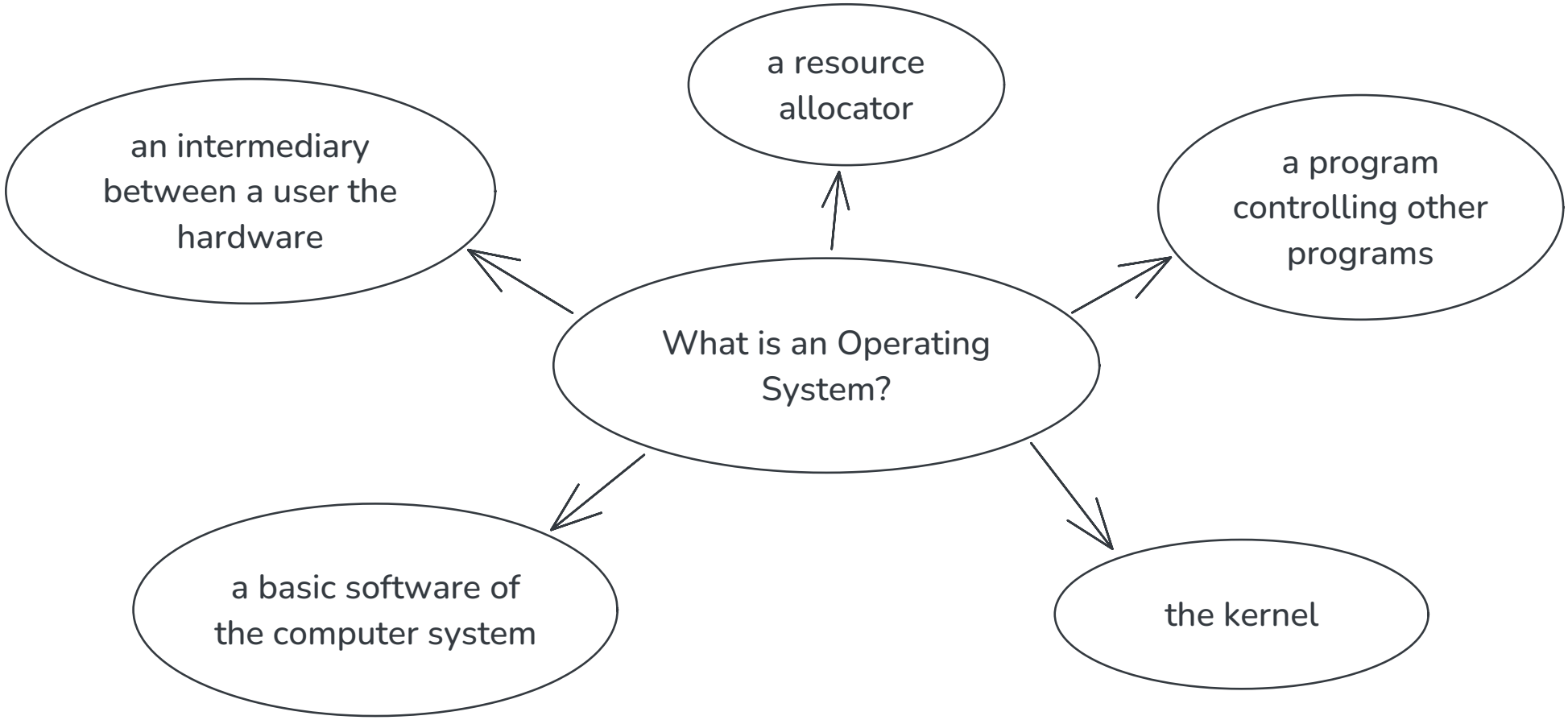
Lecture 1 - Introduction

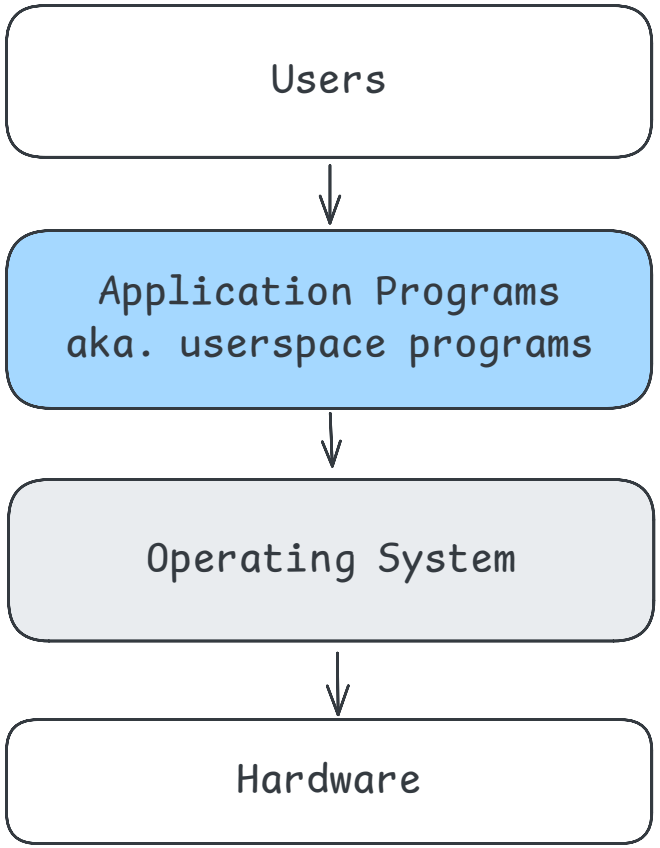
Operating Systems 1

Warsaw University of Technology - Faculty of Mathematics and Information Science

There's no clear definition of the Operating System



Layers of the computer system

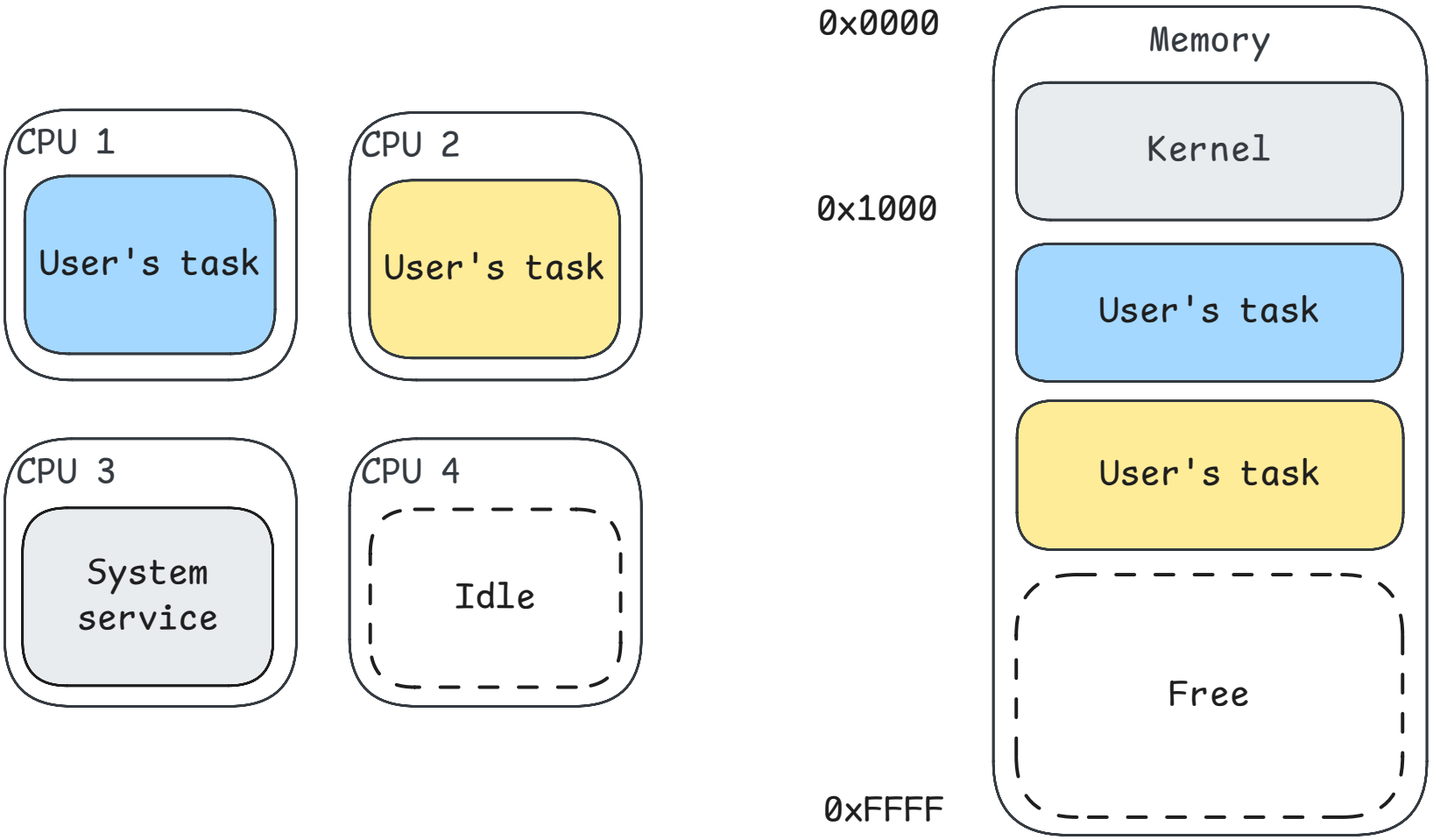


Users are running and using various programs running on the system

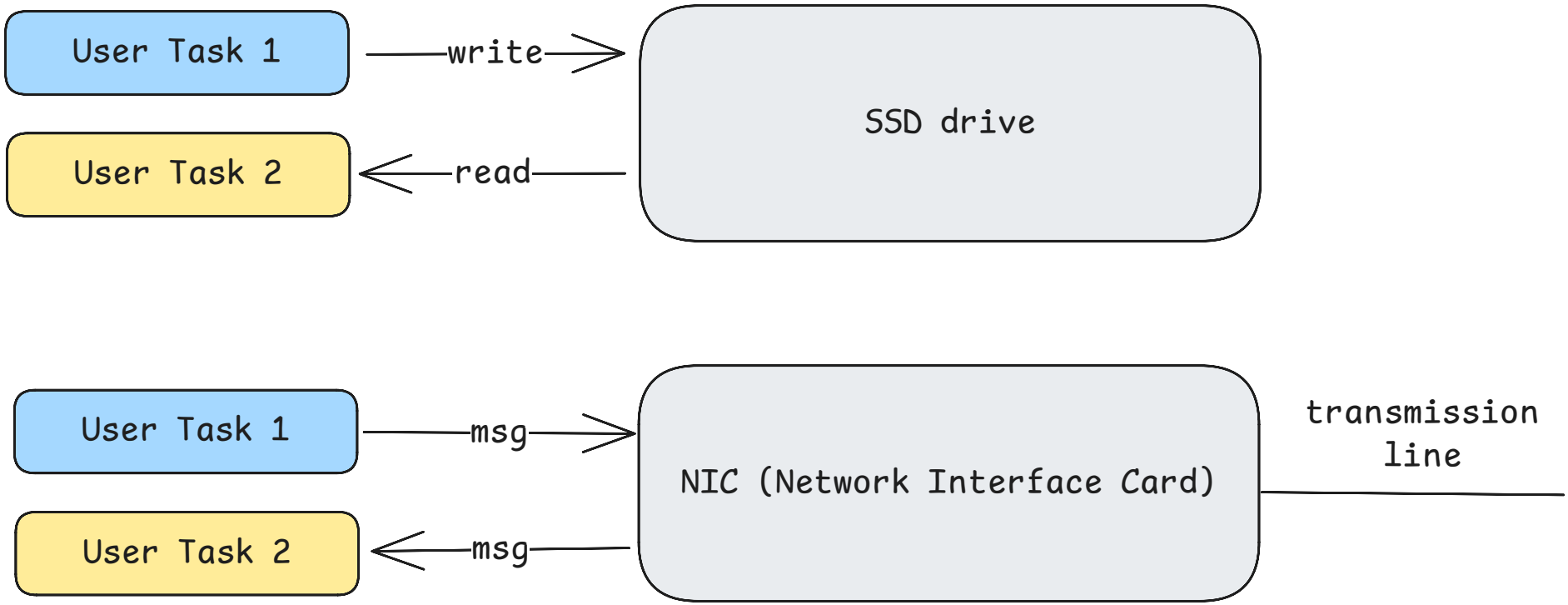
