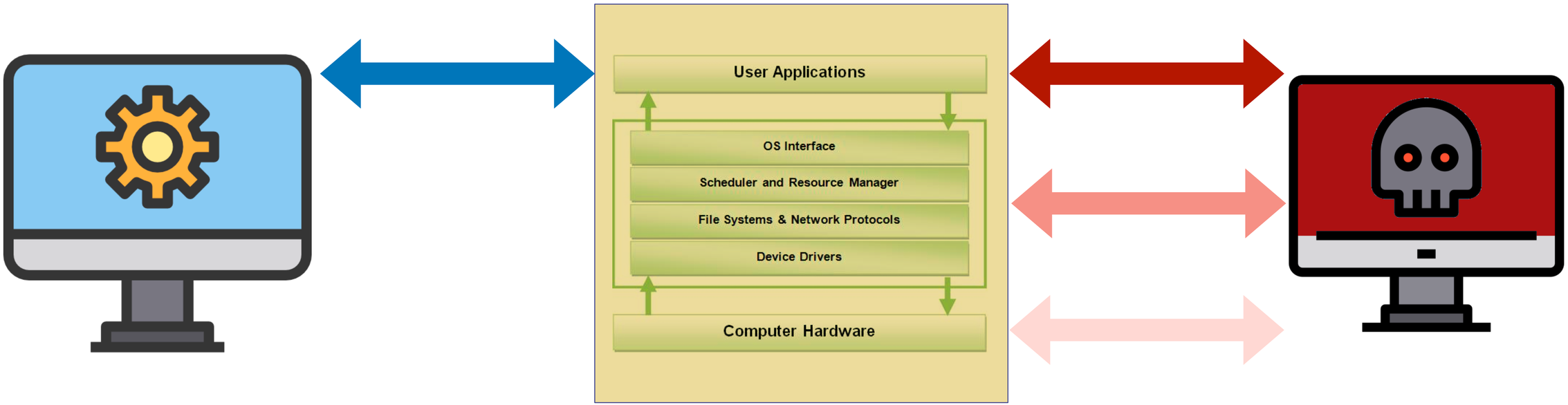
Fundamentos de Segurança Informática (FSI)

2024/2025 - LEIC

Systems Security (Part 2)

Hugo Pacheco
hpacheco@fc.up.pt

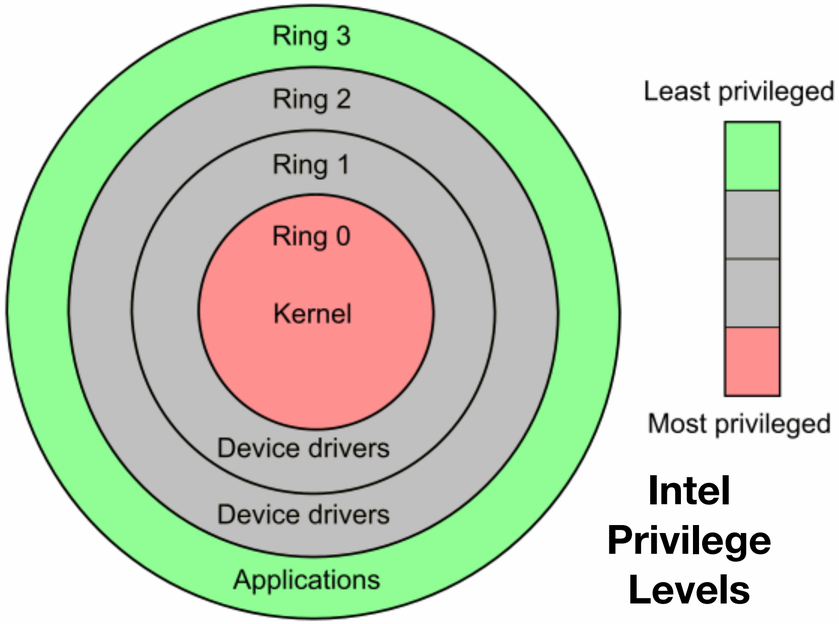
Systems Security



- **Honest processes** run “isolated” (OS “guarantee”)
- **Dishonest processes** try to break isolation and/or abuse resources
- **Hierarchy of trust:** ↑ increasingly less trusted levels ↓ increasingly critical attacks

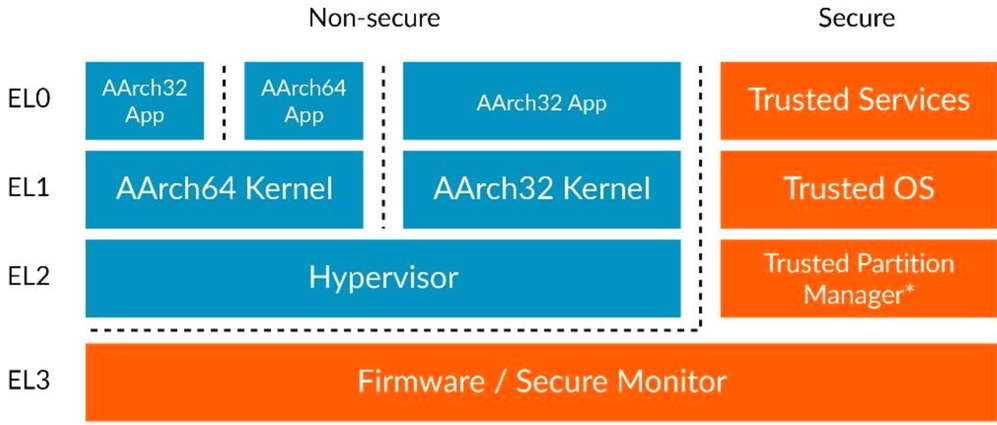
Kernel

- The **kernel** is a part of the OS that performs the most critical operations (complete mediation 🕎):
  - When in **kernel mode**, the code is allowed (almost) all operations
  - HW support: processors define various levels of privilege (defense in depth 🕎)
  - In many cases, only 2 levels: **kernel mode** and **user mode**
  - When in **user mode**, the code does not have direct access to the system resources
  - Any exchange of information between the 2 levels is part of the **attack surface**



Kernel

- The **kernel** is protected from **user-mode** processes:
  - Has its own memory space managed independently by the kernel itself
  - The processor guarantees that only code running in **kernel mode** can execute a set of privileged instructions (separation of privilege 🕎)
  - Any process in **user mode** (including device drivers) must be able to access system resources using **system calls** (least privilege 🕎)
  - Part of the code of **system calls** executes in **kernel mode**! (😈 later)



ARM Privilege Levels

Monitoring

- The **entry points of system calls are critical**:
  - To cause critical damage, a user-mode process has to invoke a system call
  - Solution: implement monitoring mechanisms that control the **security perimeter**, i.e., system calls
  - **Reference monitor** (complete mediation 🕋 + compromise recording 🕋):
    - Always present: if it terminates, all monitored processes have to be terminated (fail-safe defaults 🕋)
    - Has to be simple: to be easier to analyse and validate than the whole system (economy of mechanism 🕋)

