# Introduction

Lecture 1

# Welcome

to the *Internal Strcuture of Operating Systems* class

## You will learn, understand and experiment

- learn how operating systems work
- understand how an embedded operating system works (Tock)
- *understand how a new generation research oriented operating system works (Redox OS)*
- understand how a production grade operating system works (Linux)
- write code using the Linux and Tock architecture language that boots a computer
- experiment with building your own small operating system

## We expect

- to come to class
- ask a lot of questions

# Our team

## Lectures

- Alexandru Radovici

## Labs

- Teona Severin
- Ioana Culic
- Vlad Bădoiu
- Alexandru Vochescu

## Advisor

- Răzvan Deaconescu

# Outline

## Lectures

- 12 lectures

## Labs

- 7 labs
- will show you how to boot an x86 in Rust

## Project

- Build a minimal working OS that boots
- Build a minimal feature of the OS
  - filesystem
  - memory management
  - …
- Work in teams of 3 or 4

# Grading

| Part | Description | Points |
|---|---|---|
| Lecture tests | You will have a test at every class with subjects from the previous class. | 2p |
| Project | You will have to design and implement a hardware device. Grading will be done for the documentation, hardware design and software development. | 5p |
| Exam | You will have to take an exam during the course's last lecture. | 3p |
| **Total** | *You will need at least 4.5 points to pass the subject.* | **10p** |

# Operating System

the purpose of an OS

# Operating System

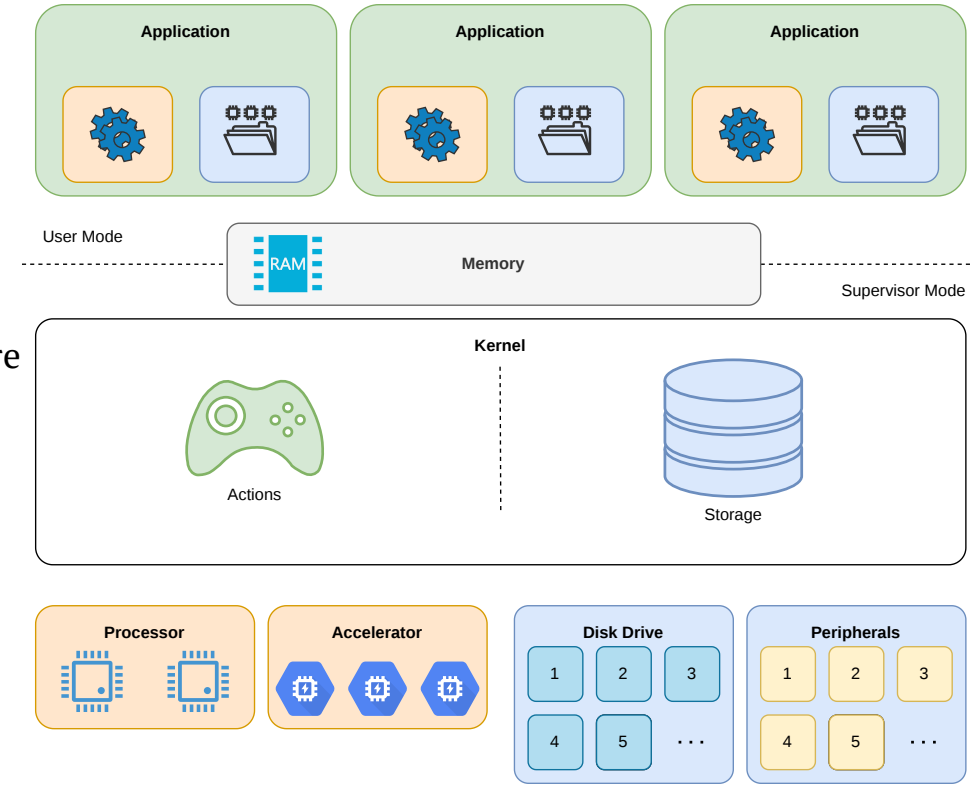
the main role

## Allow Portability

- provides a hardware independent API
- applications should run on any hardware

## Resources Management and Isolation

- allow applications to access resources
- prevent applications from accessing hardware directly
- isolate applications