Programs are using services and interfaces exposed by the operating system

Operating system uses available hardware resources directly to do what is asked by the programs

Hardware resources managed by the OS



Hardware resources managed by the OS



Logical resources managed by the OS

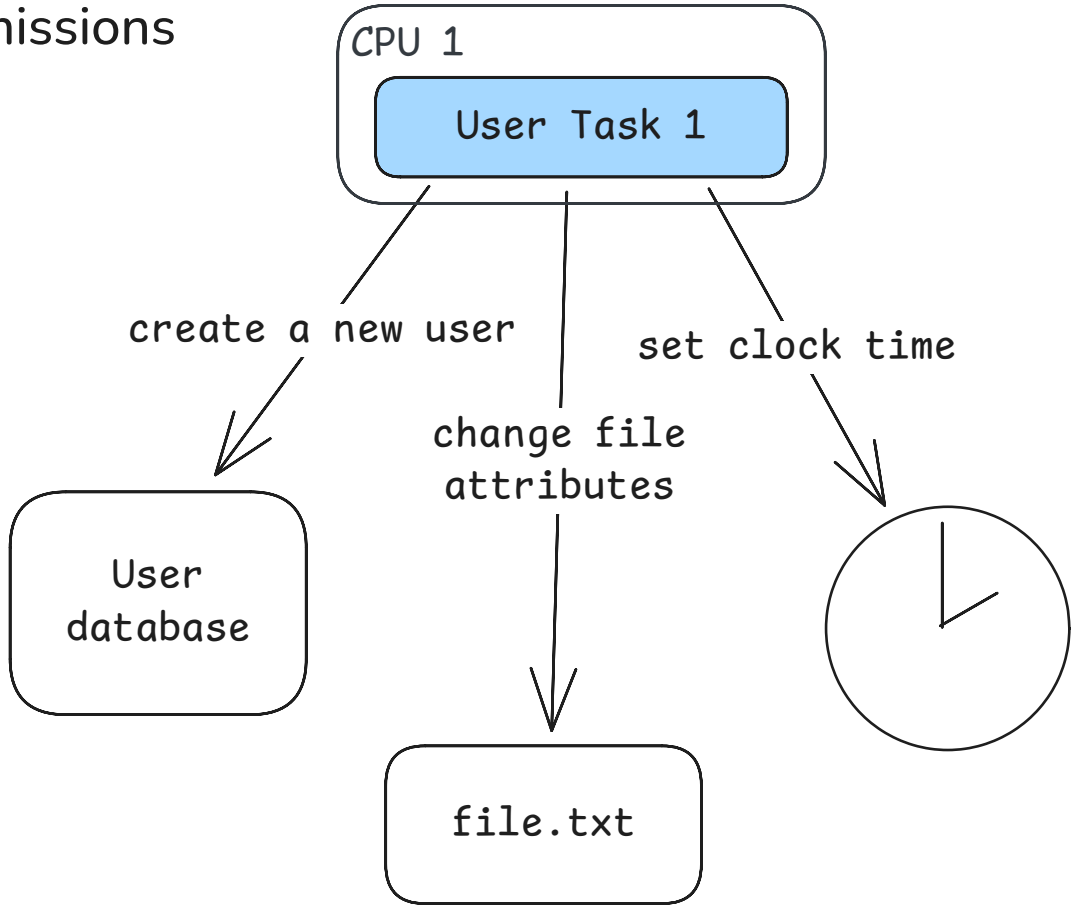
Users, user groups, credentials, permissions

Files, file attributes

Clocks, timers

System configuration

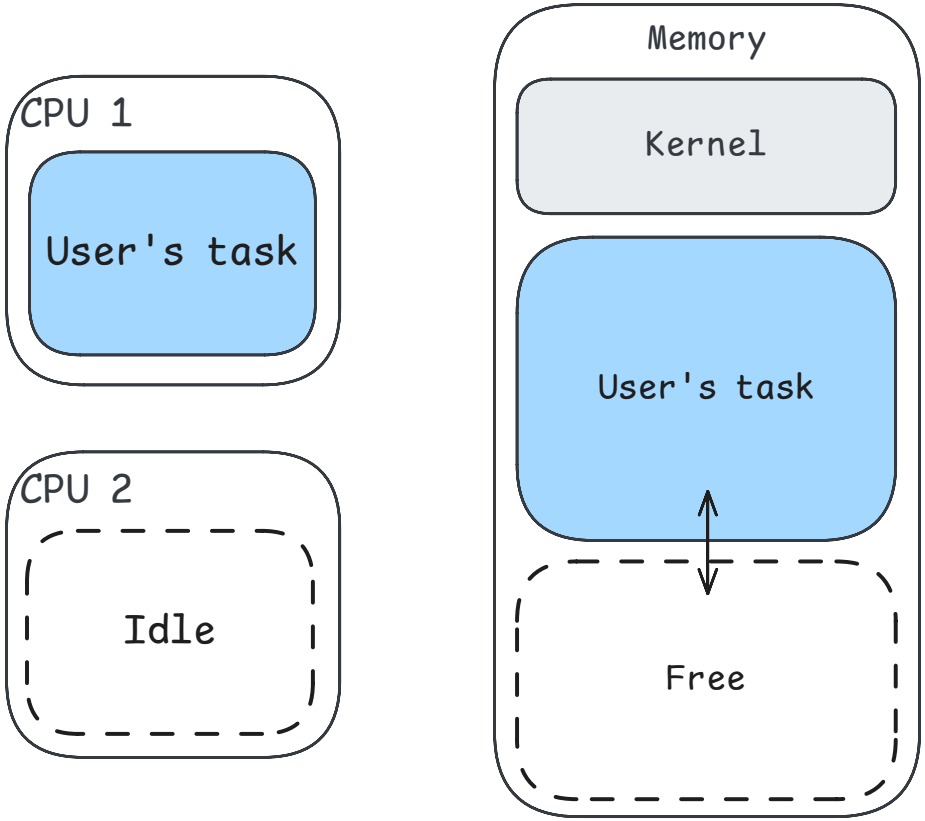
...