System Calls

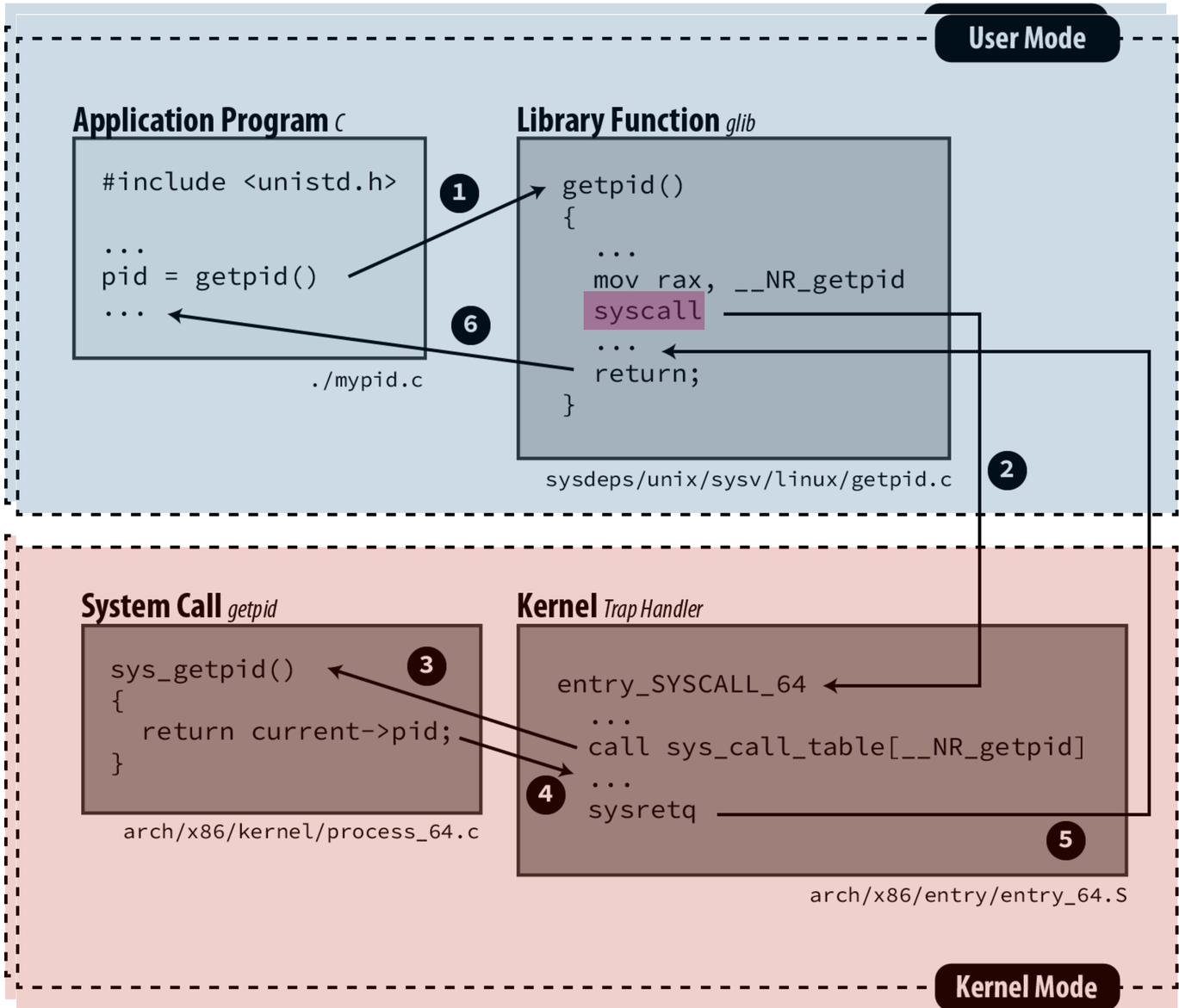
- Control processes (e.g., fork, load, execute, wait, alloc, free)
- Access files (create, read, write, etc.)
- Manage devices (obtain access, write/read, etc.)
- Configure the system (time, data, characteristics, state, etc.)
- Network (establishing connections, send/receive messages, etc.)
- Capabilities (change permissions, obtain access to resources)

System Calls

- To escalate privileges, the **user mode** code must:
  - Prepare arguments, and identify an allowed entry point (system call) for accessing **kernel mode**
  - Execute a **special instruction** that passes control to the kernel
- **Well-defined attack surface** ⇒ **There is a limited number of entry points** (economy of mechanisms 🕎):
  - Specific registers for parameters, which are typically pointers to memory of processes in user mode
  - The processing of this information is the full responsibility of the kernel

System Calls

- **User-mode** code invoking **kernel-mode system call**:
  1. Prepare arguments
  2. Pass control to kernel
  3. Validate arguments
  4. Perform operation
  5. Return to user-mode

