The Big Picture

Thierry Sans

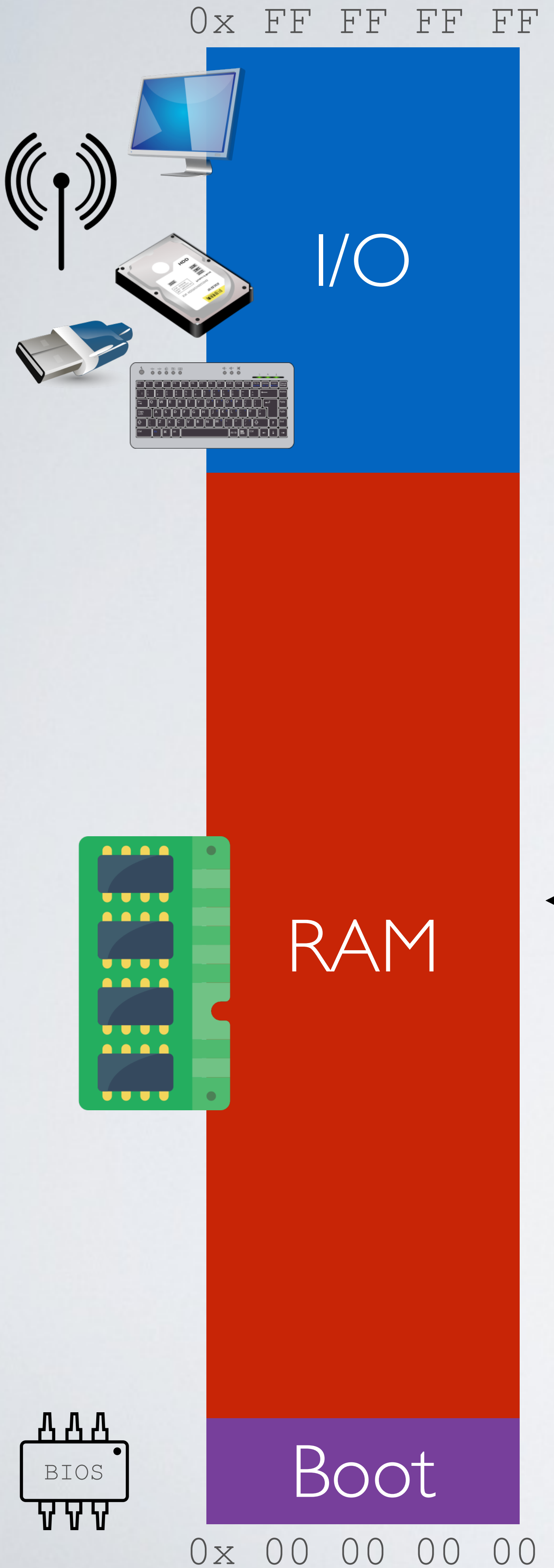
Goals of this lecture

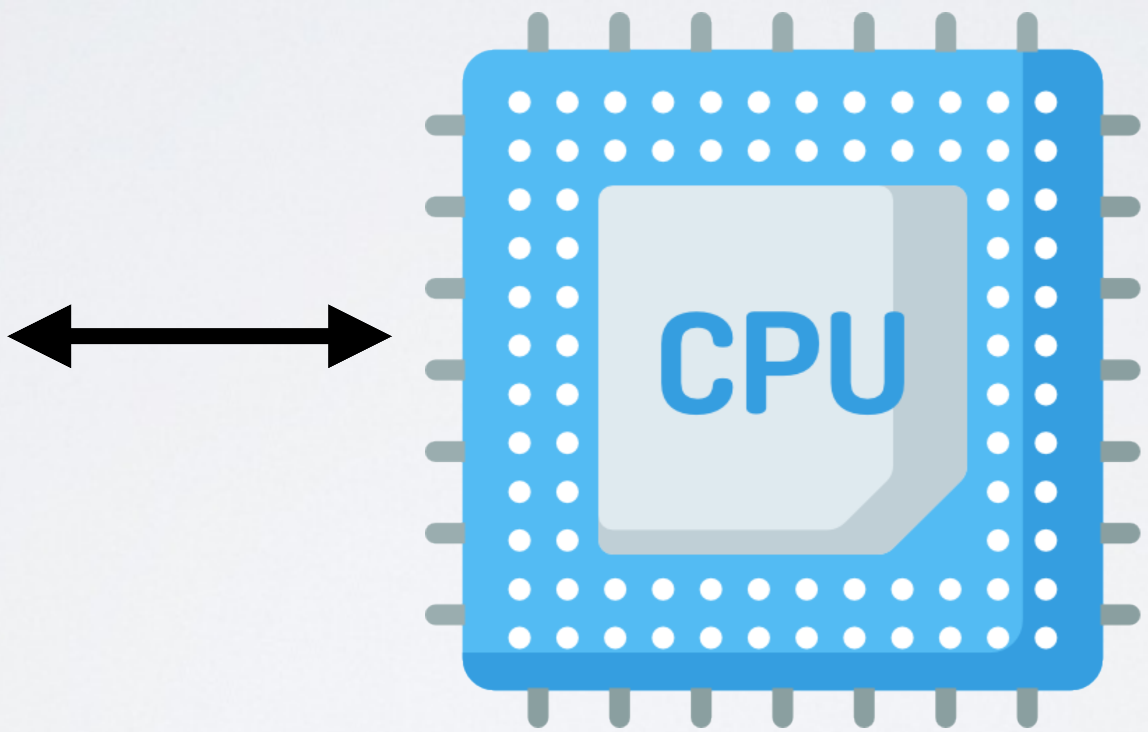
- Define what an Operating System is
- Explain how an OS works in a nutshell
- Bridge the gap between hardware (CSCB58) and systems programming (CSCB09)
- Give an overview of the course content and projects

The big picture in 5 pieces

The need for **bootstrapping**	
The need for **concurrency**	project 1
The need for **user programs**	project 2
The need for **virtual memory**	project 3
The need for **a filesystem**	project 4

Simple Computer Architecture
Memory + CPU





for a more accurate and detailed map of the x86 memory look at https://wiki.osdev.org/Memory_Map_(x86)

Each processor has its Instruction Set Architecture (ISA)

