Protection Platform Servi...
smss.exe	532	Running	SYSTEM	00	244 K	x64	Windows Session Manager
svchost.exe	612	Running	SYSTEM	00	6,912 K	x64	Host Process for Windows Services
fontdrvhost.exe	660	Running	UMFD-0	00	1,064 K	x64	Usermode Font Driver Host
csrss.exe	684	Running	SYSTEM	00	880 K	x64	Client Server Runtime Process
fontdrvhost.exe	688	Running	UMFD-1	00	1,212 K	x64	Usermode Font Driver Host
wininit.exe	756	Running	SYSTEM	00	676 K	x64	Windows Start-Up Application
svchost.exe	764	Running	SYSTEM	00	1,028 K	x64	Host Process for Windows Services
csrss.exe	768	Running	SYSTEM	00	912 K	x64	Client Server Runtime Process
svchost.exe	852	Running	LOCAL SE...	00	2,712 K	x64	Host Process for Windows Services

Step 5: Init System & Services

Power On → **Firmware initialization** → **Boot loader** → **Kernel** → **Rest of operating system**

Power On

Firmware initialization
UEFI
(prev. BIOS)

- Hardware check (POST)
- Bootable devices (USB, HDD...) detection and selection
- Start bootloader from ESP
- May load temporary (UEFI only) drivers

Boot loader
GRUB
Windows Boot Manager + Windows Boot Loader

Load kernel with parameters

Kernel
vmlinuz-*
initramfs
ntoskrnl.exe
hal.dll

Load minimal drivers for storage / encryption

Rest of operating system
systemd
smss.exe
lsass.exe

Starts services (e.g., networking), sessions, graphical interface...

Secure Boot

Problem:

- Malware can persist in boot partitions or as EFI applications (*bootkits*)
- Before the OS takes over $\Rightarrow$ **antivirus cannot detect them**
- **No integrity or authenticity guarantee by default**

Solution: $\Rightarrow$ **Create a chain of trust starting from the firmware up to the OS**

Secure Boot

How?

- UEFI firmware contains trusted public keys
- Each boot component (firmware drivers, EFI apps, bootloader, kernel) must be signed
- On startup:
  - Firmware verifies digital signatures before execution
  - Only trusted code is allowed to run
- If verification fails $\Rightarrow$ no more boot.

Chain of trust:

(Hardware) $\rightarrow$ Firmware $\rightarrow$ Bootloader $\rightarrow$ Kernel

Secure boot

Who has the keys?

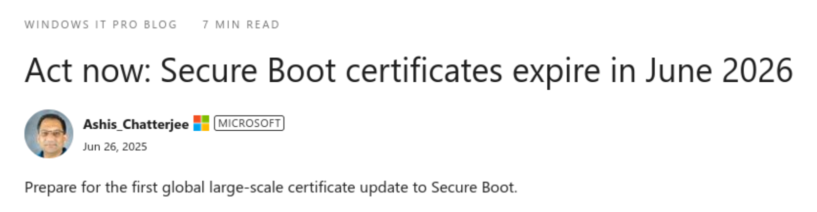
- Most consumer hardware is shipped with Microsoft keys
- To setup secure boot with a Linux distribution, one needs to ask Microsoft to sign the EFI application
- Microsoft don't want to sign everything every time there is an update
- Instead: distributions asked to sign a small EFI application, `shim`
    - Contains keys of the distribution
    - Can verify the signatures of the rest of EFI apps / drivers / kernel, signed by the distribution

Secure boot

Example for Debian:

Power On ⚡ → **UEFI** 🔒 *Signed by Microsoft* → **shim** 🔒 *Signed by Microsoft, includes Debian keys* → **Any other EFI application (e.g., GRUB)** 🔒 *Signed by Debian* → **Kernel** 🔒 *Signed by Debian*

Secure Boot


WINDOWS IT PRO BLOG 7 MIN READ

Act now: Secure Boot certificates expire in June 2026

Ashis_Chatterjee MICROSOFT

Jun 26, 2025

Prepare for the first global large-scale certificate update to Secure Boot.

https://techcommunity.microsoft.com/blog/windows-itpro-blog/act-now-secure-boot-certificates-expire-in-june-2026/4426856

Resources

Computer architecture & OS

- Modern operating systems, Andrew S. Tanenbaum
- Operating Systems: Three Easy Pieces, Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau (University of Wisconsin-Madison)
- In French : Architecture des Ordinateurs, François Pellegrini (U. Bordeaux)

Resources

- 🎁 Modern CPUs have a backstage cast, Hugo Landau, www.devever.net/~hl/backstage-cast
- 🐧 Arch Wiki: boot process, https://wiki.archlinux.org/title/Arch_boot_process
- 🐧 More information on initrd/initramfs, https://en.wikipedia.org/wiki/Initial_ramdisk#Implementation
- Boot and UEFI, https://learn.microsoft.com/en-us/previous-versions/windows/drivers/bringup/boot-and-uefi
- 🎁 nt-load-order Part 1: WinDbg'ing our way into the Windows bootloader, Colin Finck, https://colinfinck.de/posts/nt-load-order-part-1/

Resources

System & services

- Windows System Initialization, https://os.cybbh.io/public/os/latest/006_windows_boot_process/winboot_fg.html#_3_windows_system_initialization
- 🎁 It rather involved being on the other side of this airtight hatchway: Elevation from Administrator to SYSTEM, Raymond Chen, https://devblogs.microsoft.com/oldnewthing/20150923-00/?p=91531
- SUSE doc: Introduction to systemd Basics, https://documentation.suse.com/smart/systems-management/html/systemd-basics/index.html

Resources

Secure boot

- Debian Wiki: secure boot, https://wiki.debian.org/SecureBoot
- 🎁 There's a Hole in the Boot (GRUB vulnerability report), https://eclypsium.com/blog/theres-a-hole-in-the-boot/