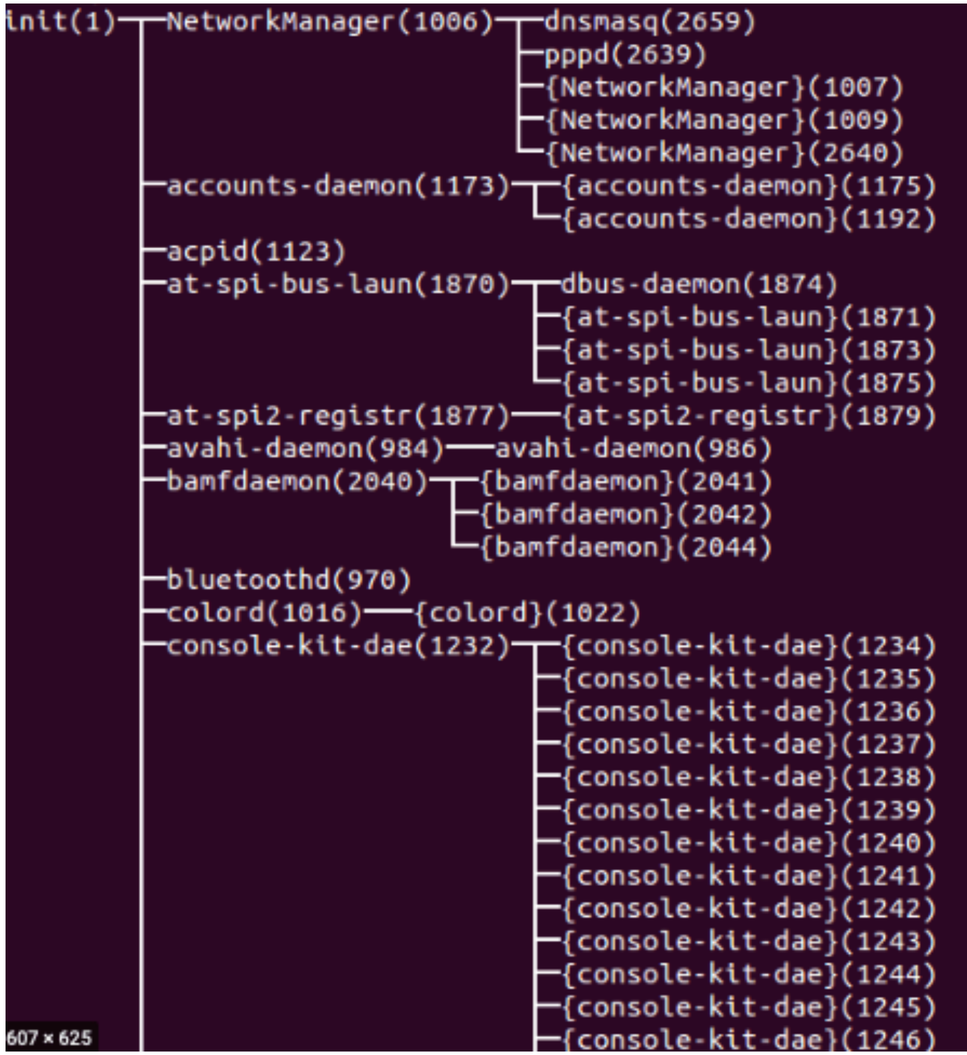


Processes

- The **kernel** **defines a** **process**: an instance of a program that is executing
- Programs are initially stored in non-volatile memory (e.g., disk)
- To be executed, they have to be loaded into memory and receive a process ID
- Each process must execute in a context in which it has access to a set of **resources**, which must be available independently of other processes
- The **frontier between** **processes** is a trust boundary: processes have to be confined/isolated in-between them (separation of privilege 🕋)

Processes

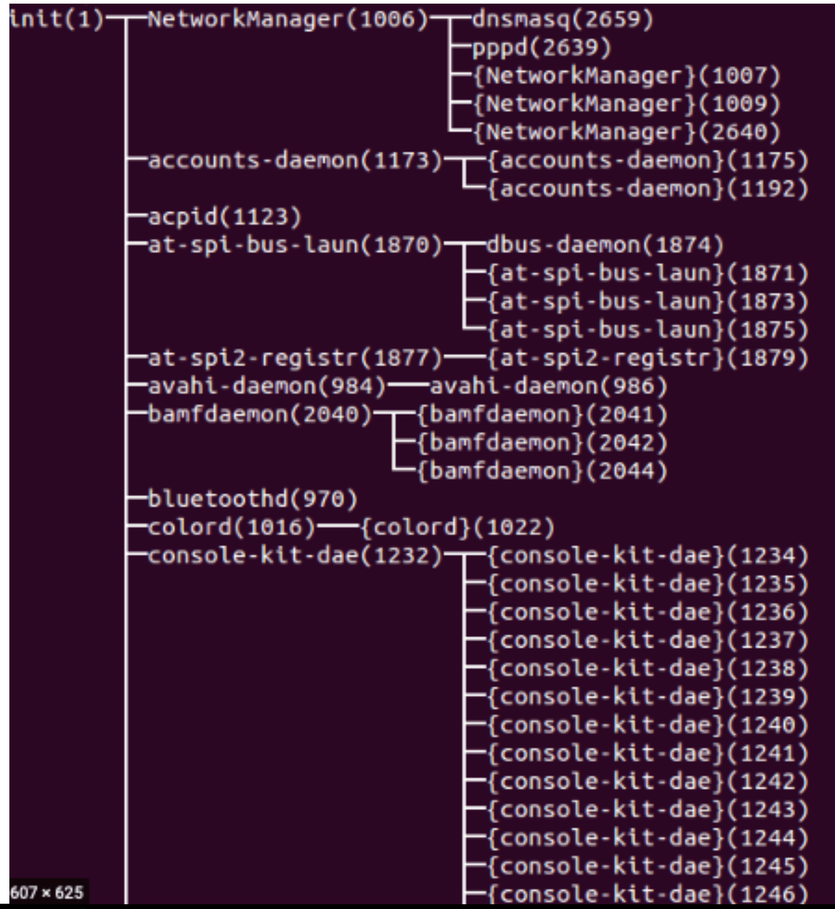
- For each **process**, the **kernel** mediates access to **resources**:
  - Assigns a reasonable fraction of processor time
  - Assigns a reasonable size of memory space
  - Grants access to other resources via system calls
- There is a set of base processes that interact with users:
  - When the user launches an application, an OS process (e.g., shell, GUI) **forks** a new process
  - OS manages an **hierarchy of processes**; descendants inherit their creator's privileges



Linux pstree

Processes (Linux)

- `pstree` lists the process tree
- The root process is called `init` (PID 1)
- PID is a unique process ID
- The **permissions of a process** depend on **who creates it**:
  - Each user has a UID (unique user ID) and a GID (unique group ID)
  - The UID 0 is typically reserved for the super-user (root)
  - Processes are associated to user UIDs / GIDs

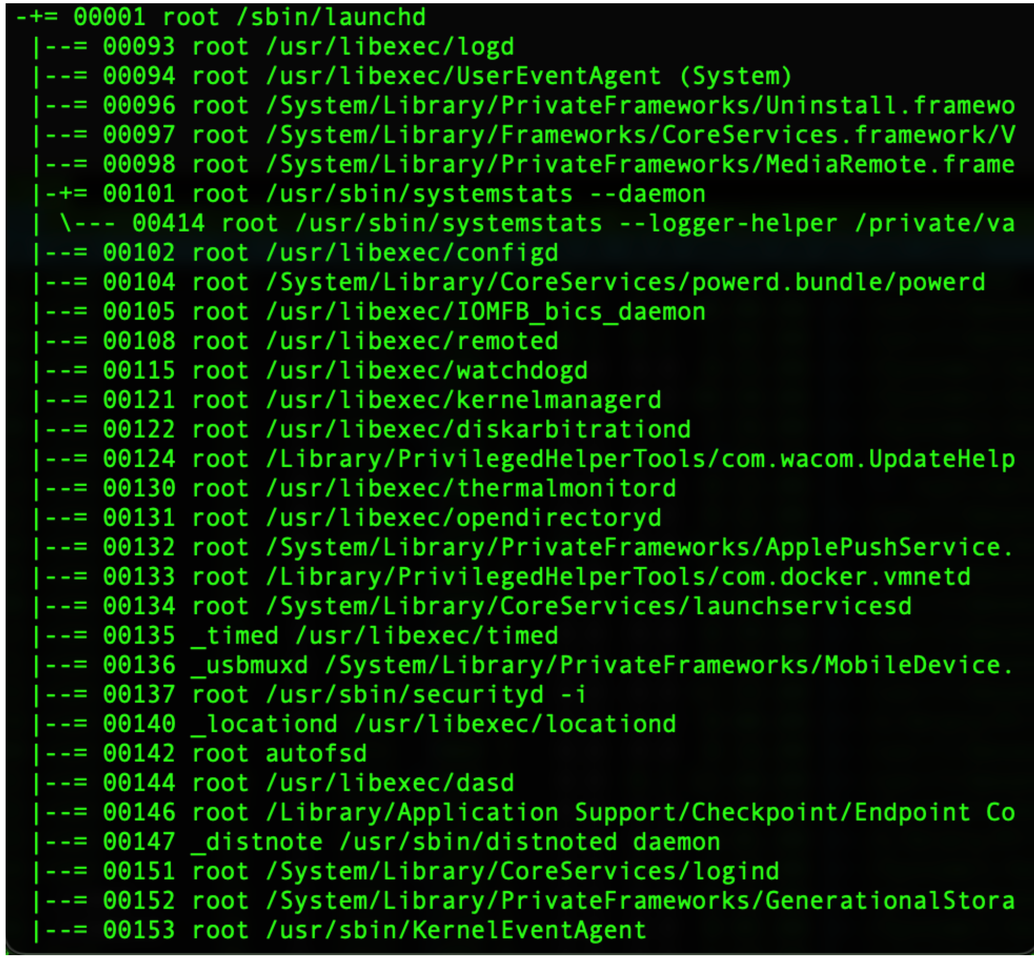



```
tuts@fosslinux:~$ sudo cat /etc/passwd
[sudo] password for tuts:
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/n
ologin
```

```
tuts@fosslinux:~$ id fosslinux_admin
uid=1001(fosslinux_admin) gid=1001(fosslinux_admin) groups=1001(fosslinux_a
dmin)
tuts@fosslinux:~$ id tuts
uid=1000(tuts) gid=1000(tuts) groups=1000(tuts),4(adm),24(cdrom),27(sudo),3
0(dip),46(plugdev),116(lpadmin),126(sambashare),129(ubridge),130(libvirt)
tuts@fosslinux:~$
```