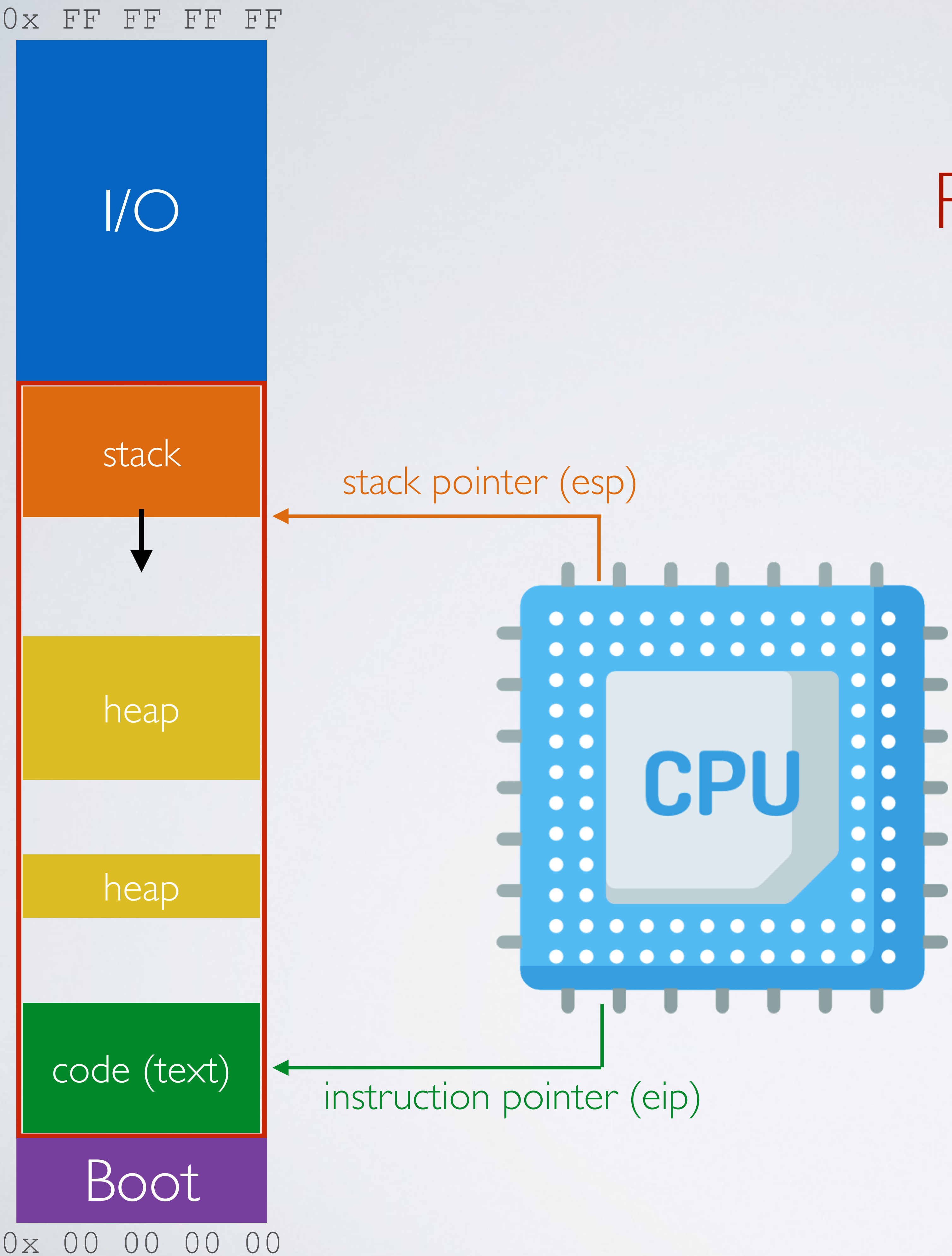
Processor executes instructions stored in memory

➡ Each instruction is a bit string that the processor understands as an operation

- arithmetic
- read/write bit strings
- bit logic
- jumps

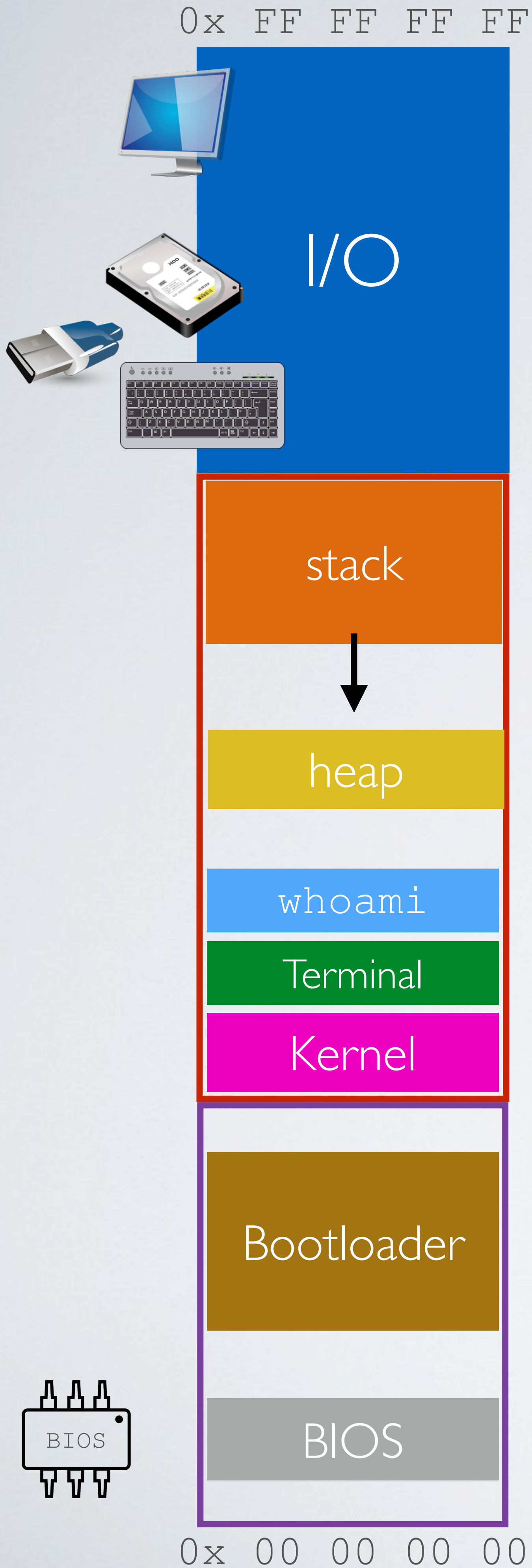
✓ ~2000 instructions on modern x86-64 processors

Running one program



The need for **bootstrapping**

Bootstrapping



Step 5: using the terminal, users can execute programs (e.g Bash terminal) ... and repeat

Step 4: the kernel starts the user-interface program (e.g `Bash` terminal)

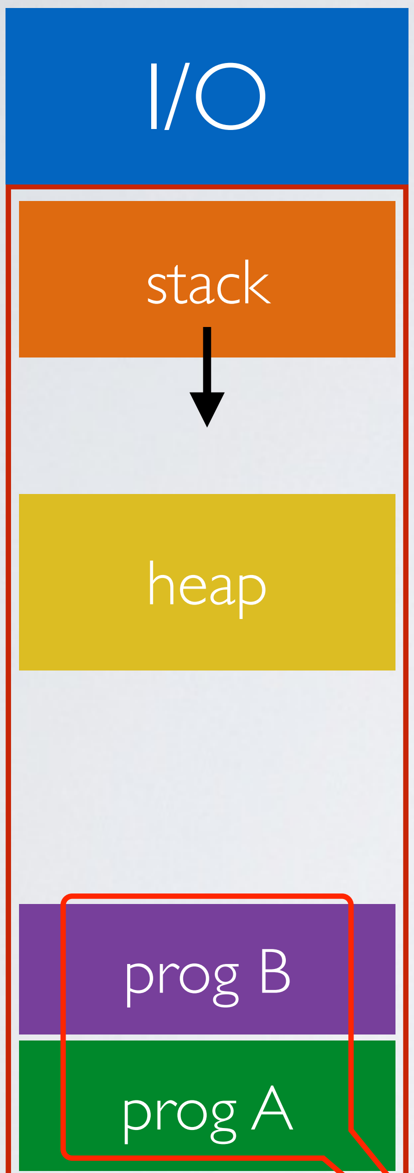
Step 3: the bootloader loads the OS kernel in RAM

Step 2: the BIOS loads the **bootloader** from a device (hard-drive, USB, network ...) based on the configuration