Executing just a single task at a time is fairly simple for the OS (and wasteful)

A task might not need all the hardware

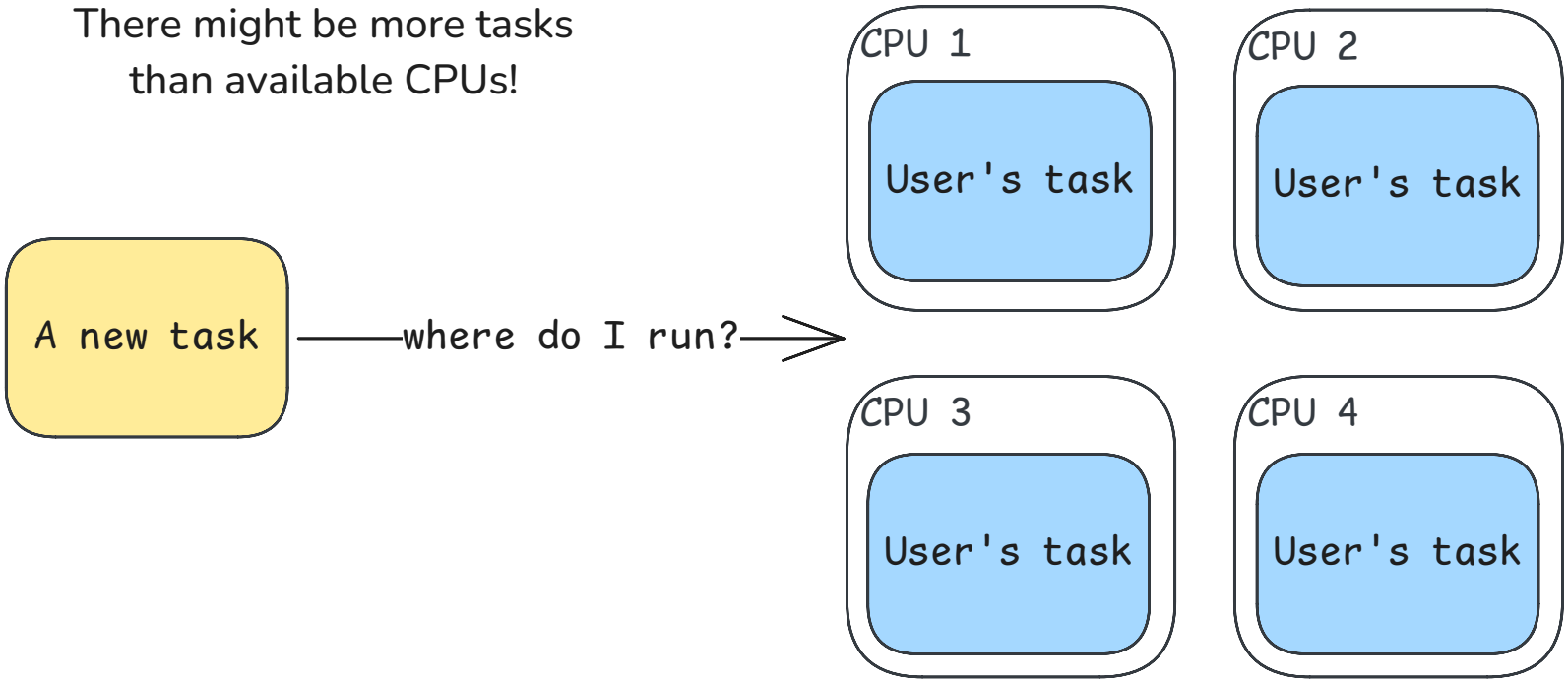
It may need to wait for I/O like:
- user input
- incoming network data
- data from a slow HDD



Allowing to run multiple tasks concurrently raises multiple issues

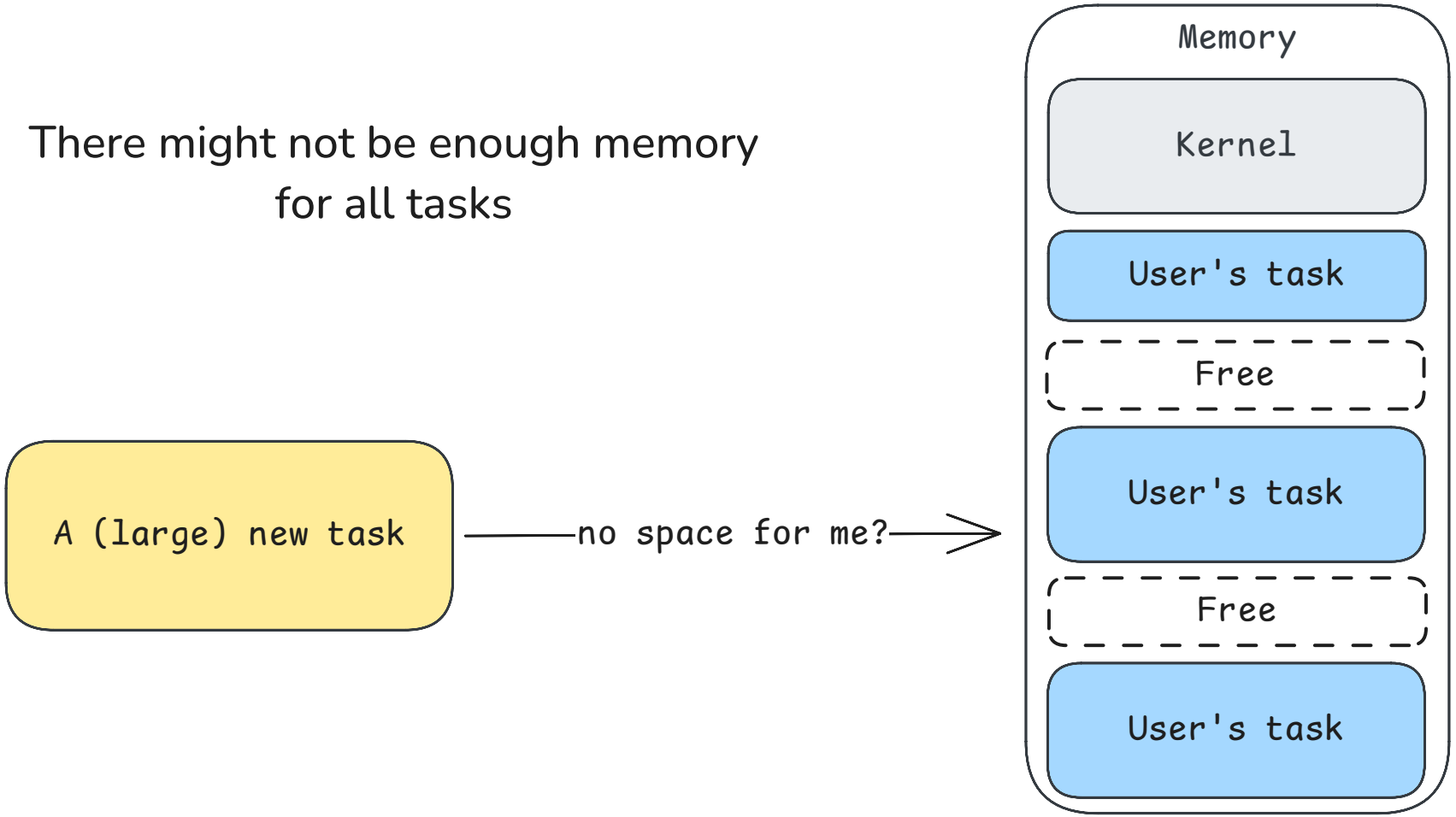
multiprogramming

There might be more tasks
than available CPUs!