Processes

- Often need **Inter-Process Communication (IPC)** ⇒ **system calls**
  - File system
  - Shared memory
  - Synchronous messages: pipes, sockets
  - Asynchronous messages: signals

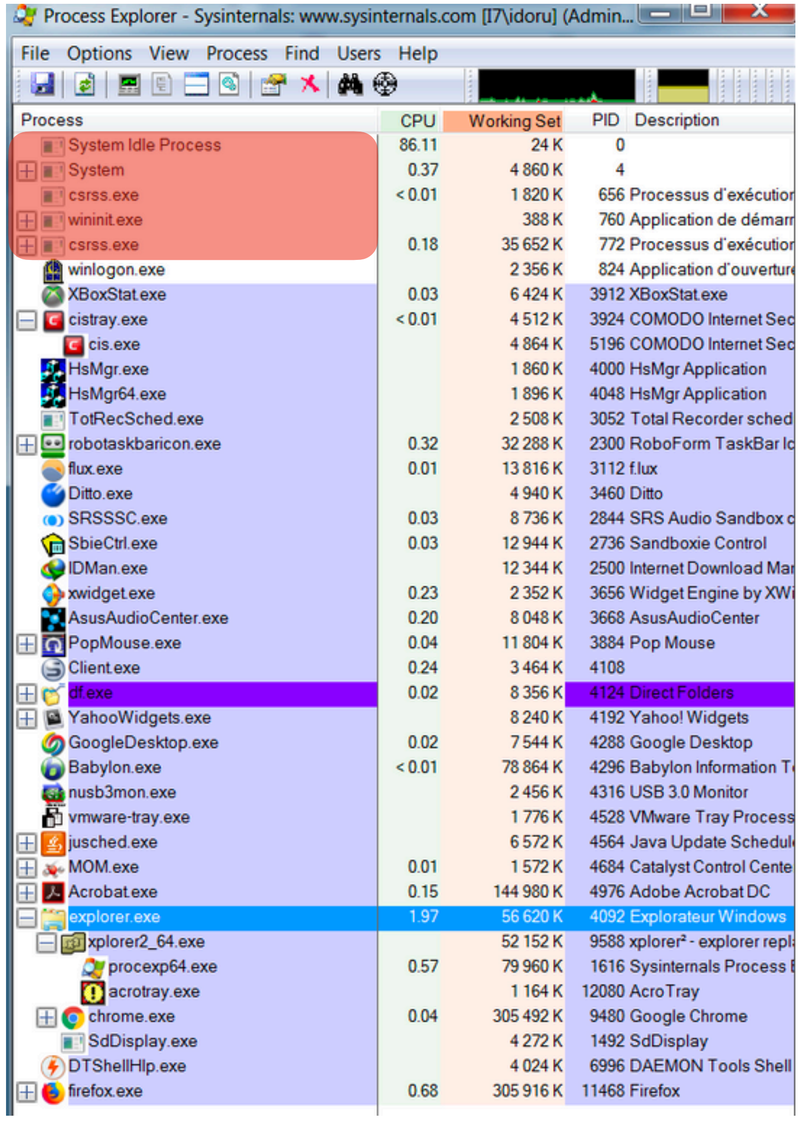


Mac pstree

Processes

- **Daemons, services:**
  - Processes that are not (directly) visible to the user
  - E.g., indexing, remote login, printers, file synching
  - Generally launched before the processes that interact with the user
  - Typically execute with higher privileges than the users and survive user sessions

```
tuts@fosslinux:~$ sudo cat /etc/passwd
[sudo] password for tuts:
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/n
ologin
```



Windows Process Explorer

Trust Model

- The trust placed in processes is **inductive**:
  - The code stored in the computer (namely the BIOS, bootloader and kernel) after a fresh OS installation is **“trusted"** (fail-safe defaults 🕌)
  - The **boot process** uses this code to place the kernel in memory and pass control to it, **preserving the “trusted” state** in memory
  - The kernel launches processes with permissions such that **no new process can change the “trusted” state**
  - E.g., hibernation processes must **preserve the “trusted” state**
  - System administrators may change the installed software and its permissions, but must guarantee that any update **preserves the “trusted” state**

Blaster Worm (2003)

- Affected Windows 2000 and Windows XP without SP2 update
- Chinese group Xfocus (allegedly) **reverse engineered the Windows XP SP2** patch
- infection ⇒ **buffer overflow making core service crash** ⇒ OS shutdown
- So disseminated that infected almost instantaneously any vulnerable OS connected to the Internet

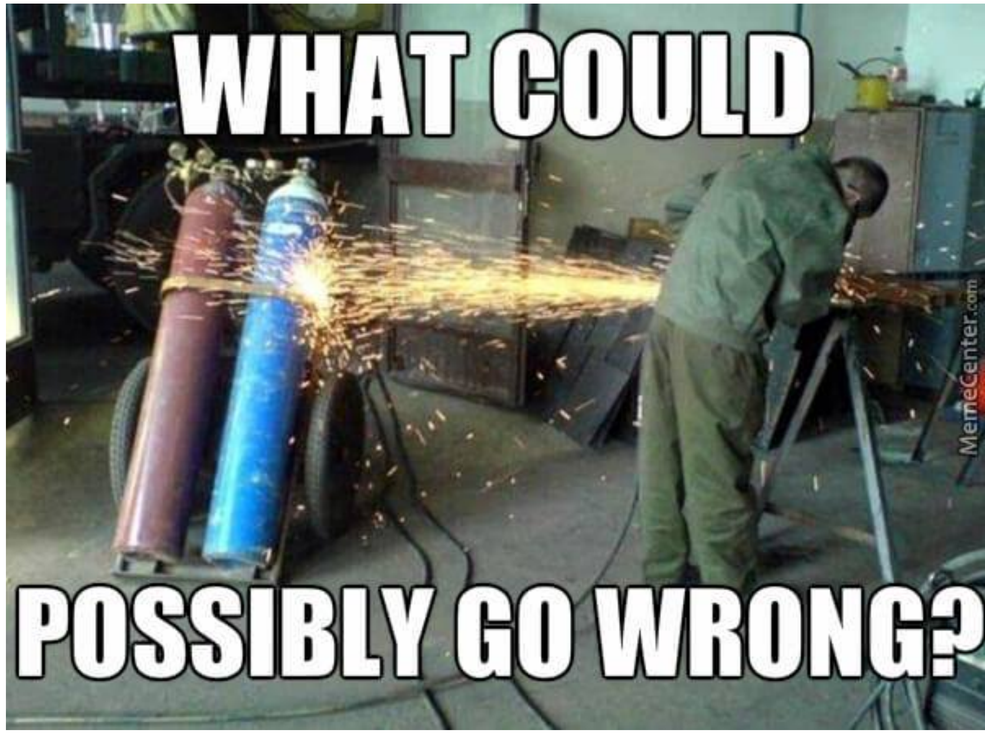
[https://en.wikipedia.org/wiki/Blaster_(computer_worm)]