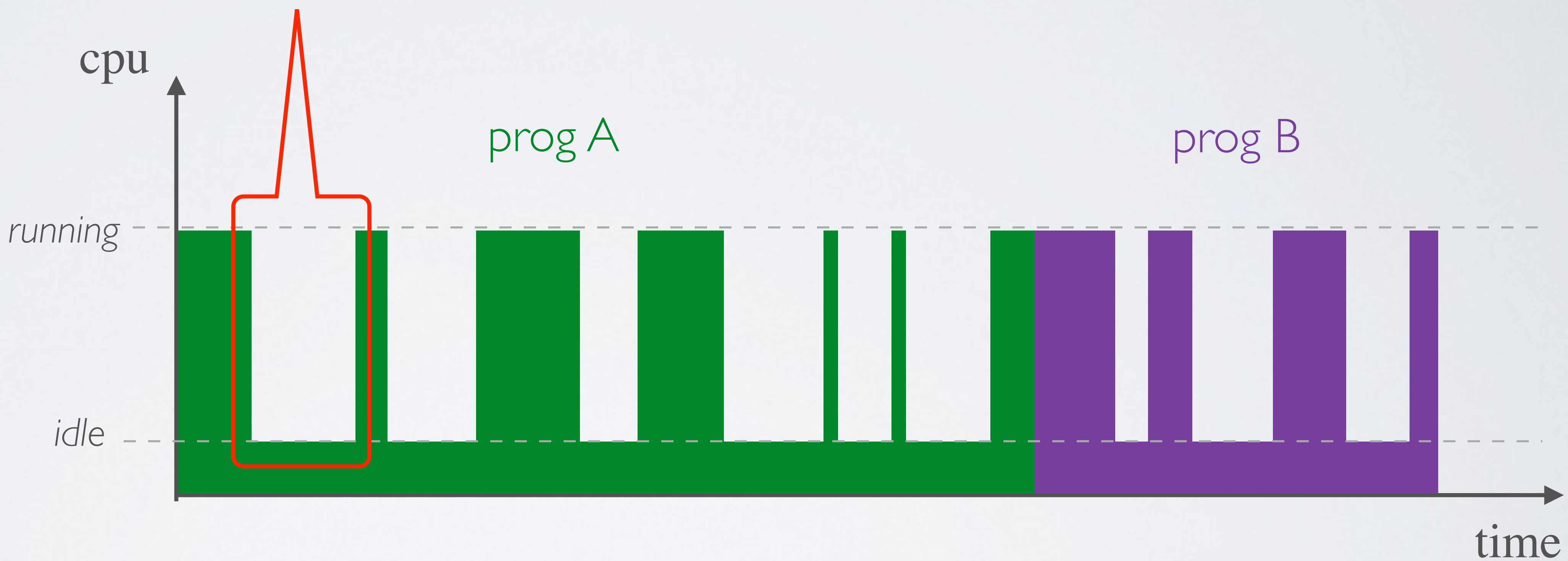
Step 1: Power -on! The CPU starts executing code contained in the **BIOS** (basic input/output system)

The need for **concurrency**

Running multiple programs one after the other

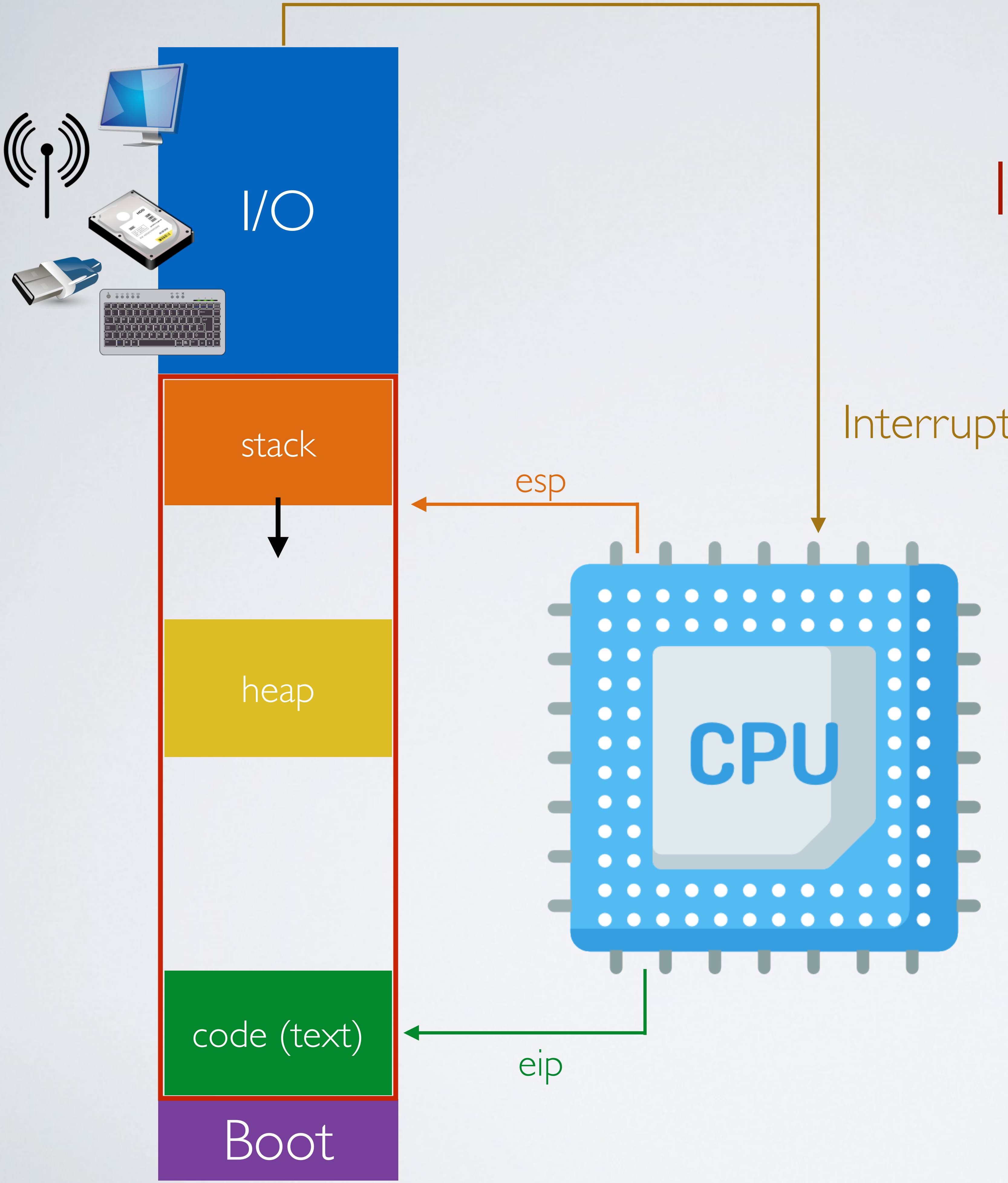


Problem: the CPU is waiting for I/O (polling)



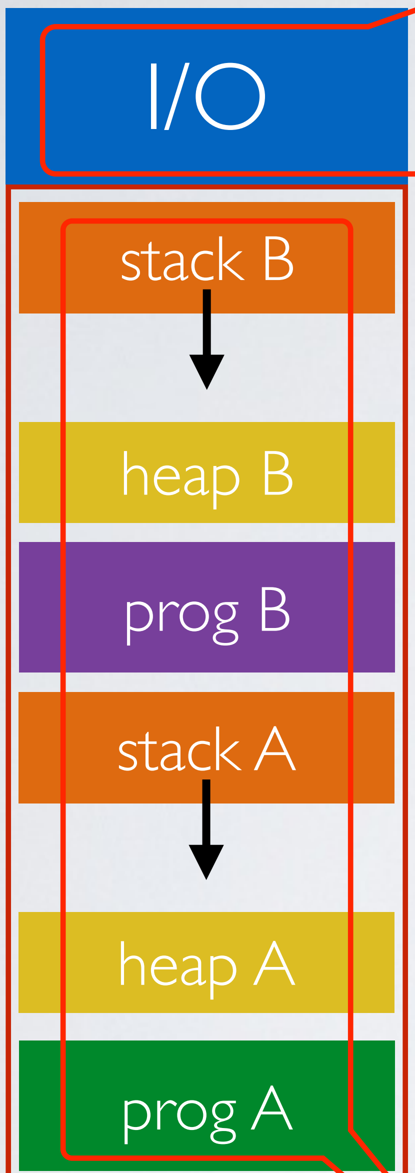
Problem: the programs must co-exists in memory (coming next with virtual memory)