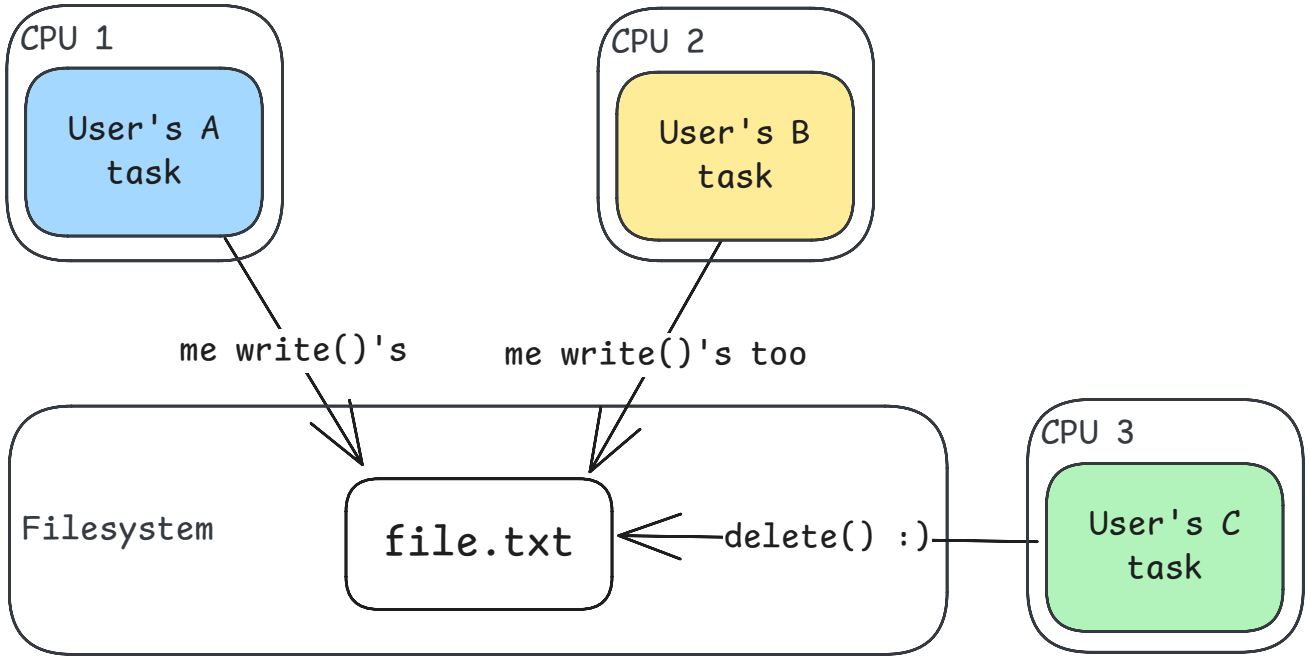
the OS has to provide job scheduling

There might not be enough memory for all tasks



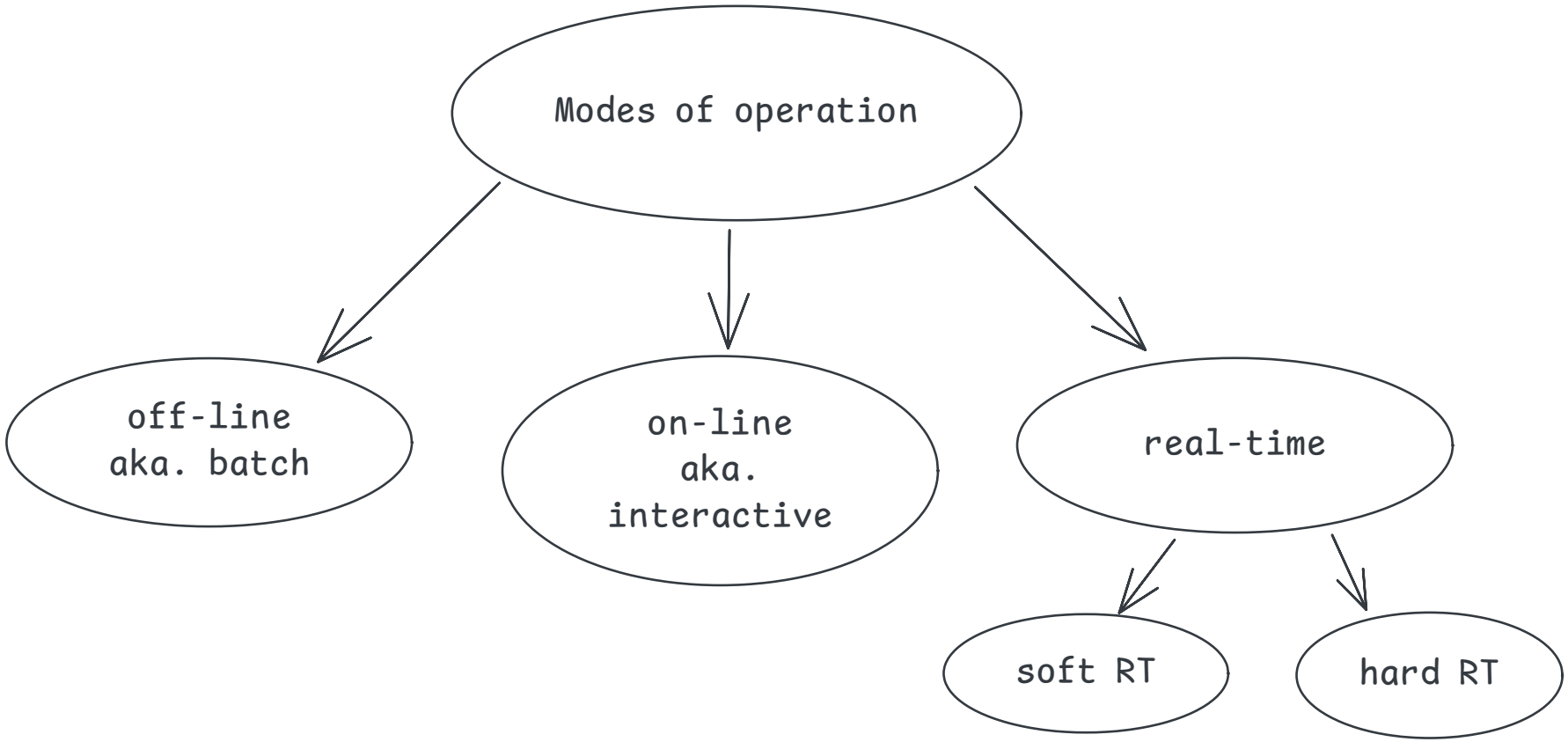
the OS has to provide main memory management

Tasks might try to share resources,
in a conflicting way

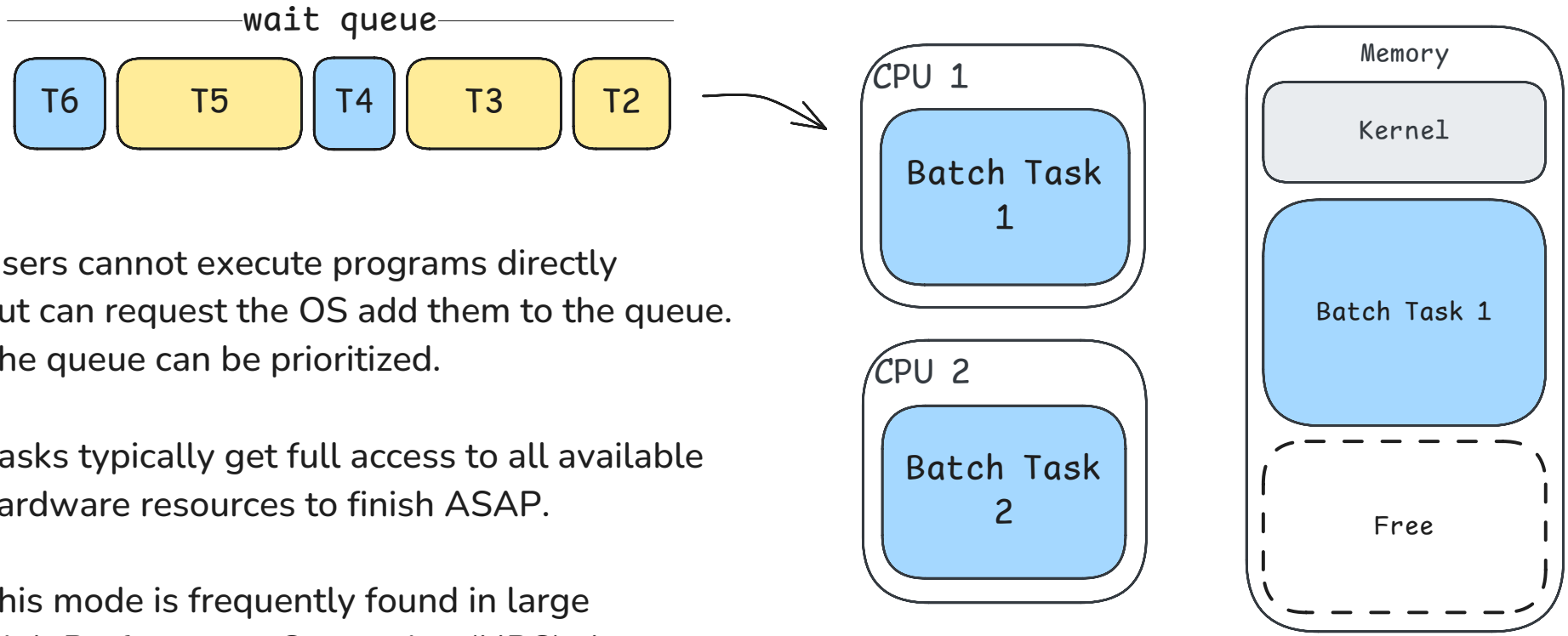


the OS has to protect all shared resources and provide
a set of safe system calls to access them

Users request multiple tasks - it's up to the OS how to execute them



In batch mode jobs are queued and executed one-by-one

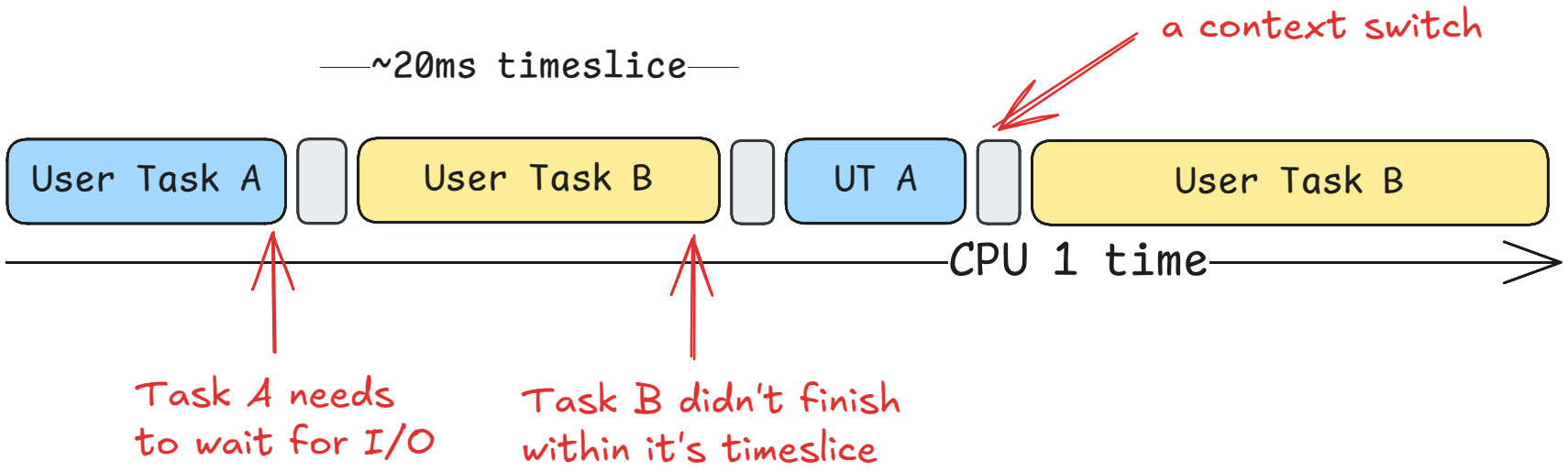


Users cannot execute programs directly
but can request the OS add them to the queue.
The queue can be prioritized.