Trust Model

- What does **“trusted”** mean?
  - **The system does exactly (and only) what it was specified to do**
  - E.g., does not transmit our sensitive information to the outside world without explicit authorisation
  - E.g., guarantees that our communications are established with the entities that we intend to communicate with (e.g., Google servers)
  - E.g., encrypts all the information stored in disk and cleans the memory when the system shuts down

Threat Model

- There are **attacks at all boot levels** 😈:
  - Corrupted BIOS
  - Corrupted hibernation files
  - Corrupted bootloader
  - Cold boot attacks (side-channels)

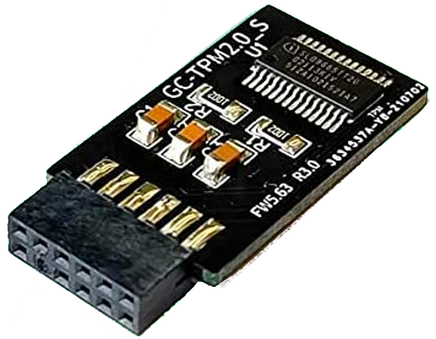


- Vulnerabilities that **affect the implementation of the boot mechanisms can completely bypass** the system's **trust model** (the boot process is the anchor)

https://learn.microsoft.com/en-us/windows/security/threat-protection/security-policy-settings/shutdown-clear-virtual-memory-pagefile

How much do we trust the OS?

- Trusted Platform Module (TPM)
  - stores credentials, encryption keys and other system sensitive data
  - A cryptographic hardware module
- UEFI Secure Boot (**Root-of-Trust**)
  - BIOS only boots **firmware components digitally signed by the supplier** of the motherboard



Windows 11 and Secure Boot

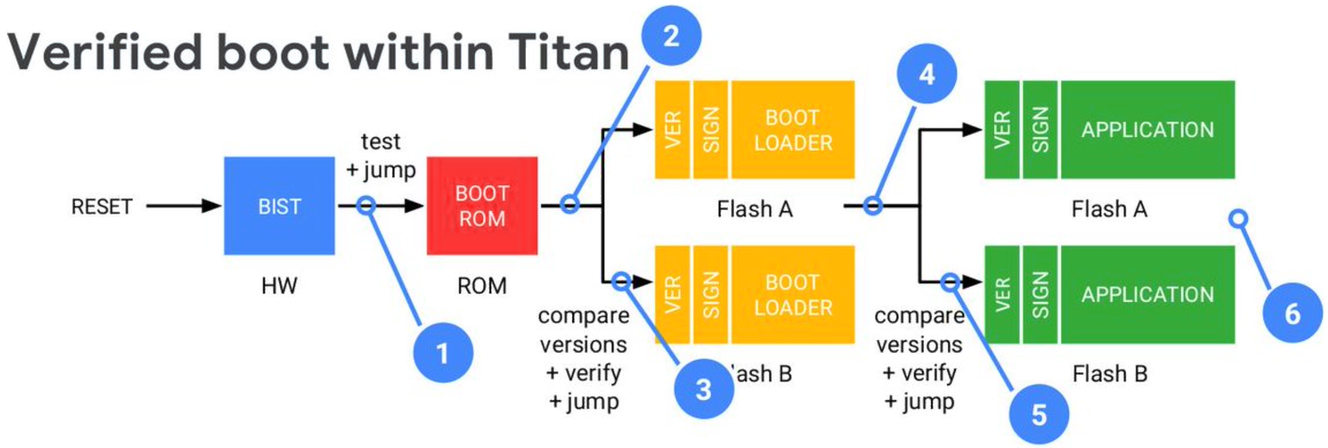
Published August 2021

This article is intended for users who are not able to upgrade to Windows 11 because their PC is not currently Secure Boot capable. If you are unfamiliar with this level of technical detail, we recommend that you consult your PC manufacturer's support information for more instructions specific to your device.

Secure Boot is an important security feature designed to prevent malicious software from loading when your PC starts up (boots). Most modern PCs are capable of Secure Boot, but in some instances, there may be settings that cause the PC to appear to not be capable of Secure Boot. These settings can be changed in the PC firmware. Firmware, often called BIOS (Basic Input/Output System), is the software that starts up before Windows when you first turn on your PC.

How much do we trust the Hardware?

- Google OpenTitan (2019 - …): open-source project to create a more trusted chip
- Open Design 🕎 ⇒ audit the HW for backdoors and security vulnerabilities
- Avoids supply-chain attacks 
- **Root-of-Trust**:
  - **Cryptographically ensure that the chip has not been modified**
  - between BIOS and processor
- Gives a trusted base for the OS



1. Test logic (LBIST) and ROM (MBIST); if fail ⇒ stay in reset; else jump to ROM
2. Compare bootloader (BL) versions A + B; choose most recent
3. Verify BL signature; if fail, retry with other BL; if fail, freeze
4. Compare firmware application (FW) versions A + B; choose most recent
5. Verify FW signature; if fail, retry with other FW; if fail, freeze
6. Execute successfully verified FW

https://opentitan.org/

OS Countermeasures

- Most security problems arise from administration errors
- Malicious processes (malware) can corrupt the OS trust model
- **Monitoring** to detect breaches to the model (compromise recording 🕮):
  - Event logs allow detecting suspicious behaviours, e.g., the repetitive crash of a process that is trying to explore a vulnerability
  - Process monitoring tools allow to visualise the running processes, their used resources and the code that they are running
- **Mediating** the installation of code using **digital signatures** hardens the system against malware (complete mediation 🕮 + fail-safe defaults 🕮)

OS Countermeasures

- We will study the security mechanisms used by the OS to guarantee **isolation** between processes
  - Prevents a user with low privileges to use the system beyond what is permitted
  - Prevents a process with a vulnerability to open a door that corrupts the whole system
- Focus on fundamental traits of any OS:
  - **Memory management, processes and file system**

Virtual Memory

- The fundamental rule says that:
  - **A process cannot access the memory space of another process** of a different user
  - Confidentiality, integrity and **control flow of the kernel must be protected** from user-level processes