I/O with interrupts

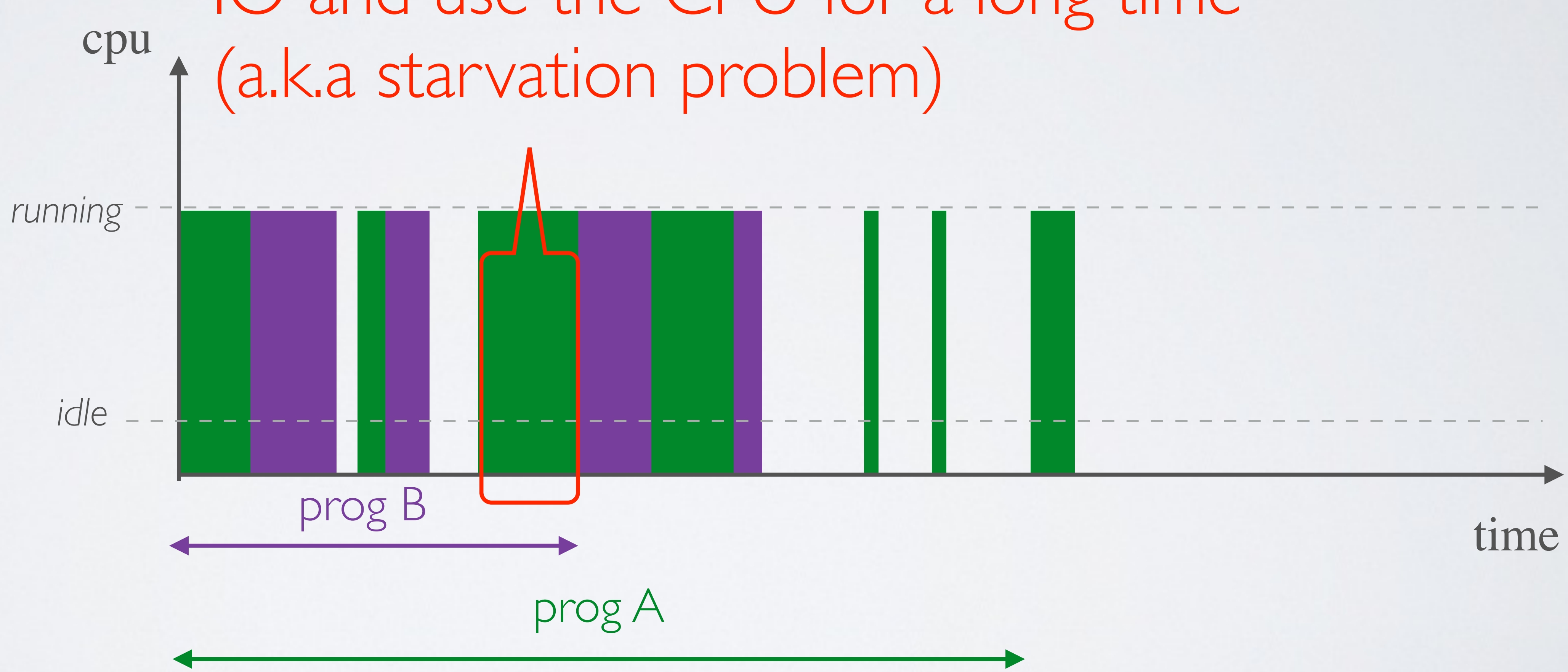


Running multiple programs concurrently

Problem: concurrent access to I/O devices must be synchronized

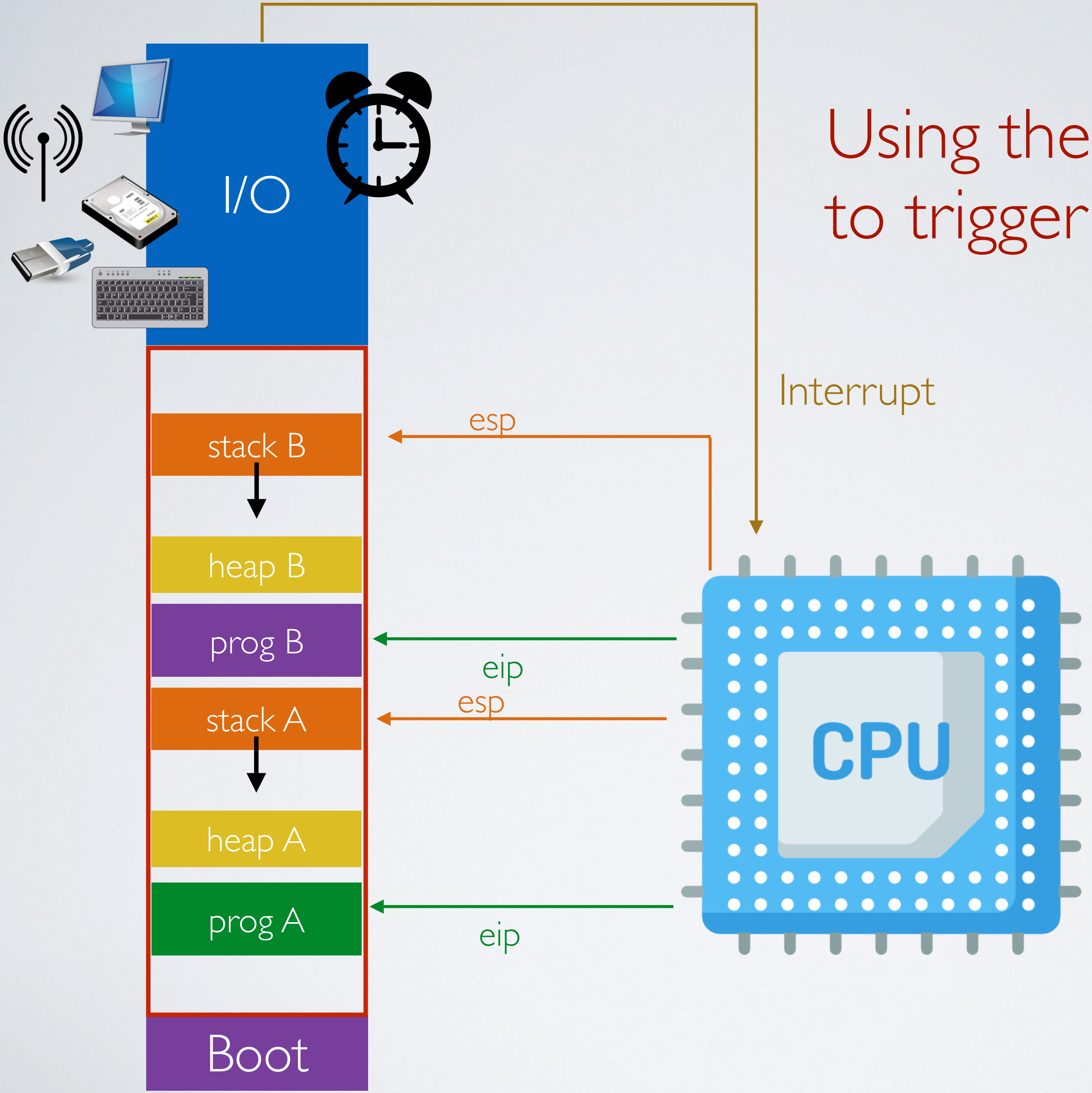


Problem: what if the program does not do any IO and use the CPU for a long time (a.k.a starvation problem)