Tasks typically get full access to all available
hardware resources to finish ASAP.

This mode is frequently found in large
High Performance Computing (HPC) clusters.

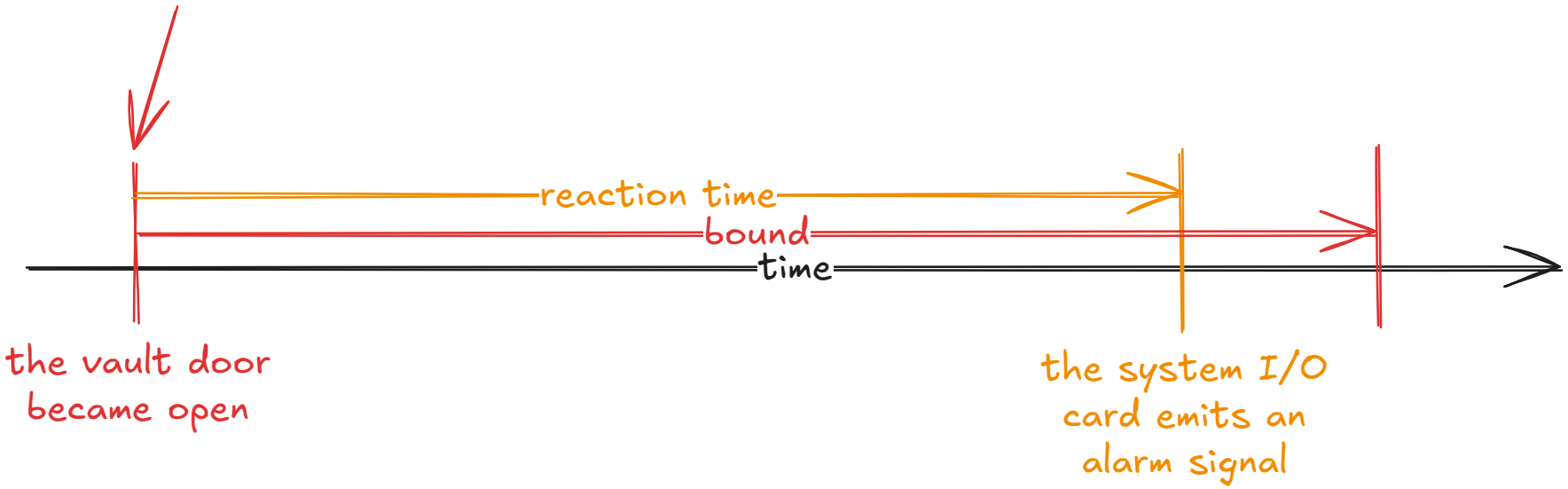
In an on-line system (also multitasking/timesharing) , the OS switches between tasks running concurrently creating interactive computing



The OS must be able to take the CPU from a running task and assign it to some other task

preemption

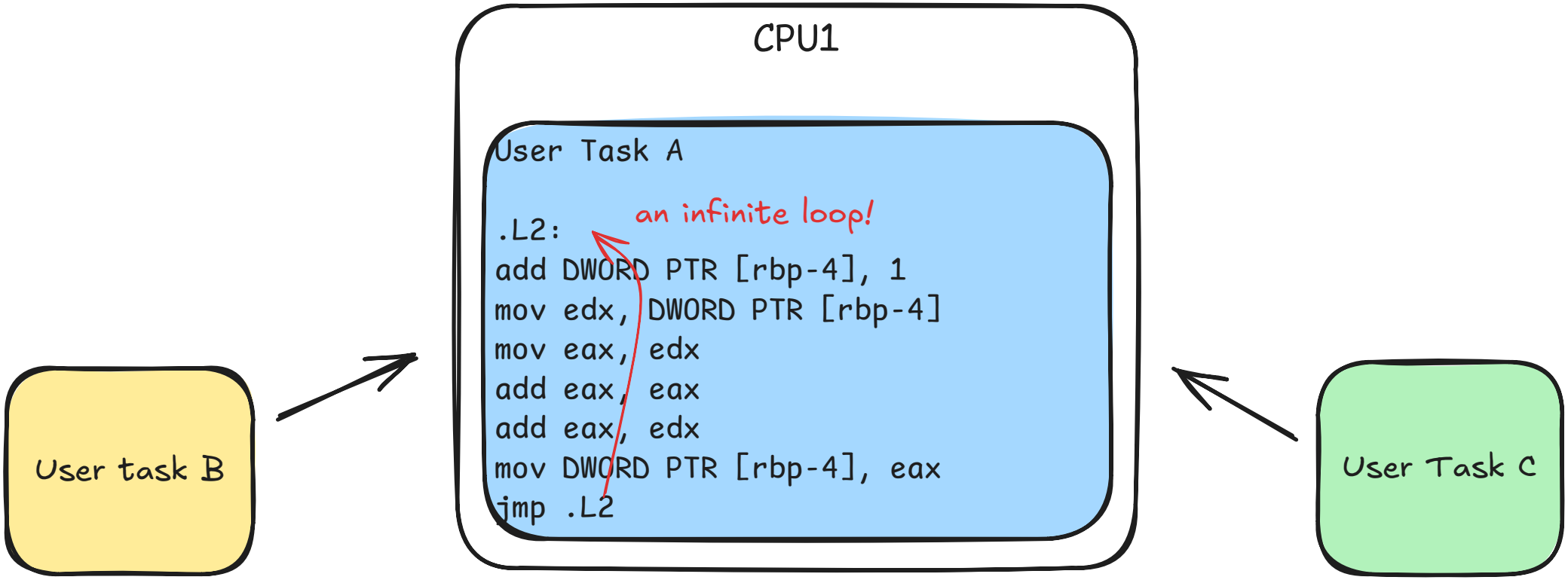
In a real time system, there exists an upper bound on the system's reaction time to an external event



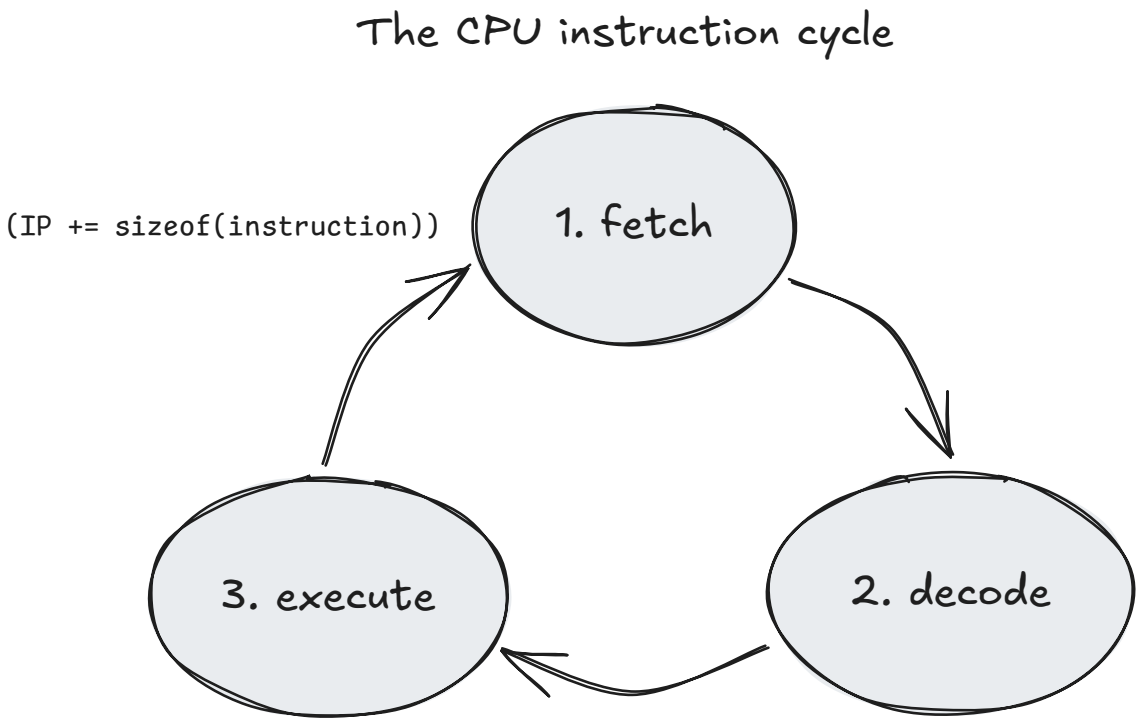
This mode is desired in mission-critical applications, i. e. in the transportation industry.

Implementing such requires many restrictions. The OS must be aware of the precise characteristics of the running tasks - what resources would they ever need. This is unpredictable for a general purpose operating system.

In a time sharing system tasks do share a single CPU core.
This is a resource which requires protection!



How to do that considering how simple CPU is?