# Desktop and Server Operating Systems

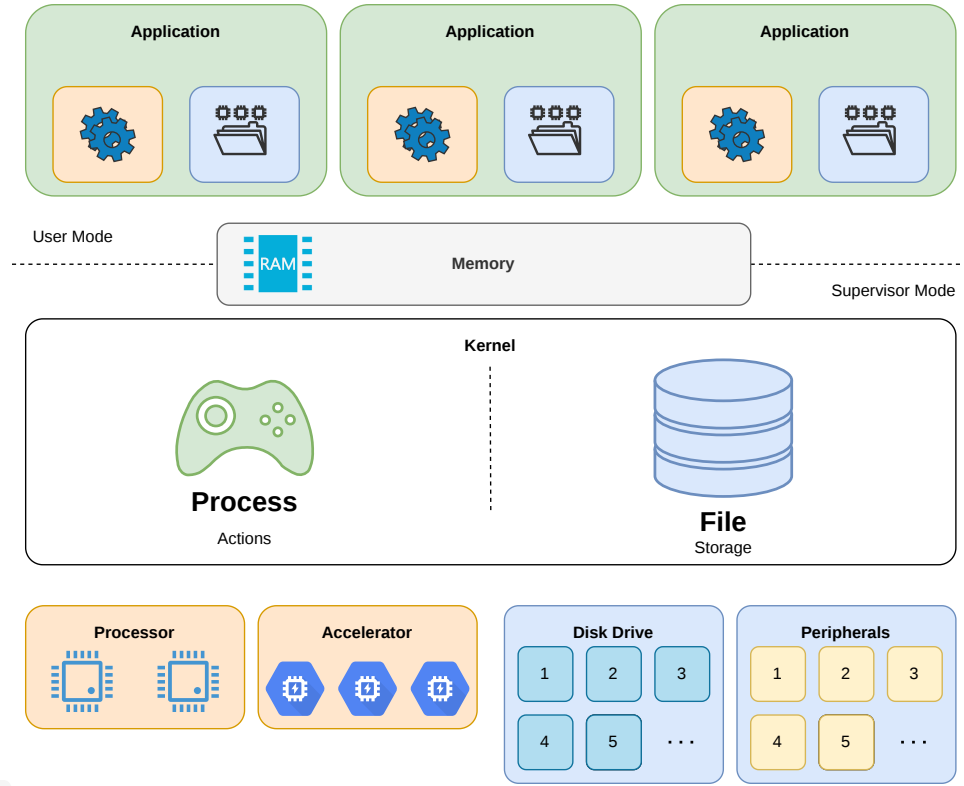
abstractions

## Actions

- **process** and **threads**
- use the *Processor* and *Accelerators* (GPU, Neural Engine, etc)

## Data

- everything is a file
- peripherals are viewed as files (*POSIX*)
  - `/sys/class/gpio/gpio5/direction`
  - `/sys/class/gpio/gpio5/value`



# Embedded Operating Systems

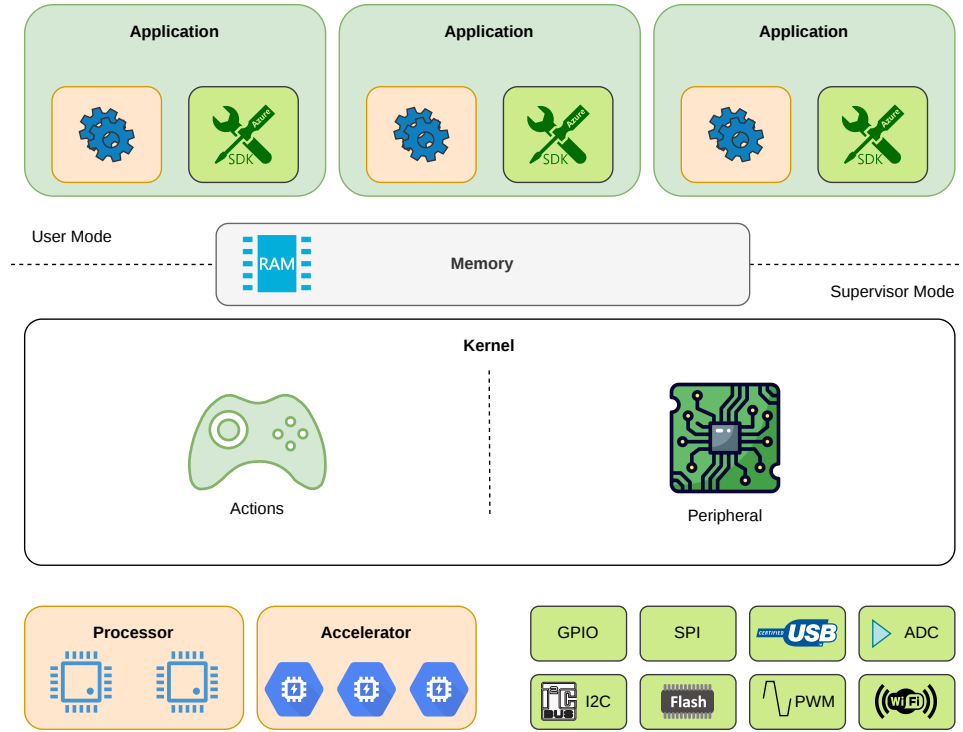
## Actions

- **process** or **threads**
- use the *Processor* and *Accelerators* (Crypto Engines, Neural Engine, etc)

## Peripheral

- provide a hardware independent API
- prevent processes from accessing the peripheral

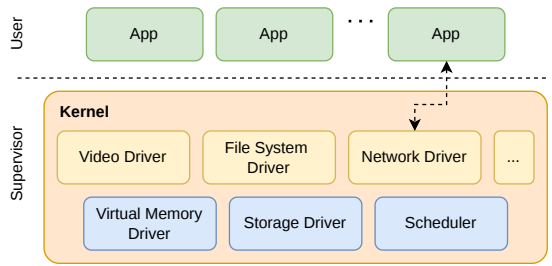
*usually* the applications and the kernel are compiled together into a **single binary**