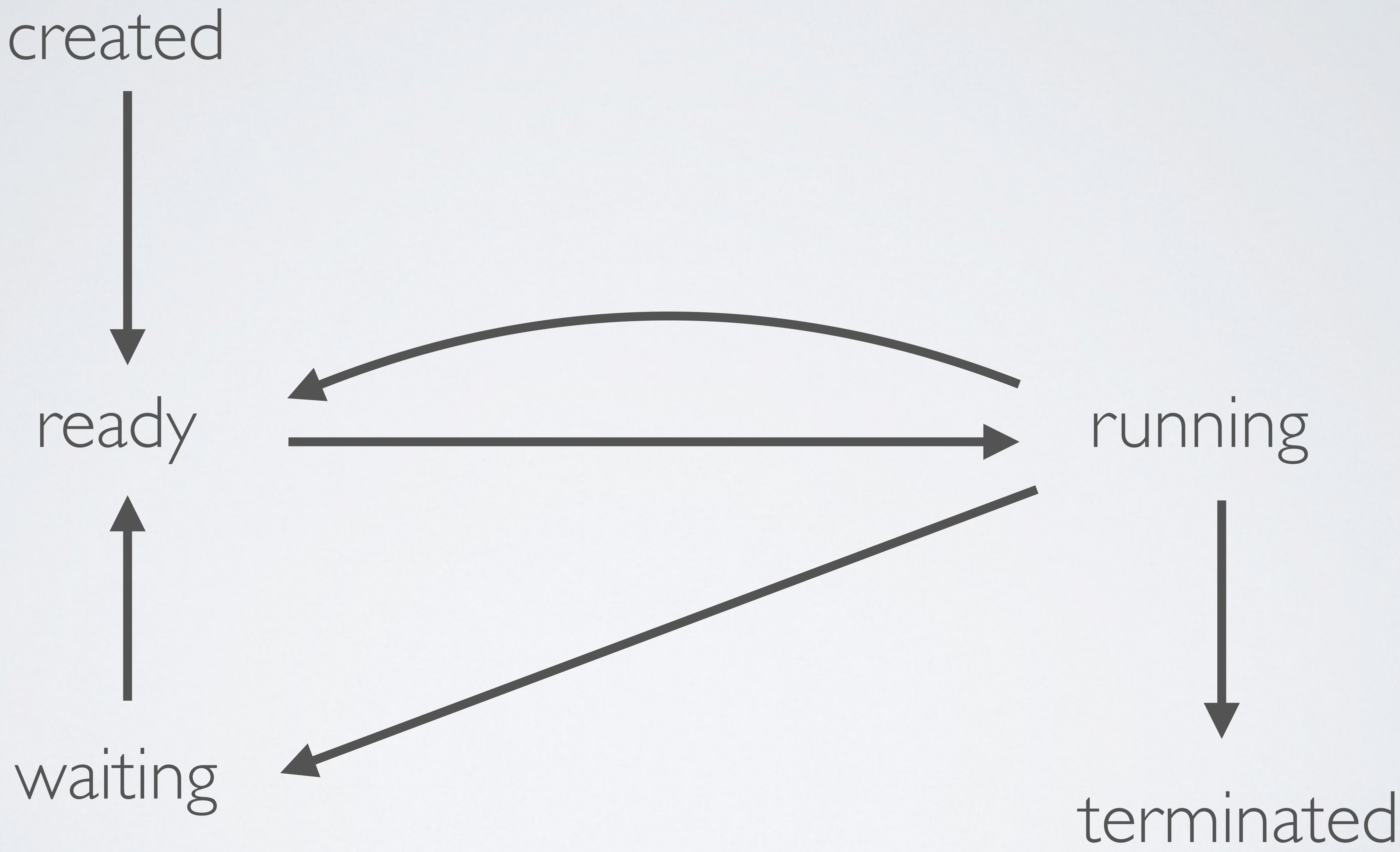


Problem: the programs and their stacks must co-exists in memory (coming next with virtual memory)

Using the clock to trigger an interrupt



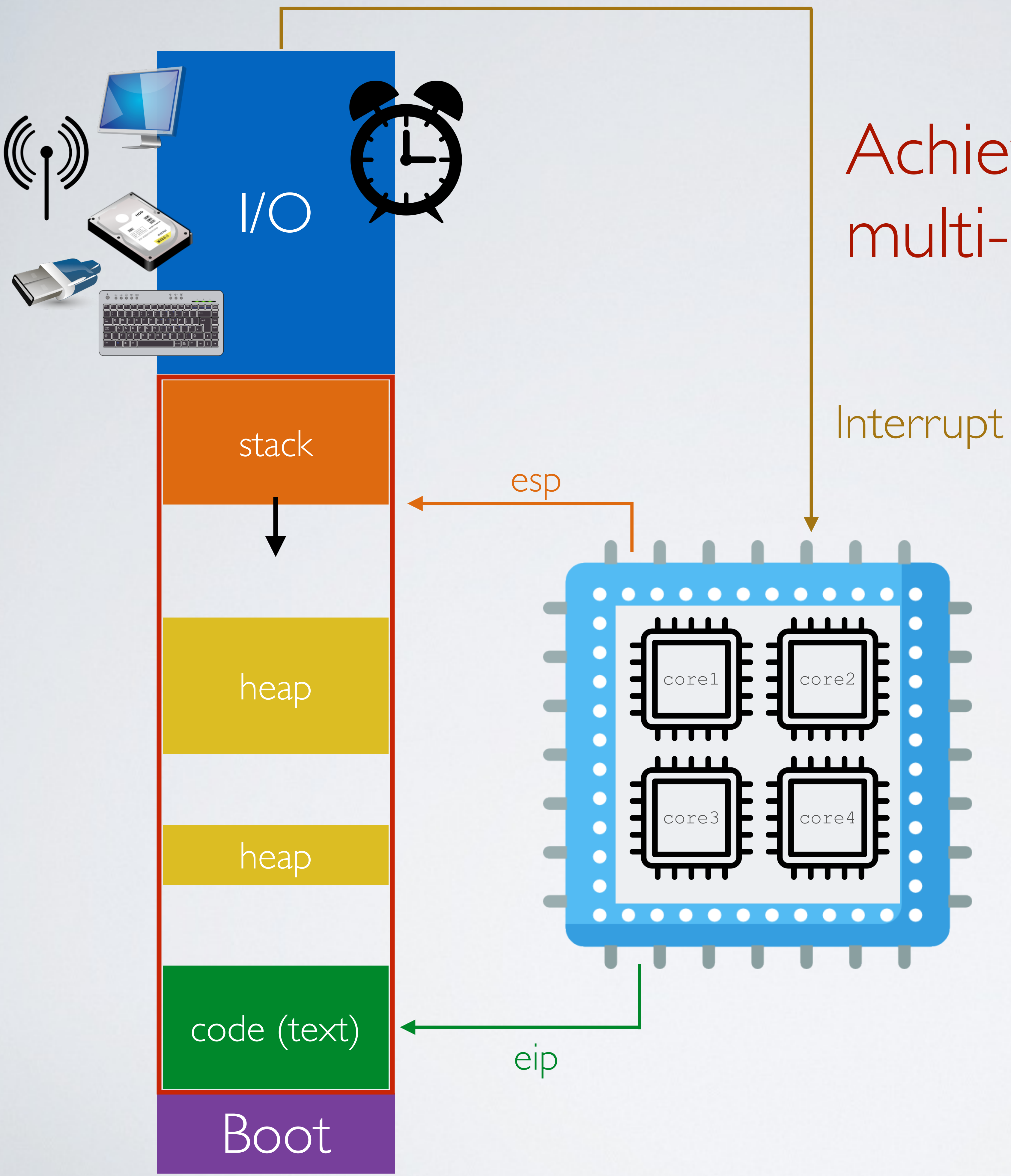
Program States



Other problems that we are going to address during the semester

- **Scheduling**
  Decide which process to execute when severals are ready to be run
- **Synchronization**
  Manage concurrent access to resources using semaphores, locks, monitors
- **Communication**
  Exchange messages between processes using IPC (sockets & signals)
- **Threads**
  Lightweight concurrency within a process