1. read next instruction from memory

   ```
   I got 5 bytes: b8 04 00 00 00
   ```

2. figure out what those bytes mean

   ```
   Okay, 0xb8 means move next byte (0x04)
   to the register EAX
   ```

3. do what the instruction said to do

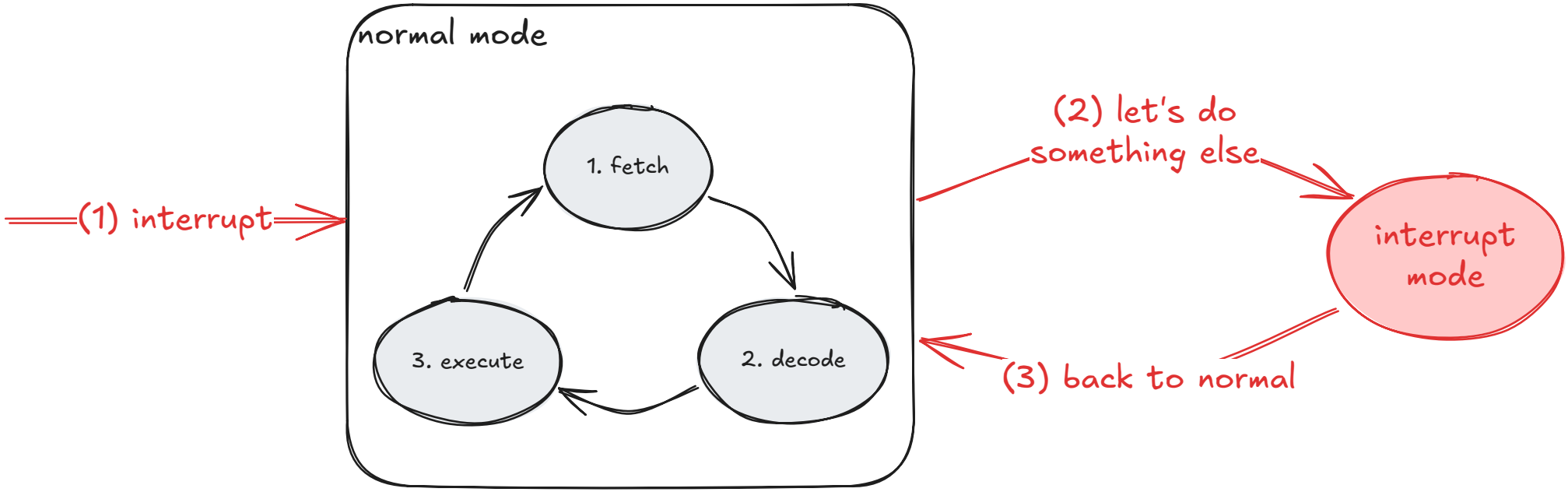
   ```
   Configure the internal CPU buses,
   registers, switches, execute many small
   substeps ending with EAX = 4
   ```

(4.) increment the Instruction Pointer (IP register)

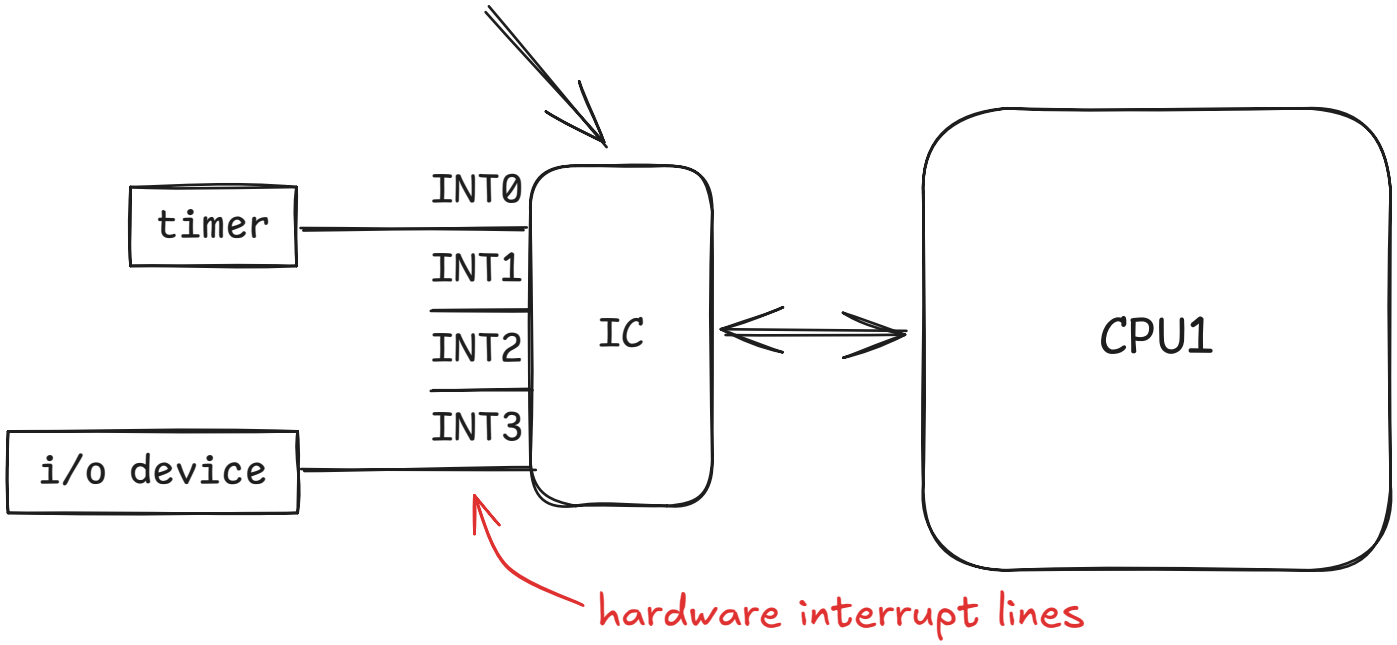
This is all hardcoded in hardware!
It's what both OS code and user code only can do!

We need a way to execute something else from time to time.

CPUs have such a mechanism - the interrupts!



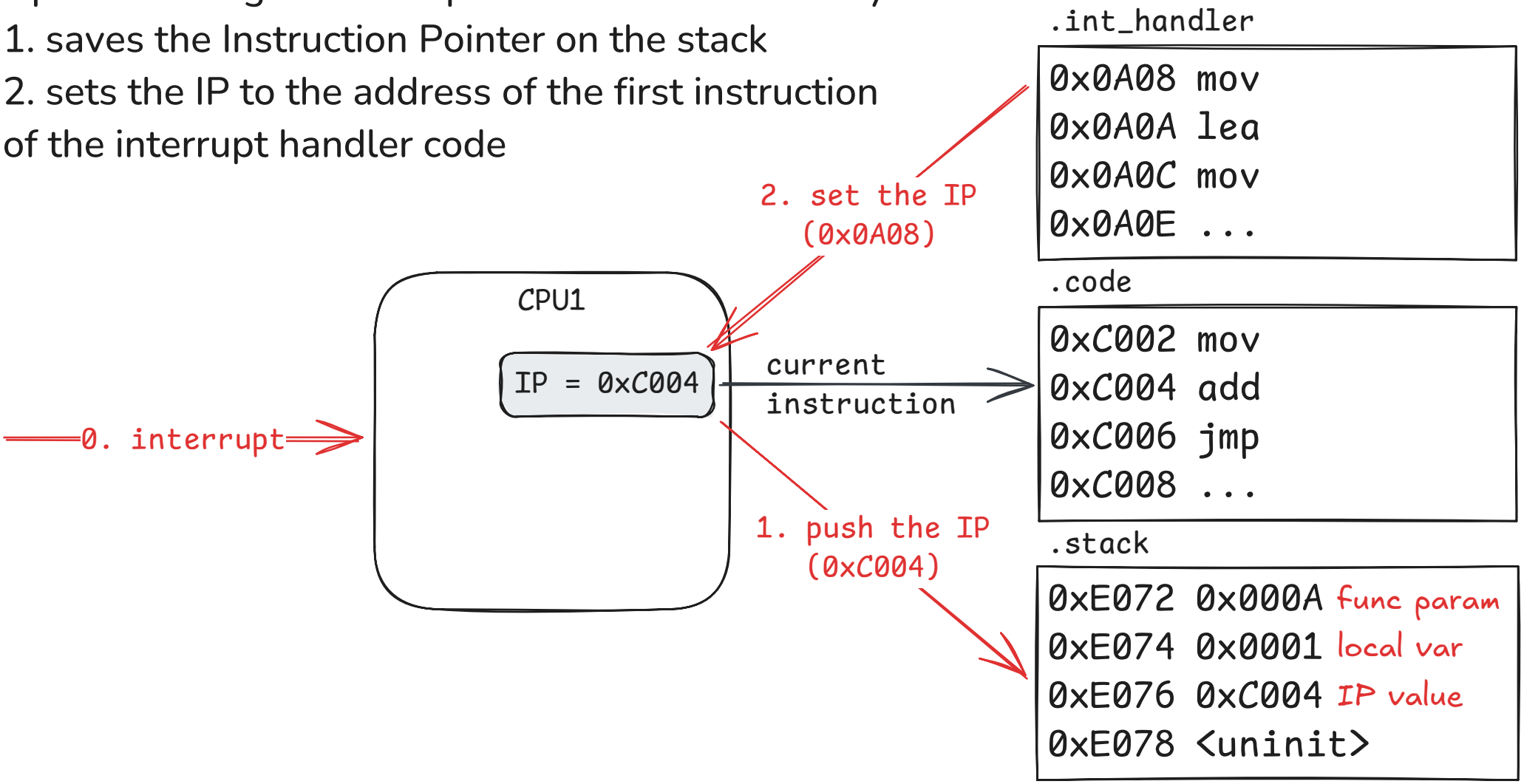
Each CPU has an Interrupt Controller attached which triggers the mode switch



OS code configures the timer to fire an interrupt in a few milliseconds with a special instruction before jumping to the user code.