- How is it guaranteed?
  - **Runtime mediation**: memory accesses are mediated by HW and SW mechanisms managed by the kernel (e.g., swap partitions, memory protection keys)
  - Virtual memory that resides in disk may be vulnerable to **offline attacks**
    ⇒ **encrypted disk**

Windows evolution

- DOS: **no memory protection** (e.g., any process could write directly to the screen's memory)
- Windows 1.x/3.x/95/98/ME: single-user OSs, **some isolation between user space and kernel space** (e.g., applications running as administrator)

> Windows 9x is a series of hybrid 16/32-bit operating systems.
>
> Like most operating systems, Windows 9x consists of kernel space and user space memory. Although Windows 9x features some memory protection, it does not protect the first megabyte of memory from userland applications for compatibility reasons. This area of memory contains code critical to the functioning of the operating system, and by writing into this area of memory an application can crash or freeze the operating system. This was a source of instability as faulty applications could accidentally write into this region, potentially corrupting important operating system memory, which usually resulted in some form of system error and halt.[24]

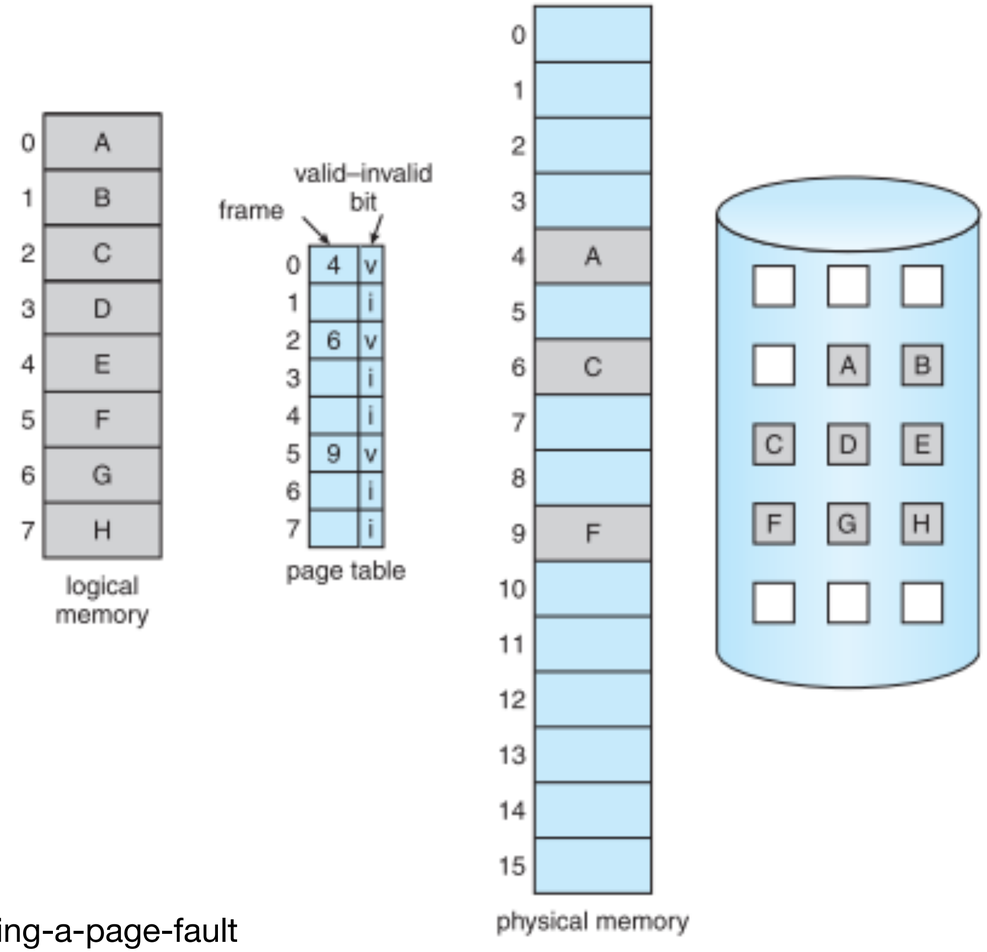
- Windows NT/2000/XP/…: multi-user SOs, **isolation between users** = *n?x systems
  - Processes in the **same user space are not isolated** (we will see containers later)

*n?x systems

- In a standard OS, **processes within the same user space are not isolated** 
- The `/proc` filesystem represents processes as files and directories within a virtual file system hierarchy. Allows peeking inside a process:
  - `/proc`: running processes
  - `/proc/{pid}/mem`: memory mapping of a specific process
  - `/proc/{pid}/mem`: reading/writing memory file with standard `fopen`, `fread`, `fwrite` functions.
  - `/proc/{pid}/stat`: dynamic statistics of the process
  - ...

Virtual Memory

- The memory space handled by the OS is **much larger than the available physical space**:
  - The processor supports virtual memory
  - The address space of a process is divided into pages
  - Some memory pages are stored in non-volatile storage
  - When a “page fault” occurs, it is necessary to switch pages



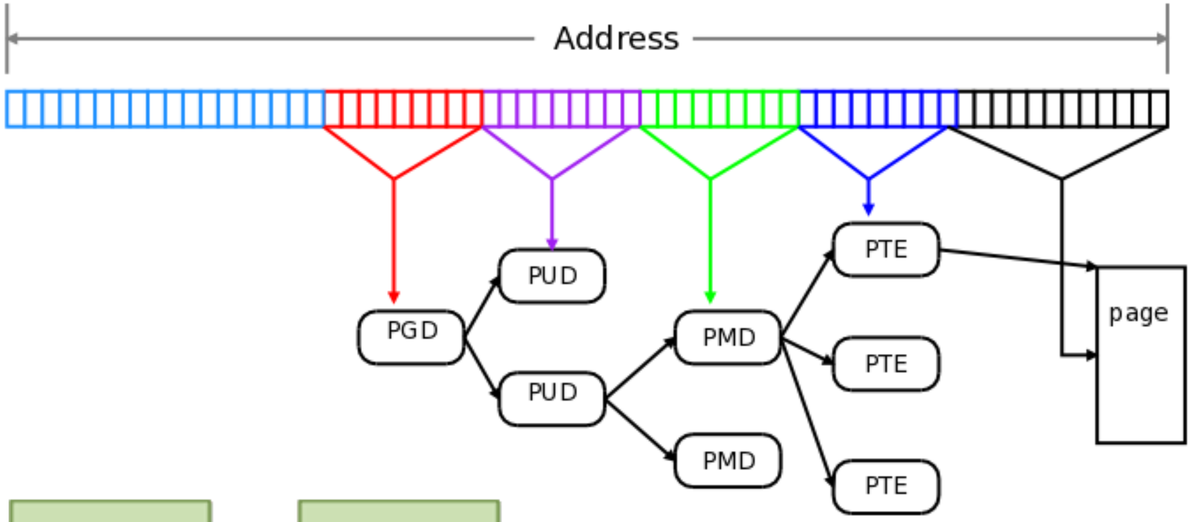
https://unix.stackexchange.com/questions/474605/does-a-pager-move-or-copy-a-page-when-handling-a-page-fault