Achieving parallelism with multi-core processors



The need for **user programs**

The need for abstraction for user programs

How to write a user program like the *Bash* shell that reads keyboard inputs from the user?

➡ Read input data from the I/O device directly? But which one?

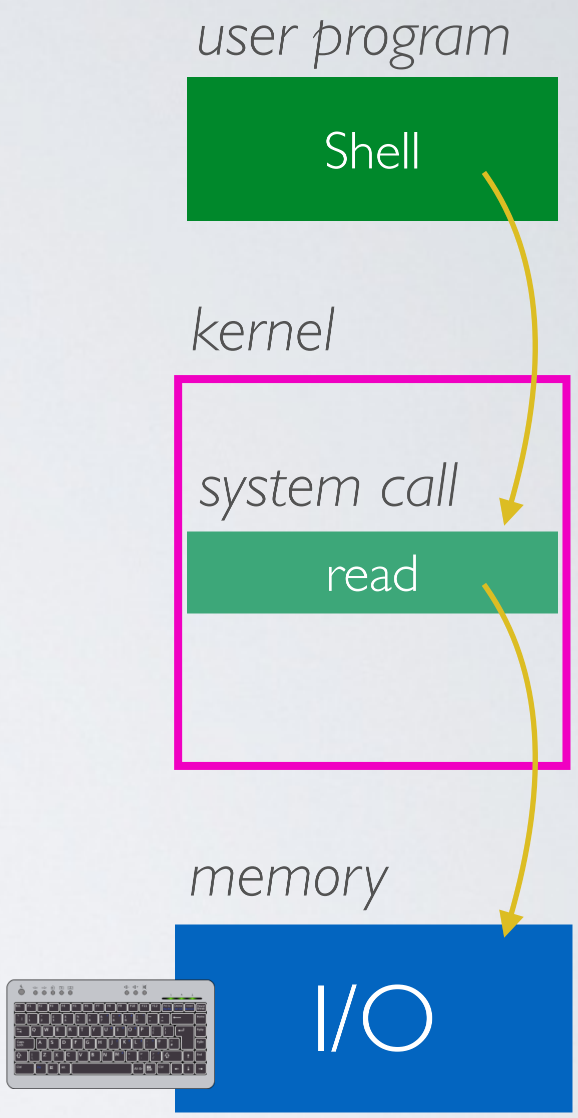
- The one connected to the PS2 port?
- The one connected to the USB?
- The one connected to the bluetooth?
- The remote one connected to the network?

◉ User programs do not operate I/O devices directly

✓ The OS abstracts those functionalities and provide them as **system calls**

System Calls

➡ Provide user programs with an API to use the services of operating system



There are 5 categories of system calls

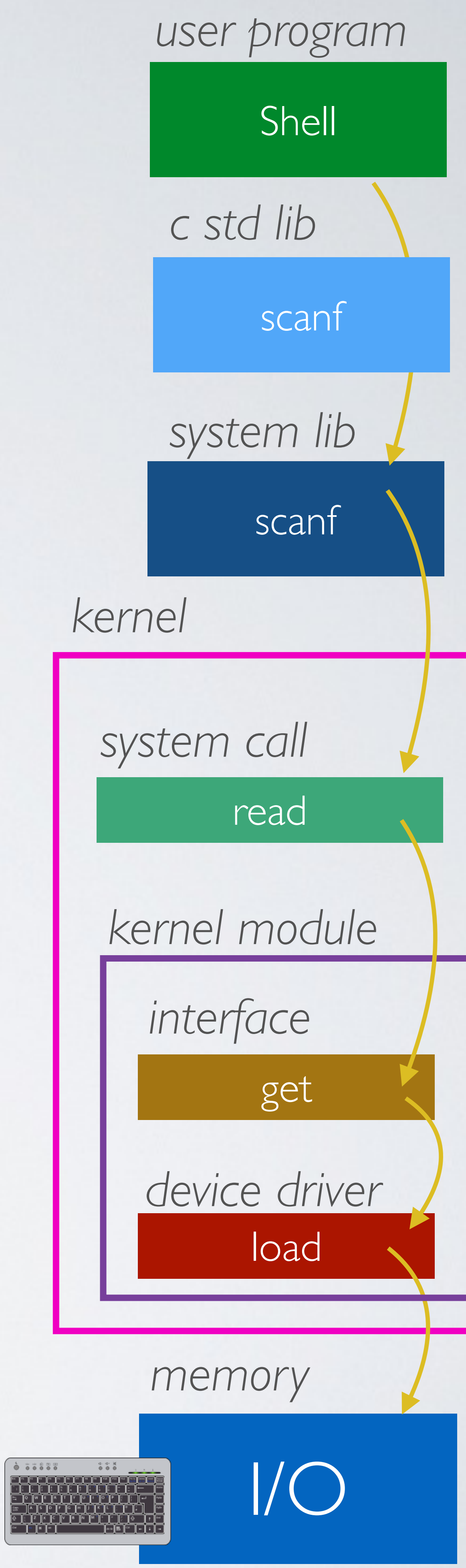
- Process control
- File management
- Device management
- Information/maintenance (system configuration)
- Communication (IPC)
- Protection

✓ There are 393 system calls on Linux 3.7

http://www.cheat-sheets.org/saved-copy/Linux_Syscall_quickref.pdf

In reality, many (many) level of abstraction and modularity

➡ This is what makes developing OS very challenging (CSCB07)



With concurrency

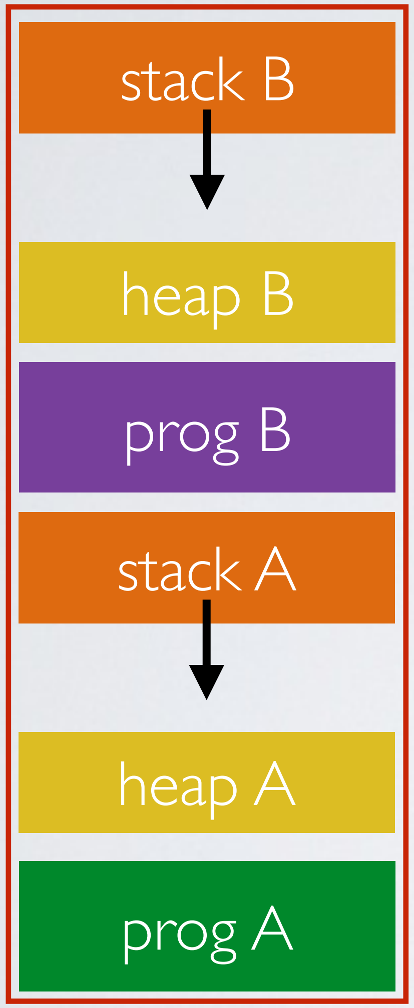
- ✓ **From the system perspective**
  better CPU usage resulting in a faster execution overall (but not individually)

- ✓ **From the user perspective**
  programs seem to be executed in parallel

- ➡ But it requires scheduling, synchronization and some protection mechanisms

The need for **virtual memory**

The problem of managing the memory



How to make programs and execution contexts co-exists in memory?