# Kernel Types

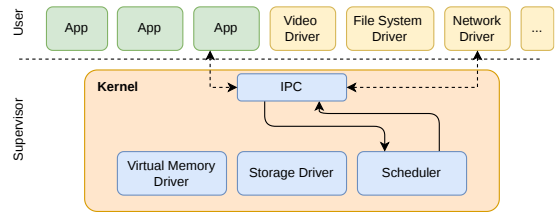
from the **kernel and drivers** point of view

## Monolothic



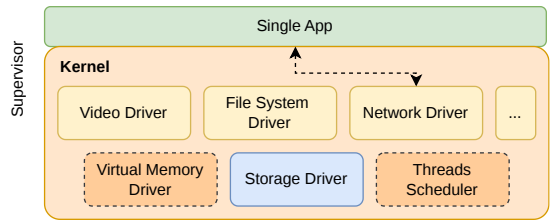

- all drivers in the kernel
- Windows, Linux, MacOS

## Microkernel




- all drivers are applications
- Minix
- Redox OS

## Unikernel




- the kernel is bundled with all the drivers and one single application
- Unikraft/Linux
- Most of the microcontroller RTOSes

# Memory Management

MMU

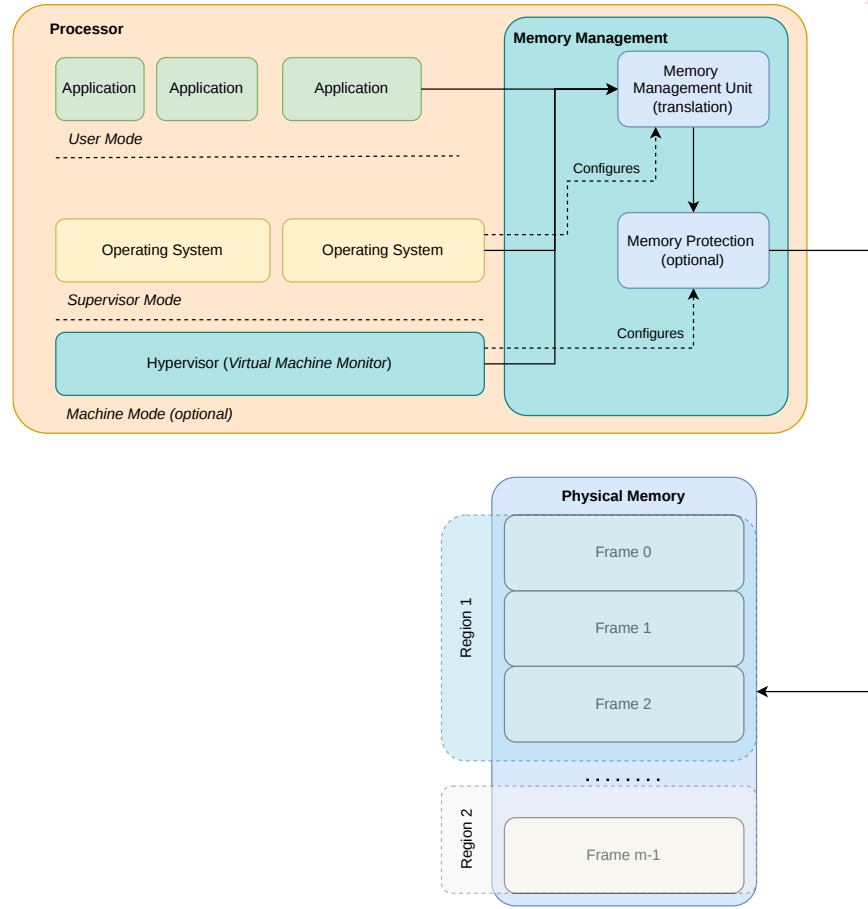
# Memory Management

memory access defined page by page

- uses *logical addresses*
- **translates** to *physical addresses*

The processor works in at least two modes:

- **supervisor** mode
  - restricts access to some registers
  - accesses virtual addresses through Memory Protection (*if machine mode exists*)
- **user** mode
  - allows only ALU and memory load and store
  - accesses memory access through the Memory Management Unit (*MMU*)