Address Translation

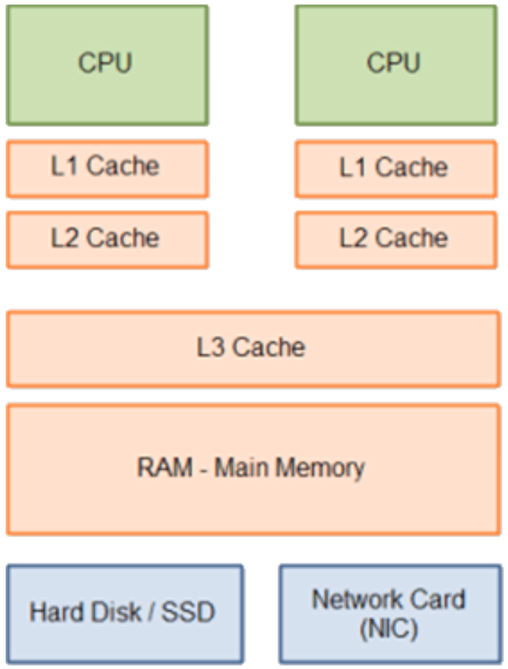
- **Address translation** is used to implement virtual memory mechanisms:
  - **Isolation**: each process accesses a memory region that does not physically exist and gives a limited vision of the resources (complete mediation 🪨)
  - **Efficiency**: includes optimisation mechanisms (caching, speculative access, paging, etc.)
- Recent years have shown that many optimisations are, in fact, real attack vectors ⇒ side-channels like Spectre and Meltdown 

Address Translation

- **Virtual memory** is split into (a hierarchy of) pages of fixed size, e.g., 4KB
- The OS stores the **physical location** of each page (in use)
- Page-table: sparse tree with information in the leaves
- Processor offers support for managing these structures



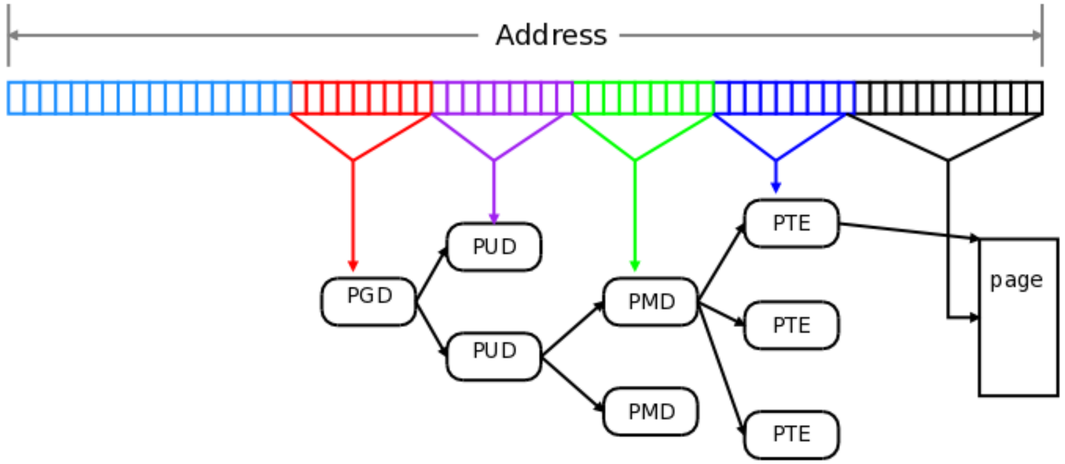
https://lwn.net/Articles/717293/



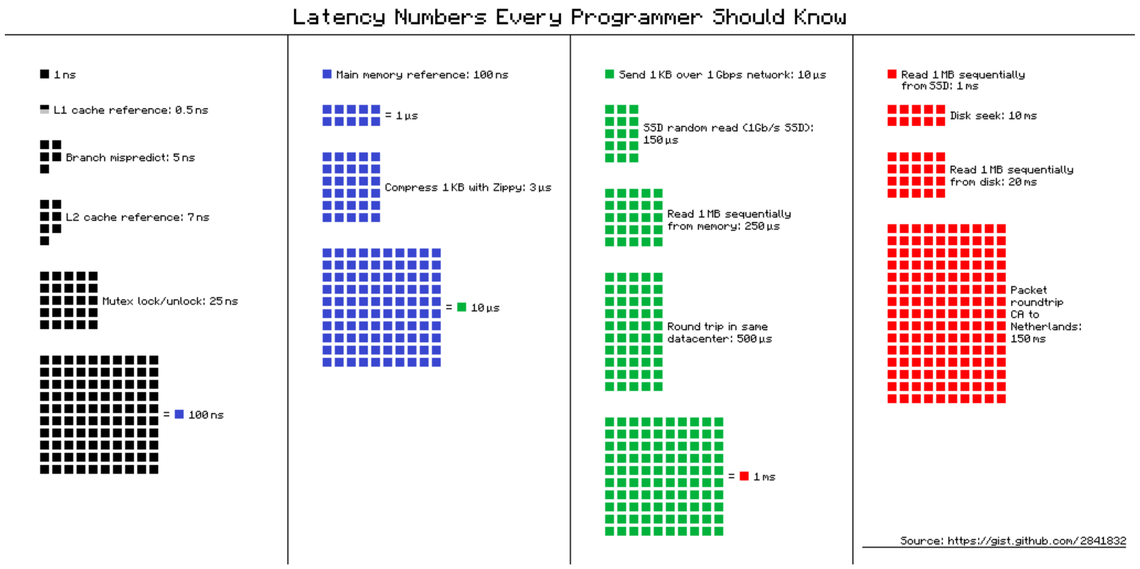
https://cpuninja.com/cpu-cache/

Address Translation

- Accessing a page table located in physical memory is **much slower**
- Translation Lookaside Buffer (TLB):
  - Cache of recently translated pages ⇒ Dirty Pipe 
  - Controlling accesses to each page (much like ACL)
  - Read/Write/eXecute (NX bit)