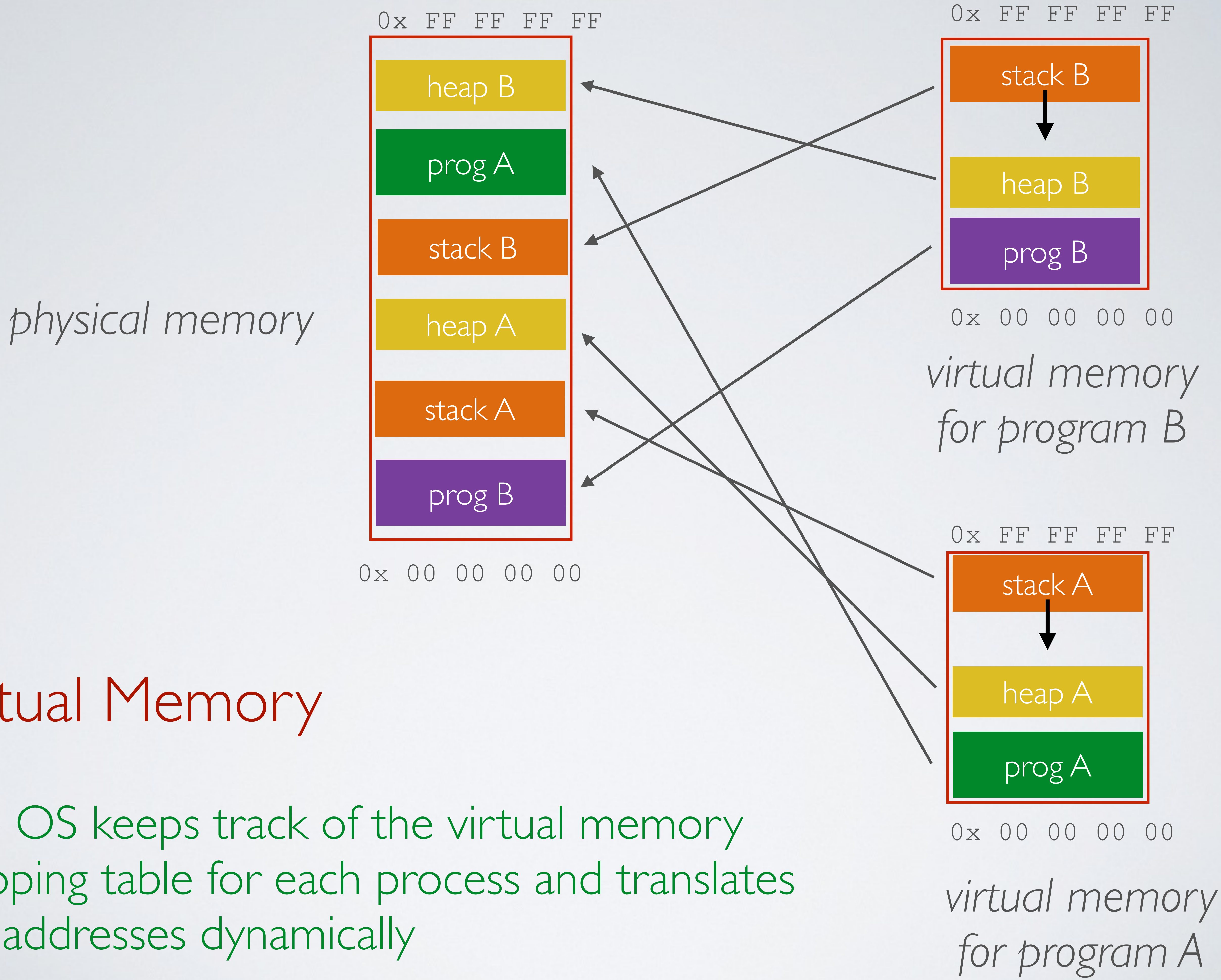
- ✓ Placing multiple execution contexts (stack and heap) at random locations in memory is not a problem ... ... well, as long as your have enough memory
- ◉ However having programs placed at random locations is problematic

Let's look at some C code and its binary

```c
#include <stdio.h>

int foo(){
    printf("hello world!");
}

int main(int argc, char **argv){
    foo();
}
```

Since function addresses and others are hard-encoded in the binary, the program cannot be placed at random locations in memory

```
0804840b <foo>:
 804840b:   55                      push   ebp
 804840c:   89 e5                   mov    ebp,esp
 804840e:   83 ec 08                sub    esp,0x8
 8048411:   83 ec 0c                sub    esp,0xc
 8048414:   68 d0 84 04 08          push   0x80484d0
 8048419:   e8 c2 fe ff ff          call   80482e0 <printf@plt>
 804841e:   83 c4 10                add    esp,0x10
 8048421:   90                      nop
 8048422:   c9                      leave
 8048423:   c3                      ret

08048424 <main>:
 8048424:   8d 4c 24 04             lea    ecx,[esp+0x4]
 8048428:   83 e4 f0                and    esp,0xfffffff0
 804842b:   ff 71 fc                push   DWORD PTR [ecx-0x4]
 804842e:   55                      push   ebp
 804842f:   89 e5                   mov    ebp,esp
 8048431:   51                      push   ecx
 8048432:   83 ec 04                sub    esp,0x4
 8048435:   e8 d1 ff ff ff          call   804840b <foo>
 804843a:   b8 00 00 00 00          mov    eax,0x0
 804843f:   83 c4 04                add    esp,0x4
 8048442:   59                      pop    ecx
 8048443:   5d                      pop    ebp
 8048444:   8d 61 fc                lea    esp,[ecx-0x4]
 8048447:   c3                      ret
 8048448:   66 90                   xchg   ax,ax
 804844a:   66 90                   xchg   ax,ax
 804844c:   66 90                   xchg   ax,ax
 804844e:   66 90                   xchg   ax,ax
```



Virtual Memory

The OS keeps track of the virtual memory mapping table for each process and translates the addresses dynamically