# Paging

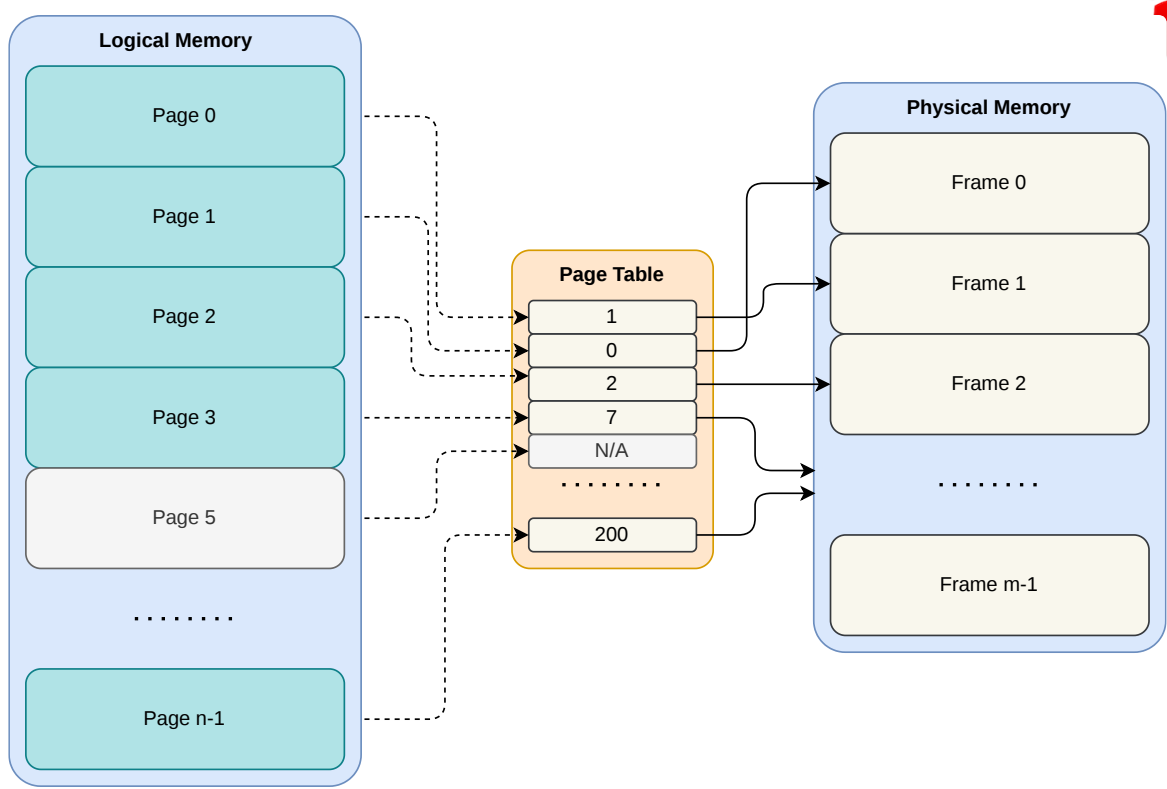
the memory *unit* is the page

- Physical Memory (*RAM*) is divided in **frames**
- Logical Memory is divided in **pages**
- *page* = *frame* = **4 KB** (usually)

*logical addresses* are translated to *physical addresses* using a **page table**

the **page table** is located in the **physical memory**

- each memory access requires at least memory 2 accesses



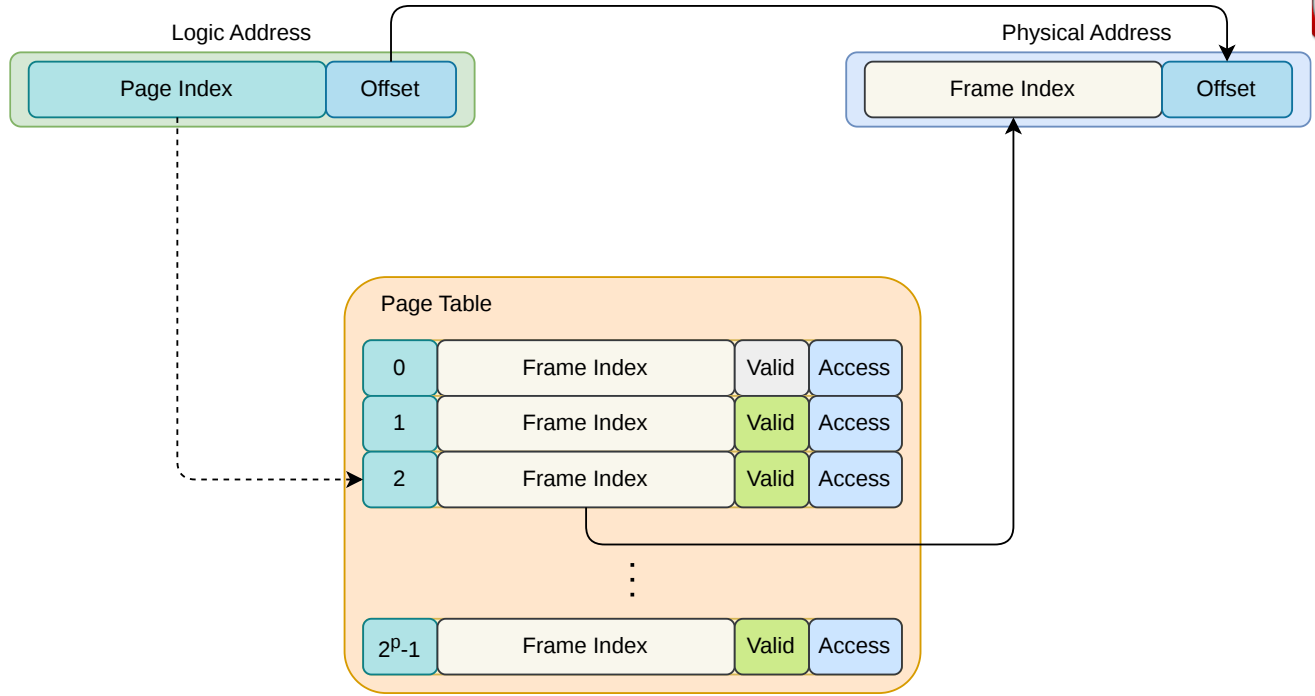
# Address Translation

page to frame

the logic address is divided in two parts:

- *page index*
- *offset* within the page

the MMU translates every logic address into a physical address using a *page table*

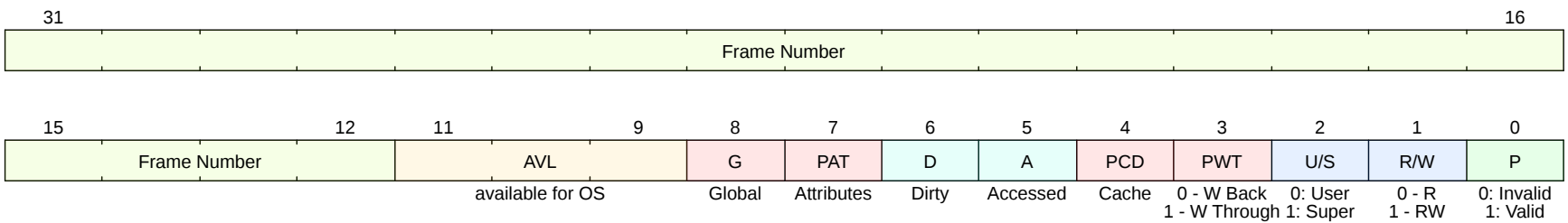


# Page Table Entry

for x86 - 32 bits

this is one entry of the page table

- **P** - is the page's frame present in RAM?
- **R/W** - read only or read write access
- **U/S** - can the page be accessed in user mode?
- **D** and **A** - has this page been written since the OS has reset these bits?
- **AVL** - bits available for the OS to use, ignored by MMU



# Address Space

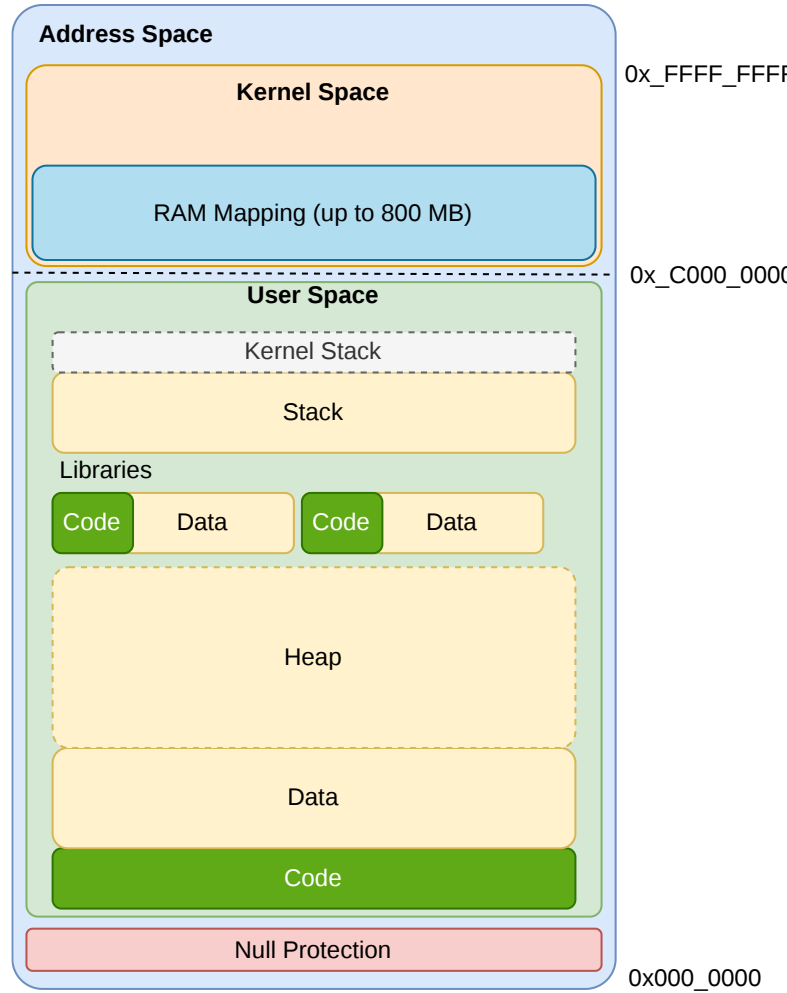
what the 32 bit x86 CPU sees when in protected mode

### Kernel Space

- accessible only in *supervisor* mode
- stores *kernel code* and *data*
- has the first 800 MB of RAM directly mapped

### User Space

- accessible in *user* mode
- stores the *process code* and *data*
- stores the libraries' *code* and *data*



# Execution

# Preemption

process vs kernel

## Process point of view

**Preemptive**

- *processes* can be suspended by the scheduler
- a misbehaving process cannot stop the system

**Cooperative**

- *processes* **cannot be suspended** by the kernel
- a misbehaving process **can stop** the system

## Kernel point of view

**Preemptive**

- *kernel jobs* can be suspended by the scheduler
- a misbehaving driver cannot stop the kernel

**Cooperative**

- *kernel jobs* **cannot be suspended** by the kernel
- a misbehaving kernel job **can stop** the system

# Scheduler Examples

| OS | Process Scheduler | Kernel Scheduler |
|---|---|---|
| Linux | preemptive | preemptive |
| Windows | preemptive | preemptive |
| macOS | preemptive | preemptive |
| Redox OS | preemptive | preemptive |
| Tock | preemptive | cooperative |

> Infinite loops in Tock driver's (capsules) will stop the system.