https://lwn.net/Articles/717293/

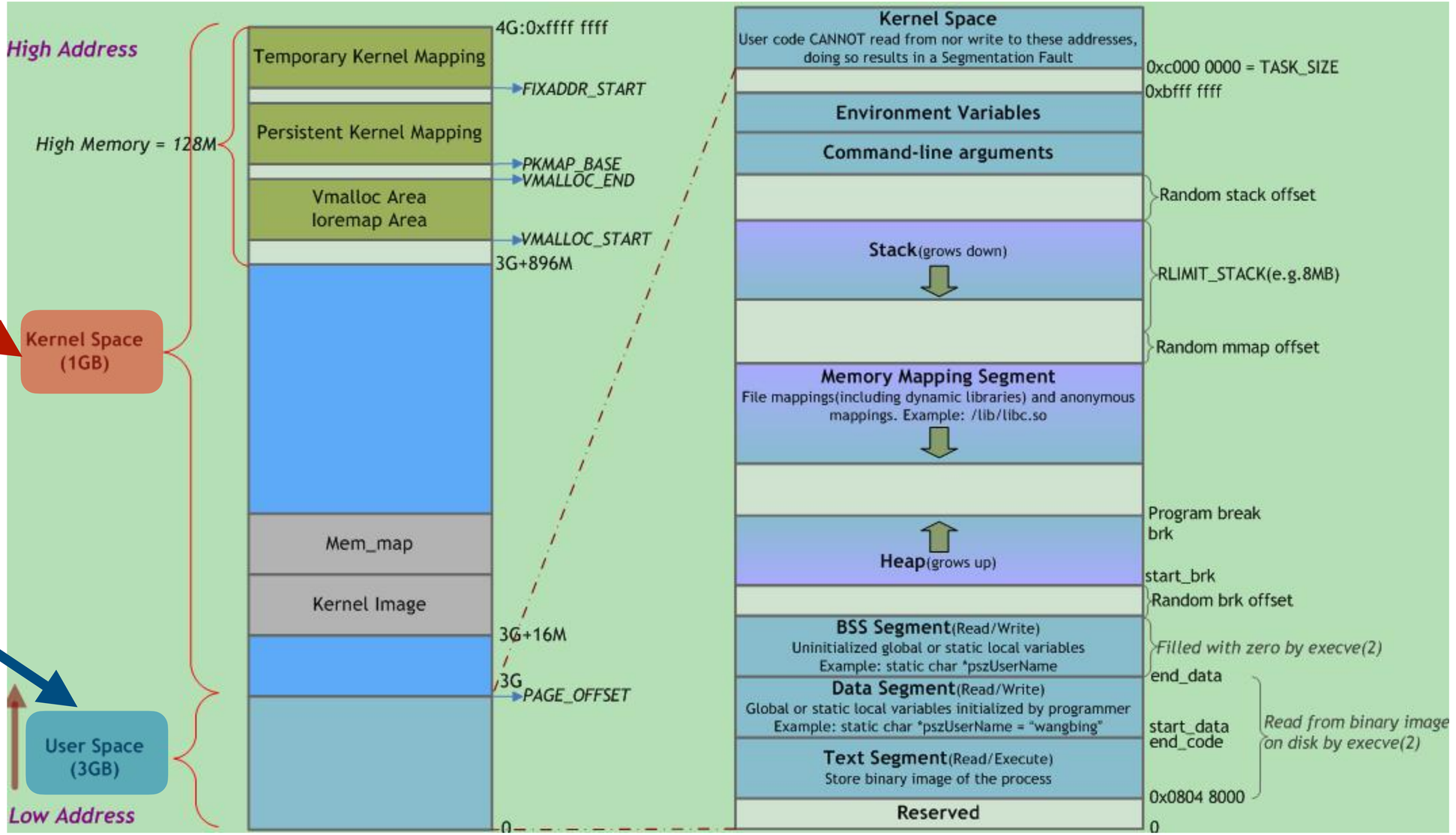


Address Translation

- How to handle system calls?
  - A completely independent management of kernel and userland address spaces would make context switches very inefficient
- **Kernel Mapping:**
  - Part of the kernel's virtual memory is mapped **directly to the virtual memory of each process**, but with different permissions
    - **User mode (UR,UW,UX)**
    - **Kernel/privileged mode (PR,PW,PX)**

Virtual Memory

- Part of the memory space if a process is **managed by the kernel**
- The **process** has no access to such space, but can **interact with it via system calls**, e.g., memory map



https://www.programmersought.com/article/1093948310/

Kernel Mapping

- When a process invokes a system call, it is not necessary to change the system's page mapping:
  - The relevant kernel memory is already mapped
  - More importantly: **the memory section of the process relevant for the system call co-exists in the same address space**
- When changing between user processes, the page tables are modified, but the ones belonging to the **kernel memory are not** ⇒ **Pegasus** 

Meltdown and Spectre 

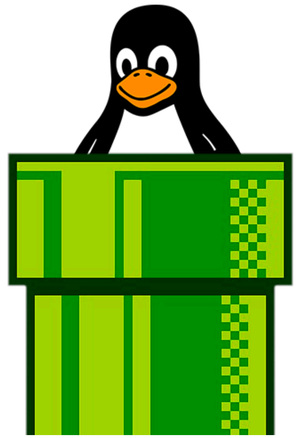
Attackers can break process isolation and read data from running programs

- Meltdown 
- Race condition between privilege checking and memory access
- CPU out-of-order execution performs/caches memory access even if process has no privileges
- Attacker will have his memory accesses rejected, but execution time will depend on cache
- Attacker can read data from any address that is mapped to the current process's memory space

- Spectre 
- CPU speculative prediction can be trained
- Attacker can execute code that tricks the CPU to misspeculate; other processes will reuse the erroneous speculative prediction
- Attacker can trick other applications into accessing arbitrary locations in his memory
- Attacker can inject speculative shellcode and observe cache/timing differences

https://meltdownattack.com/

Dirty Pipe 



- CVE-2022-0847: Attackers can overwrite any file on the system (provided read access)
- Linux Kernel:
  - Bug: creating pipes with arbitrary flags
  - Optimisation: splice() syscall quickly pushes the contents of a file into a pipe
  - Optimisation: PIPE_BUF_FLAG_CAN_MERGE kernel flag allows updating a “dirty” memory page without forcing a rewrite of the data

1. Open an arbitrary file with read-only permissions
2. Create a new pipe with the PIPE_BUF_FLAG_CAN_MERGE flag
3. Point the pipe to a section of the file
4. Write arbitrary data to the pipe, which will overwrite the cached file page

https://tryhackme.com/room/dirtypipe

Pegasus 

- Improper sanitisation of user-supplied system call arguments in `OSUnserializeBinary`
  - Intuition: XML elements with no closing tags and incorrect size declarations
- CVE-2016-4655: attackers can calculate the iOS kernel location in memory
  1. Kernel reading the malformed input into kernel objects overflows into sensitive kernel memory
  2. Attacker can infer the base kernel location
- CVE-2016-4655: memory corruption leads to Jailbreak
  3. Kernel casting the malformed objects leads to a user-after-free
  4. Attacker can launch many threads to reduce the probability of the memory pointed to by the dangling pointer being reallocated by competing non-attacker-controlled processes
  5. Attacker can inject his malicious shellcode with ROP techniques

https://jndok.github.io/2016/10/04/pegasus-writeup/ https://info.lookout.com/rs/051-ESQ-475/images/pegasus-exploits-technical-details.pdf

Defense in Depth 

- Which permission shall the kernel have on the memory of user processes?
  - all permissions ⇒  !
- Adopt defense in depth (to prevent against a **corrupted kernel**):
  - **Not** even the kernel shall be able to **violate the W^X** rule
  - **Preventing** the kernel to **write in user memory regions** is a way to protect against **information leaks** and **malicious code execution**
  - Kernel Page-Table Isolation (KPTI): **separate user-space and kernel-space page tables (almost) entirely**
    - Performance penalty: ~5% but up to 30% depending on the workload

Acknowledgements

- This lecture's slides have been inspired by the following lectures:
  - CSE127: System Security I + System Security II
  - CS155: Isolation and Sandboxing