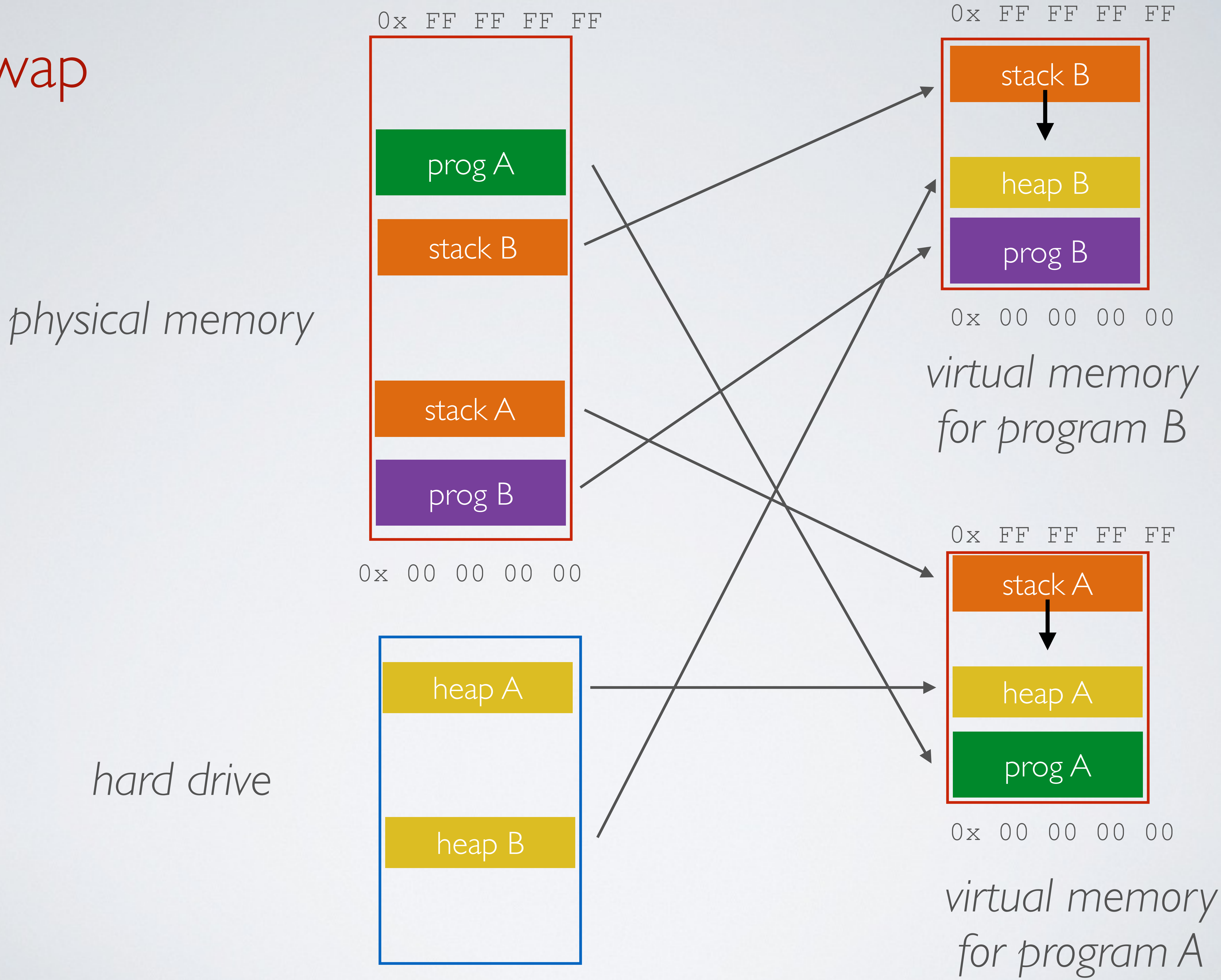
Another problem

What if we run out of memory because of too many concurrent programs?

✓ Swap memory
move some data to the disk

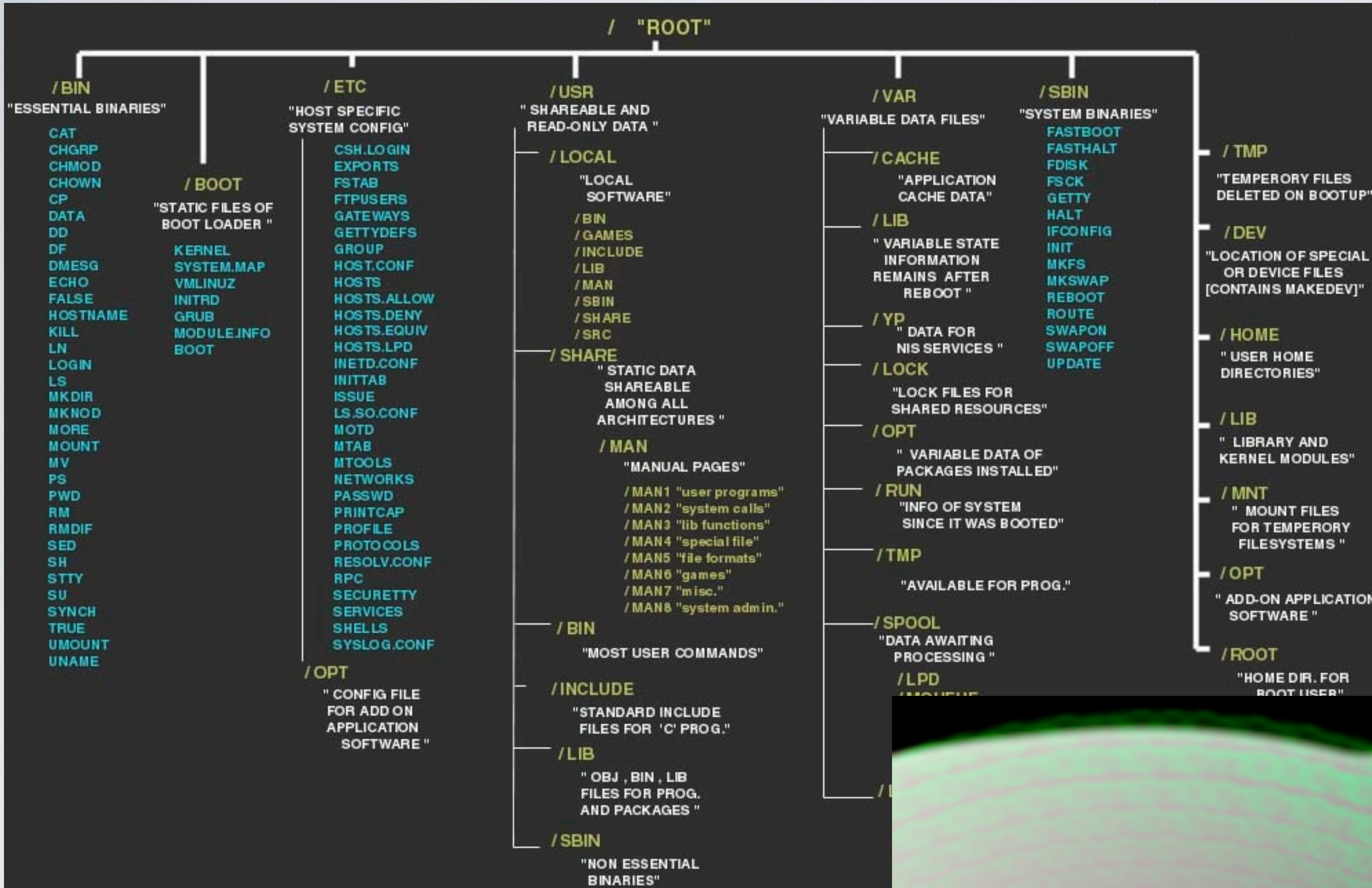
➡ Managing memory becomes very complex
but necessary

Swap



physical memory
0x FF FF FF FF
prog A
stack B
stack A
prog B
0x 00 00 00 00
hard drive
heap A
heap B
0x FF FF FF FF
stack B
heap B
prog B
0x 00 00 00 00
virtual memory for program B
0x FF FF FF FF
stack A
heap A
prog A
0x 00 00 00 00
virtual memory for program A

The need for a **file system**



Files and Directories

versus

Reality

So, what is an **operating system**?

Operating System

- ➡ In a nutshell, an OS manages hardware and runs programs
    - creates and manages processes
    - manages access to the memory (including RAM and I/O)
    - manages files and directories of the filesystem on disk(s)
    - enforces protection mechanisms for reliability and security
    - enables inter-process communication