# System Call

the OS API

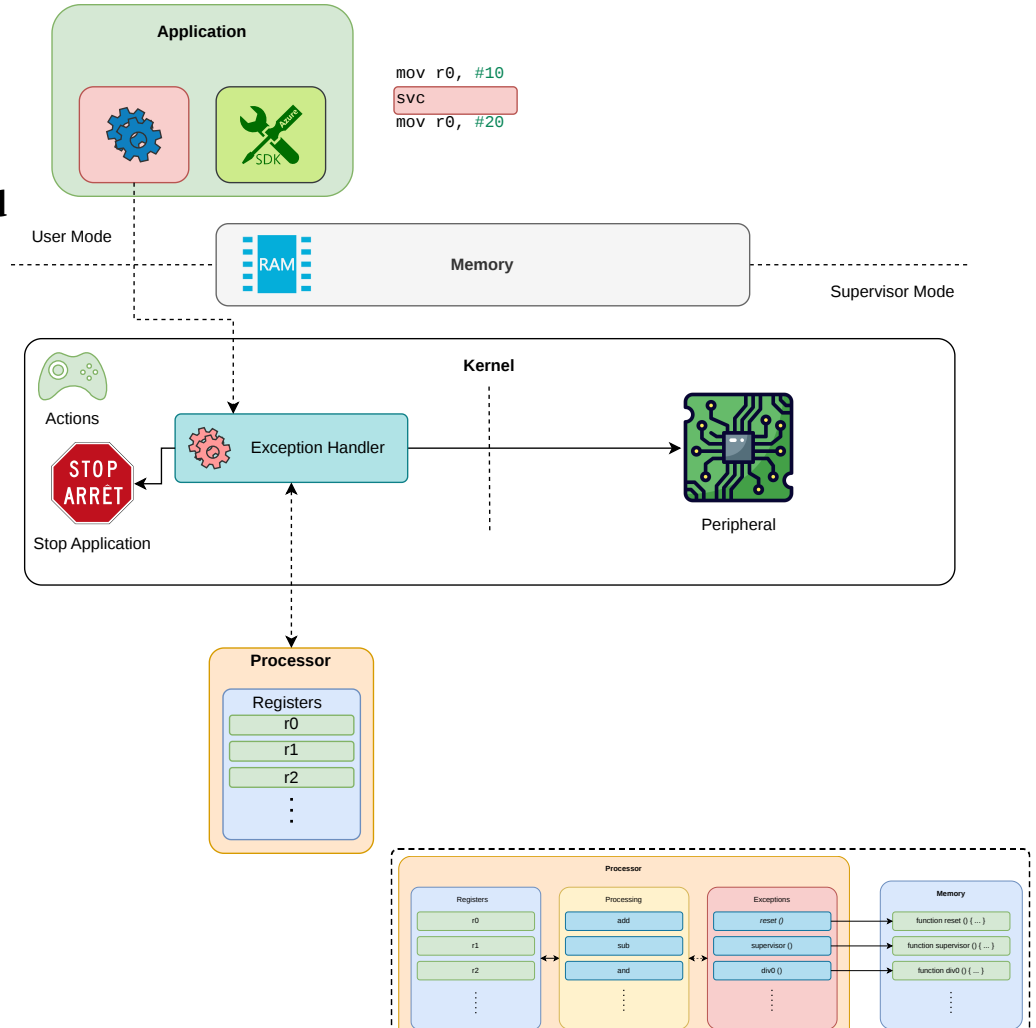
**accessing a peripheral** can be **performed** only **by the OS**

The application:

1. puts values in the registers
2. triggers an exception
   - `svc` instruction for ARM
   - `sysenter` instruction for x86

The OS:

1. looks at the registers and determines what the required action is
2. performs the action
3. puts the return values into the registers