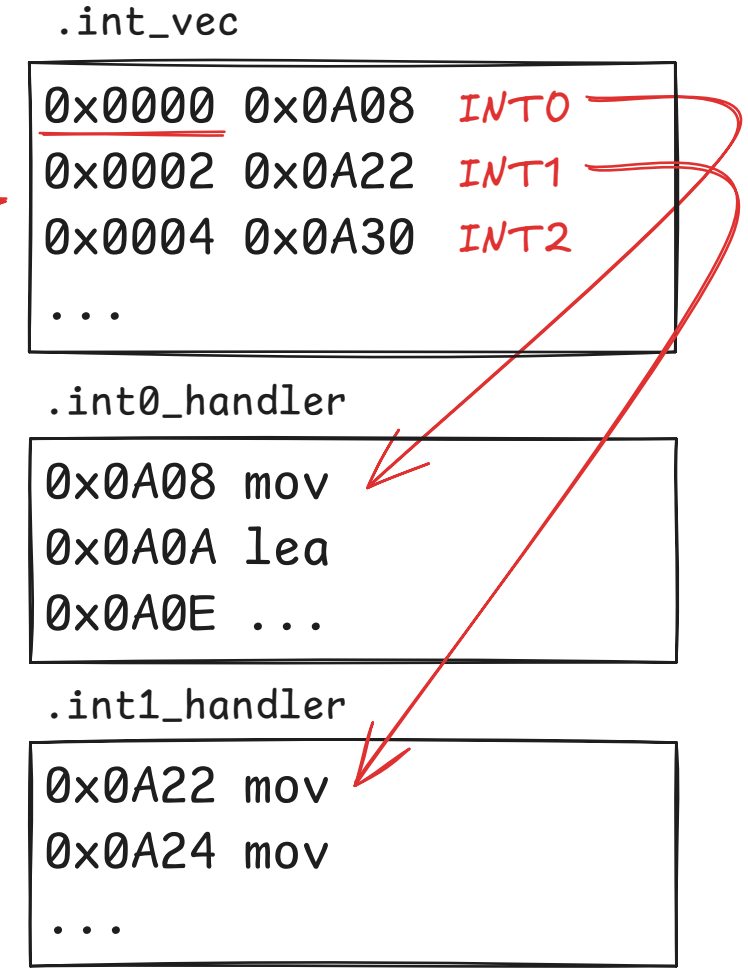
The IC can enable, disable, queue and prioritize multiple interrupts occurring at the same time and deliver them to the CPU one-by-one.

Upon receiving an interrupt the CPU automatically:
1. saves the Instruction Pointer on the stack
2. sets the IP to the address of the first instruction of the interrupt handler code



How does the CPU know the address of the interrupt handler?

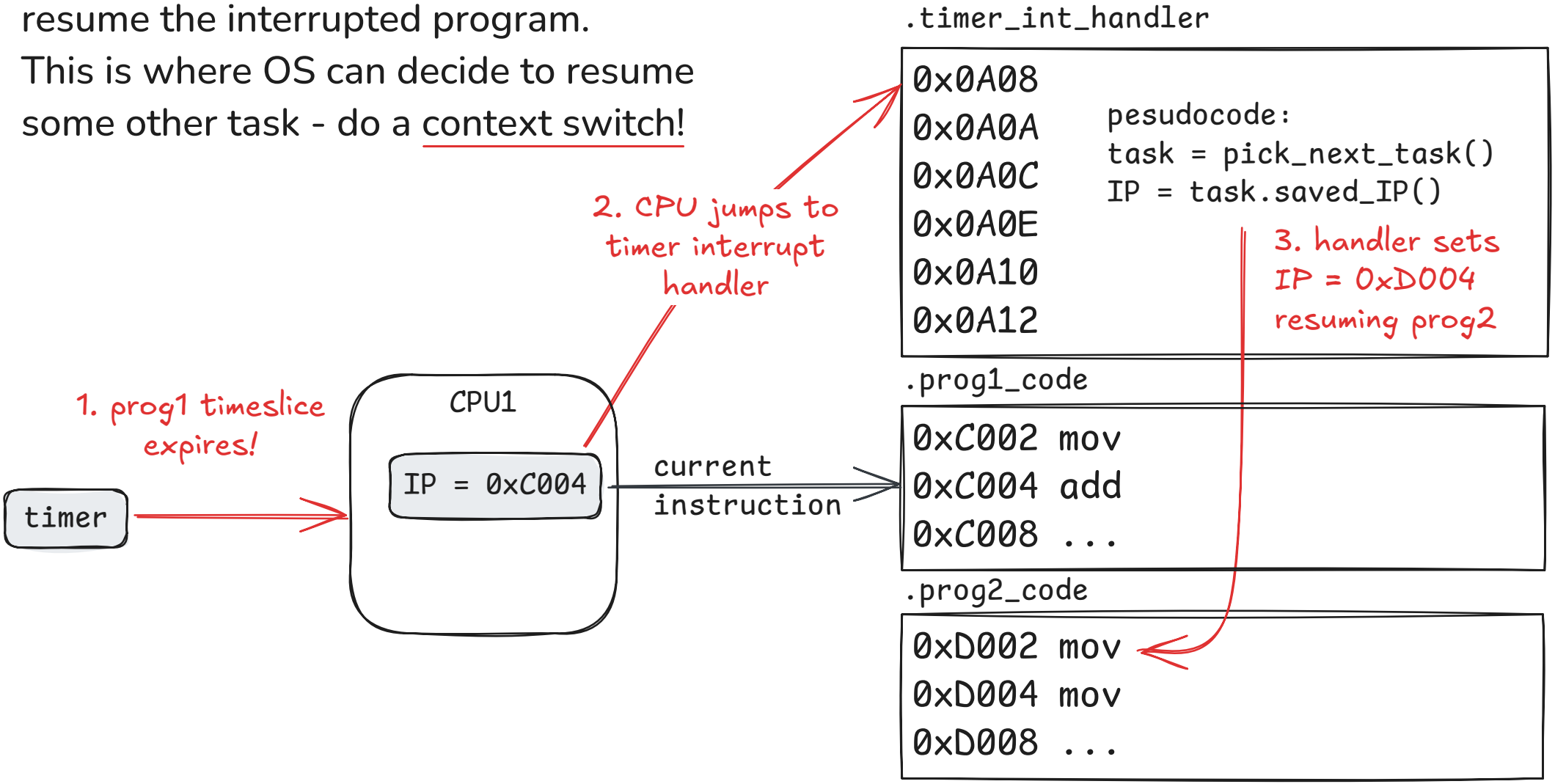
At a fixed address specified by the CPU the system (OS) keeps a table of interrupt handler addresses - the interrupt vector!



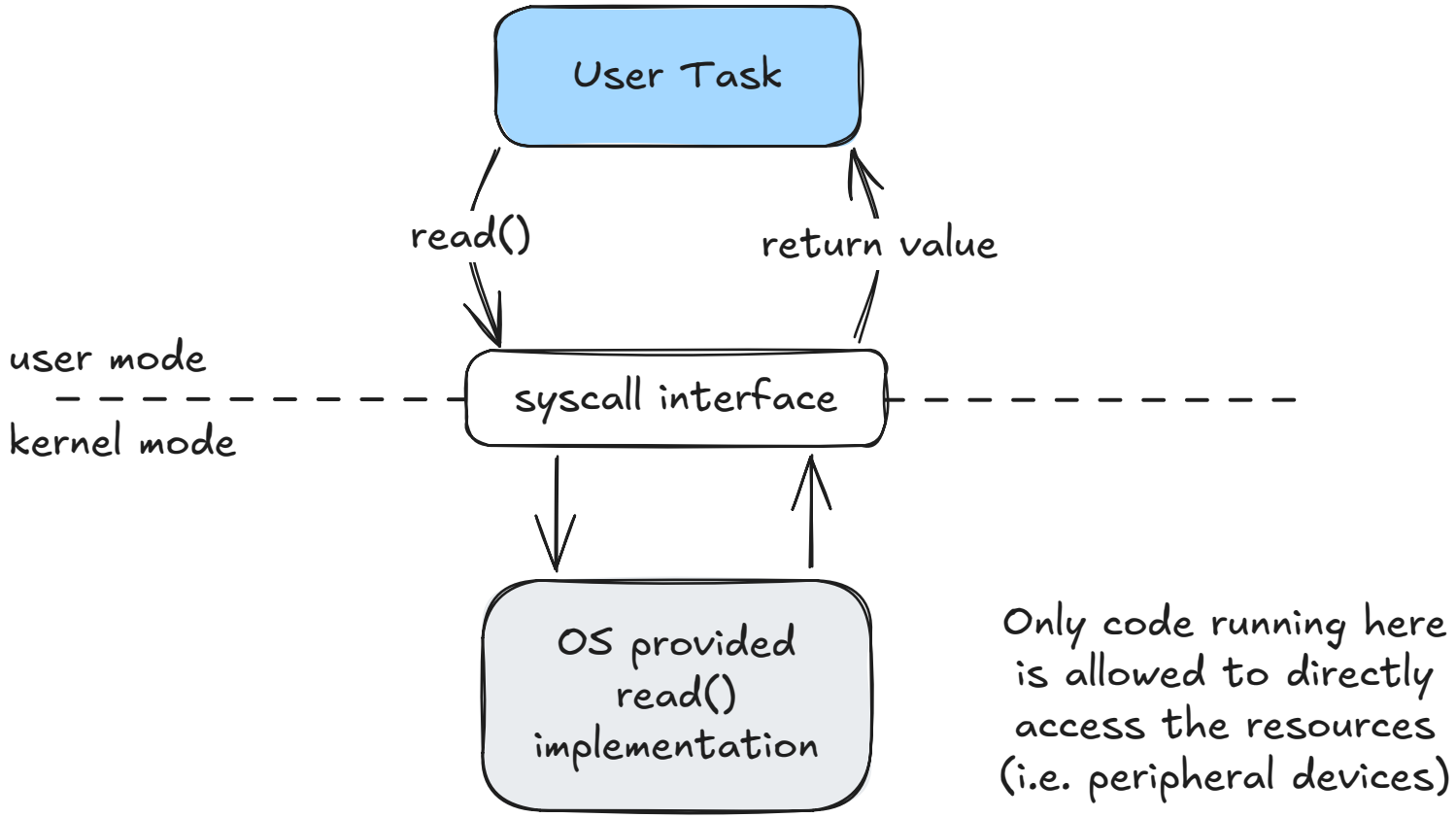
No	Addr	Source	Interrupt Definition
1	0x0000	RESET	External pin, Power-on Reset
2	0x0002	INT0	External Interrupt Request 0
3	0x0004	INT1	External Interrupt Request 1
4	0x0006	INT2	External Interrupt Request 2
5	0x0008	INT3	External Interrupt Request 3
6	0x000A	INT4	External Interrupt Request 4
7	0x000C	INT5	External Interrupt Request 5
8	0x000E	INT6	External Interrupt Request 6
8	0x0010	INT7	External Interrupt Request 7
9	0x0012	PCINT0	Pin Change Interrupt Request 0
10	0x0014	PCINT1	Pin Change Interrupt Request 1
11	0x0016	PCINT2	Pin Change Interrupt Request 2
12	0x0018	WDT	Watchdog Time-out Interrupt
13	0x001A	TIMER2_COMPA	Timer/Counter2 Compare Match A
14	0x001C	TIMER2_COMPB	Timer/Counter2 Compare Match B
...			