# Execution Context

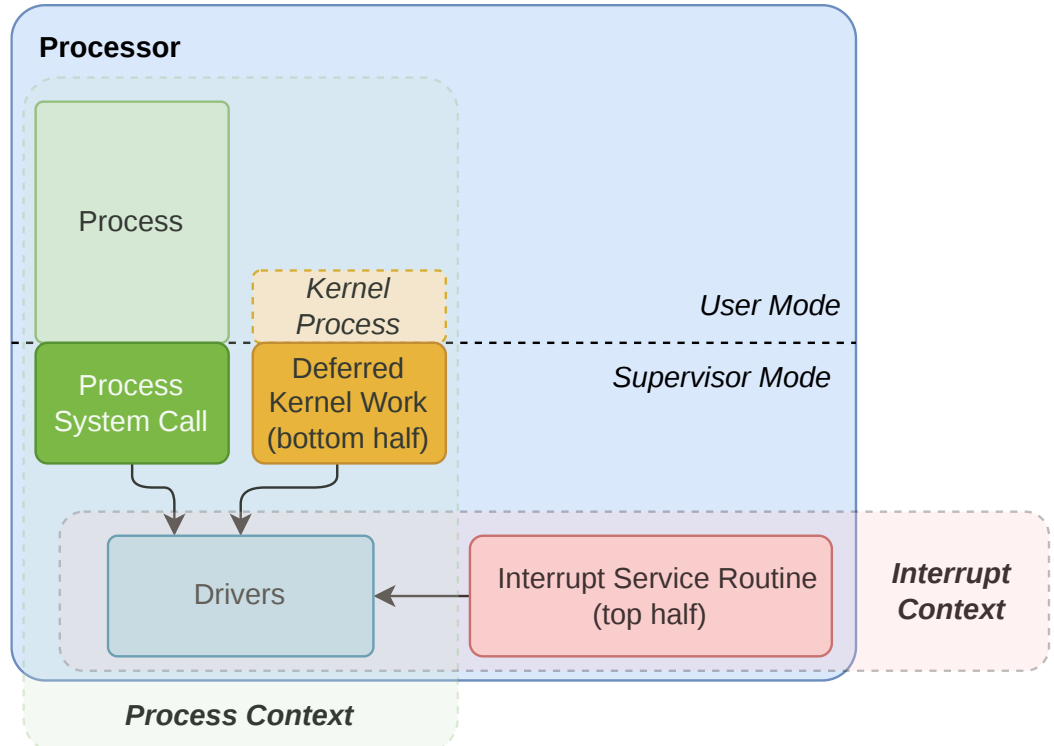
how code is executed

## Process Context

- process code (*user mode*)
- system call code (*supervisor mode*)
- kernel task code (*supervisor mode*)
  - *fake kernel process*

## Interrupt Context

- interrupt service routing
  - *supervisor* or *machine mode*
- no current process
- *should* not be interrupted