AVR® Interrupts - simple example of a microcontroller interrupt vector memory layout
https://developerhelp.microchip.com/xwiki/bin/view/products/mcu-mpu/8-bit-avr/structure/interrupts/

Interrupt handler code does not need to resume the interrupted program.
This is where OS can decide to resume some other task - do a context switch!

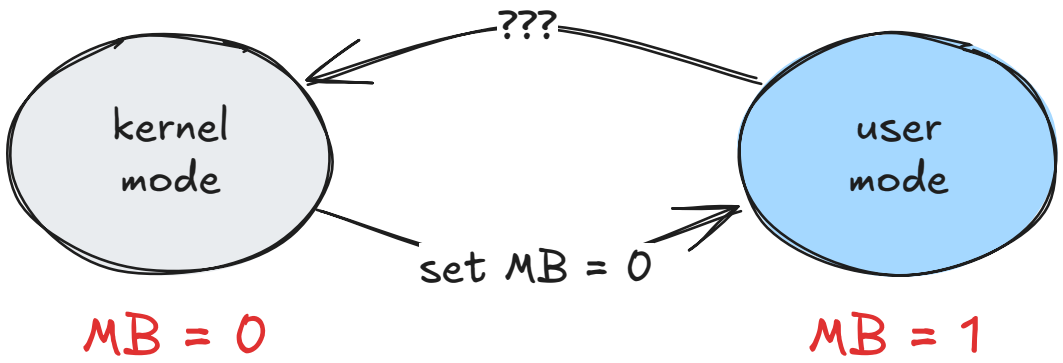


In a time-sharing system a userspace program cannot access any shared system resource directly.
It must do so via a system call!



How to disallow anything for the user code then?

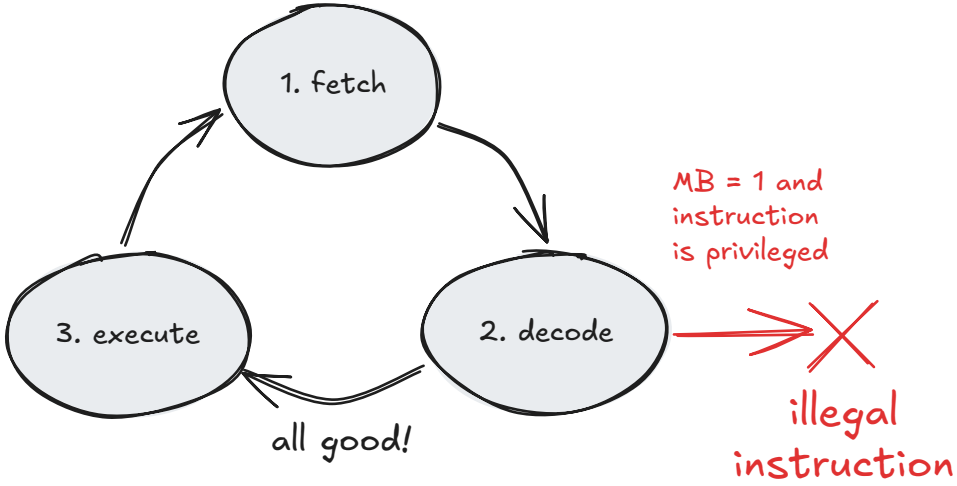
We need a Mode Bit!



A single bit stored within the CPU core, set to 1 when running the OS code.

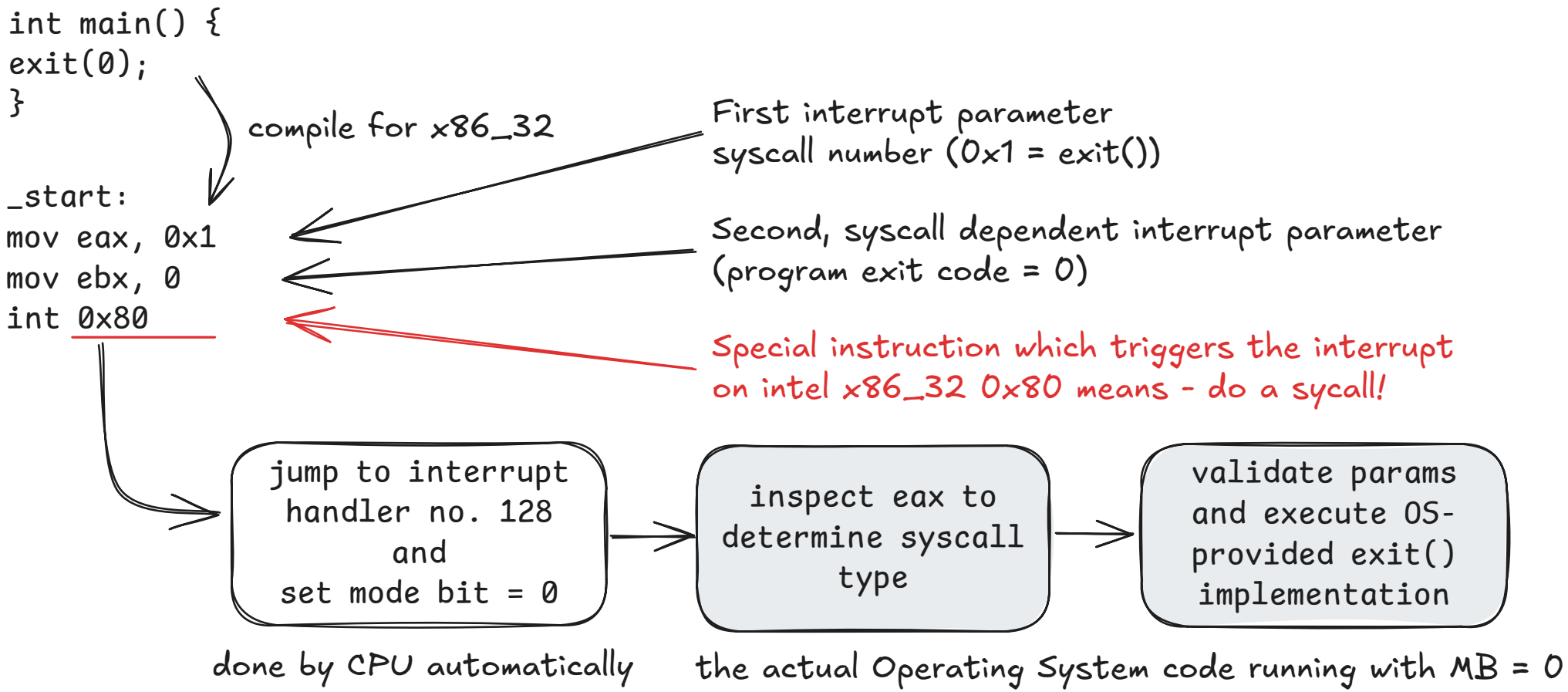
When MB = 0 some instructions are disallowed - in hardware.

A single bit stored within the CPU core, set to 1 when running the OS code.



How a program can enter the kernel mode then?

It triggers a software interrupt!



The Linux syscall table

```
> man 2 syscalls
```

https://filippo.io/linux-syscall-table/

```
%rax Name Manual Entry point
0 read read(2) sys_read
1 write write(2) sys_write
2 open open(2) sys_open
3 close close(2) sys_close
4 stat stat(2) sys_newstat
5 fstat fstat(2) sys_newfstat
6 lstat lstat(2) sys_newlstat
7 poll poll(2) sys_poll
8 lseek lseek(2) sys_lseek
9 mmap mmap(2) sys_ksys_mmap_pgoff
...
```

There 456 distinct syscalls in Linux 6.7 kernel! - and we're gonna learn just a few