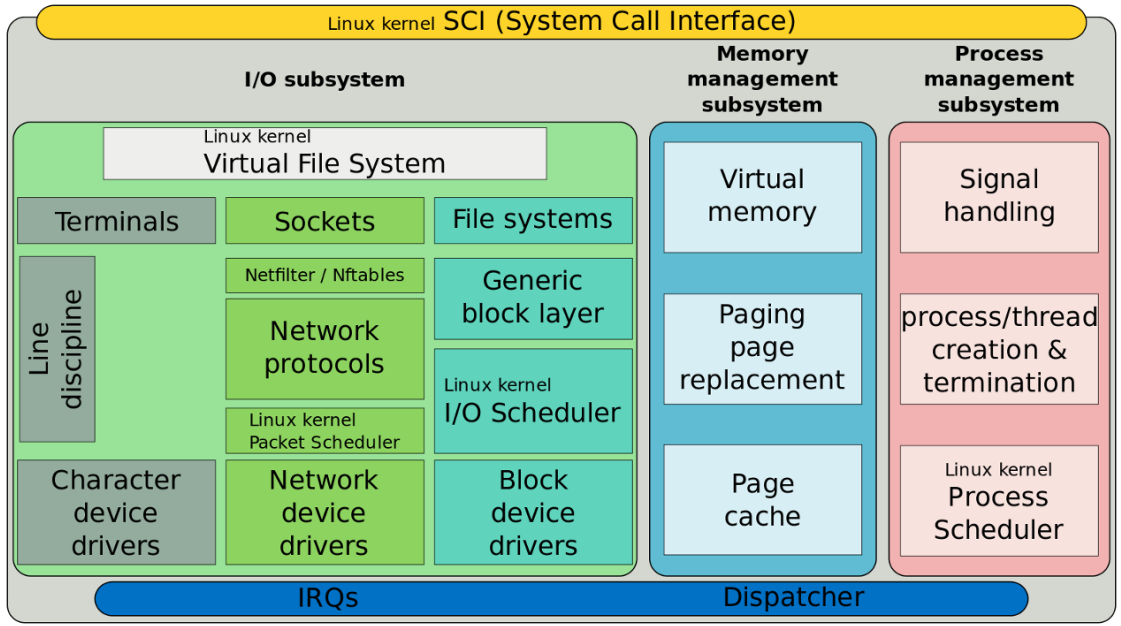


# OS Architecture

Linux and Tock

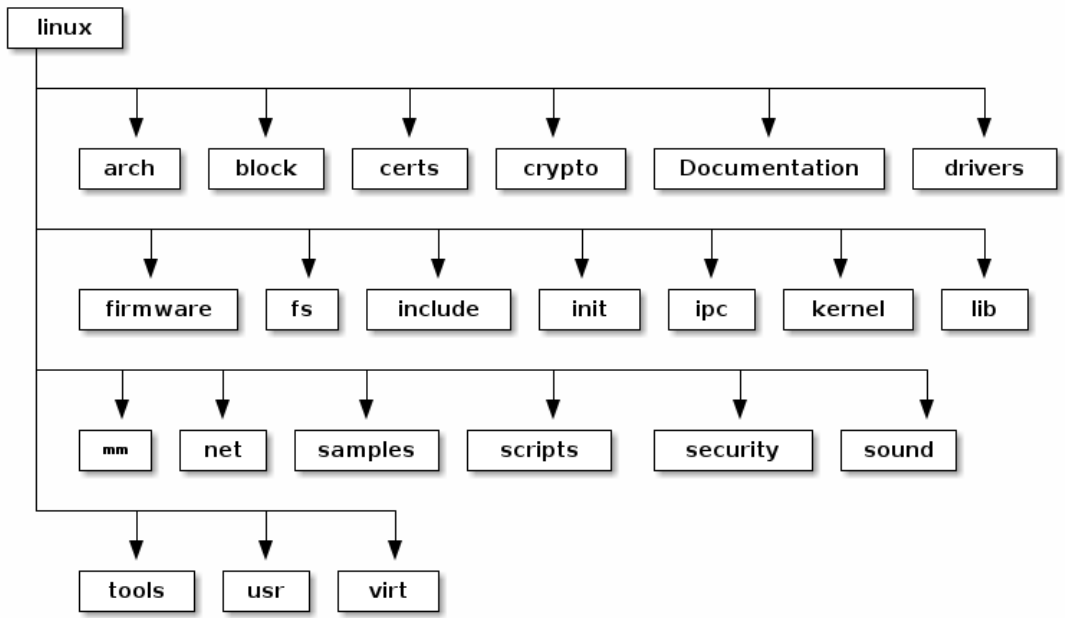
# Linux's Architecture

how a general purpose operating system works



# Linux's source tree

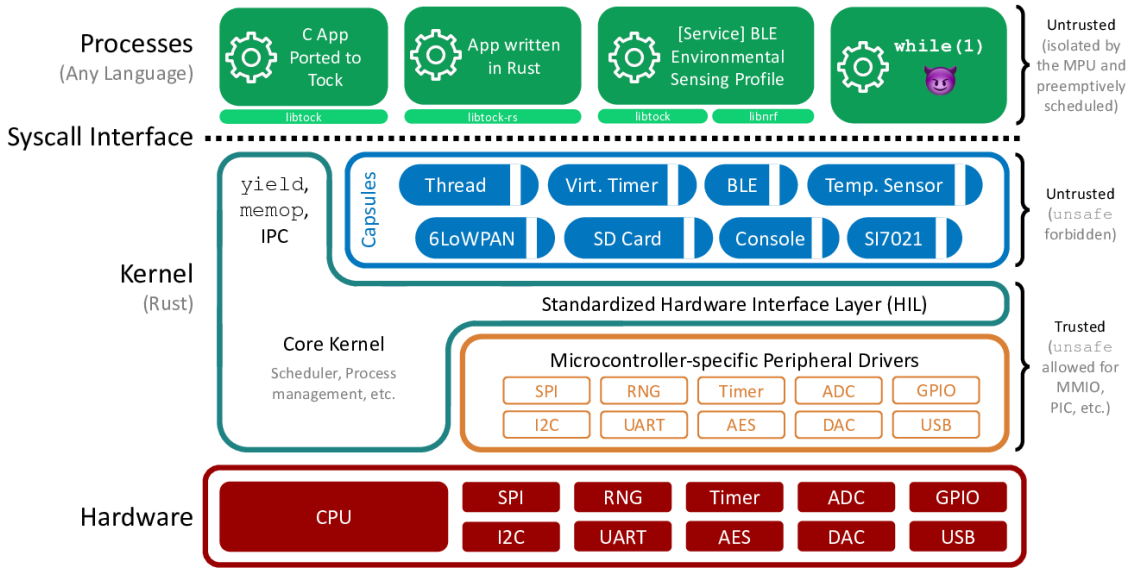
where everything is




older image, it now has a *rust* folder

# Tock's Architecture

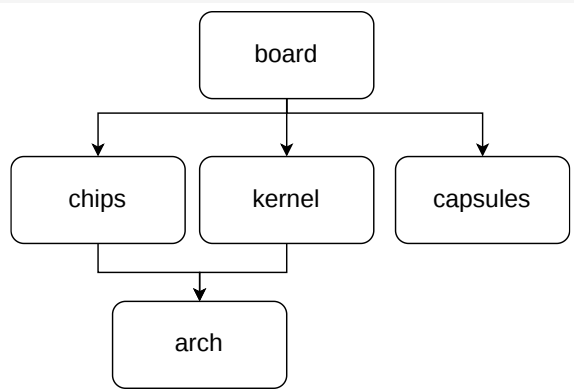
how an operating embedded system works



# Tock's source tree

where everything is

```
+-- tock            # kernel
|   +-- arch            # code specific to MCUs (ARM, RISC-V)
|   +-- boards          # code specific to boards (STM32F412G Discovery Kit)
|   +-- capsules        # drivers
|   +-- chips           # code specific to MCUs (STM32F412G, E310, )
|   +-- doc             # documentation
|   +-- kernel          # actual kernel (scheduler, ipc, memory)
|   +-- libraries       # libraries used by all the source code
|   +-- tools           # scripts for testing on other tools
|   +-- vagrant         # VM setup (different from ours)
```



# Conclusion

we talked about

- Types of OS kernel
- Preemptive and cooperative kernel
- Memory management and address space
- Execution Contexts
- Linux and Tock architecture