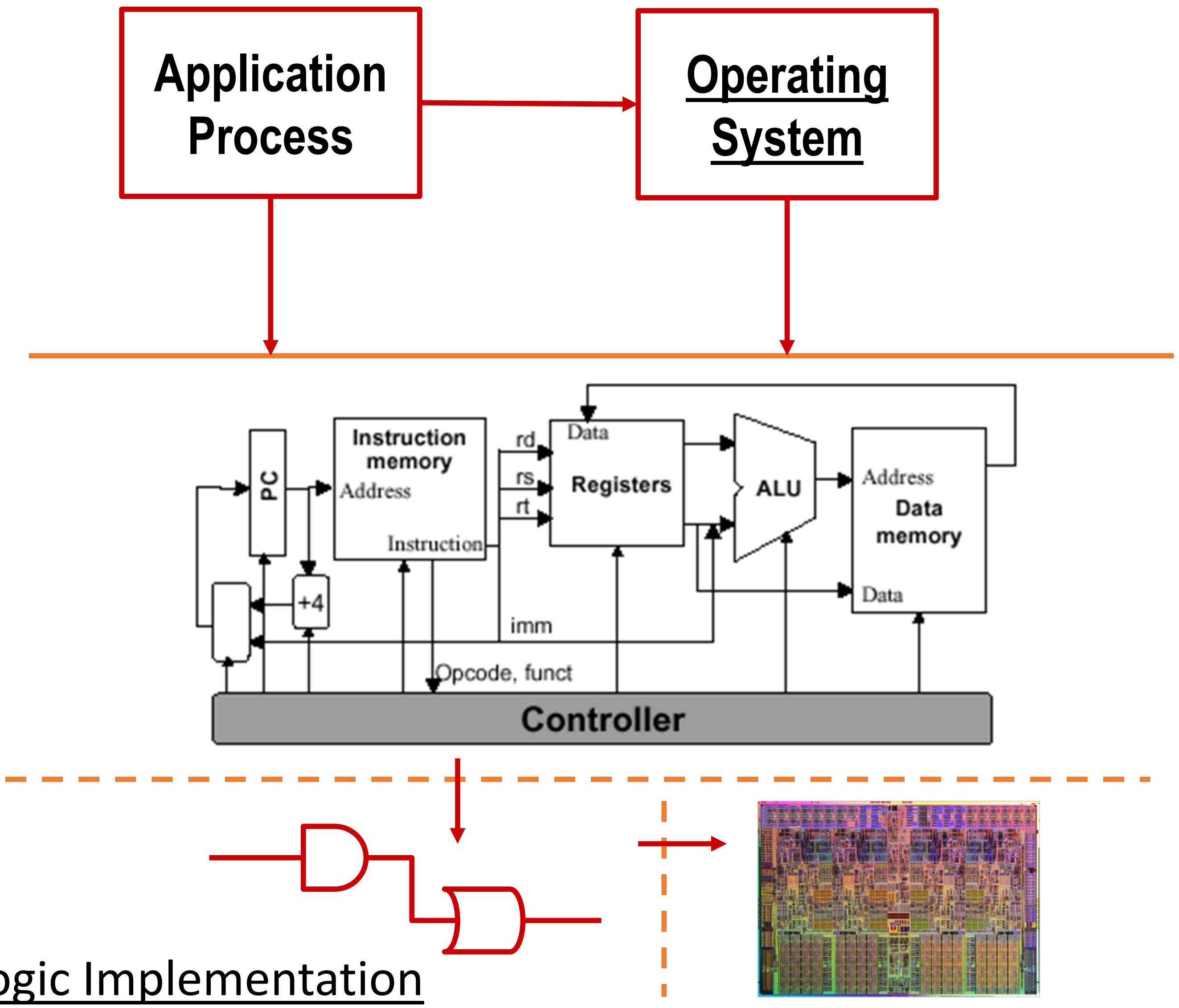
Computer Systems

CSE 410 WInter 2022

13 – OS Introduction: Basic Functioning

What This Course is About



Software in Execution

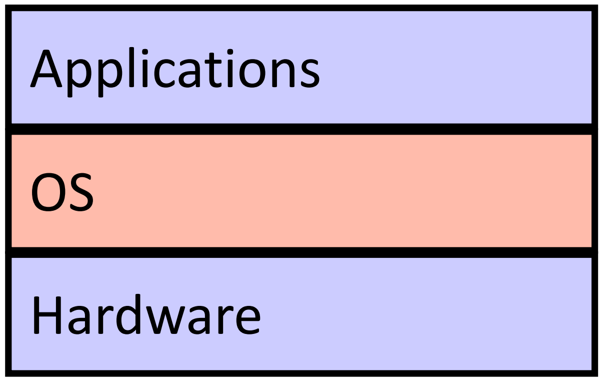
Instruction Set Architecture
(e.g., x86-64)

Machine Organization
(e.g., Core i7-8550)

Hardware

Logic Implementation

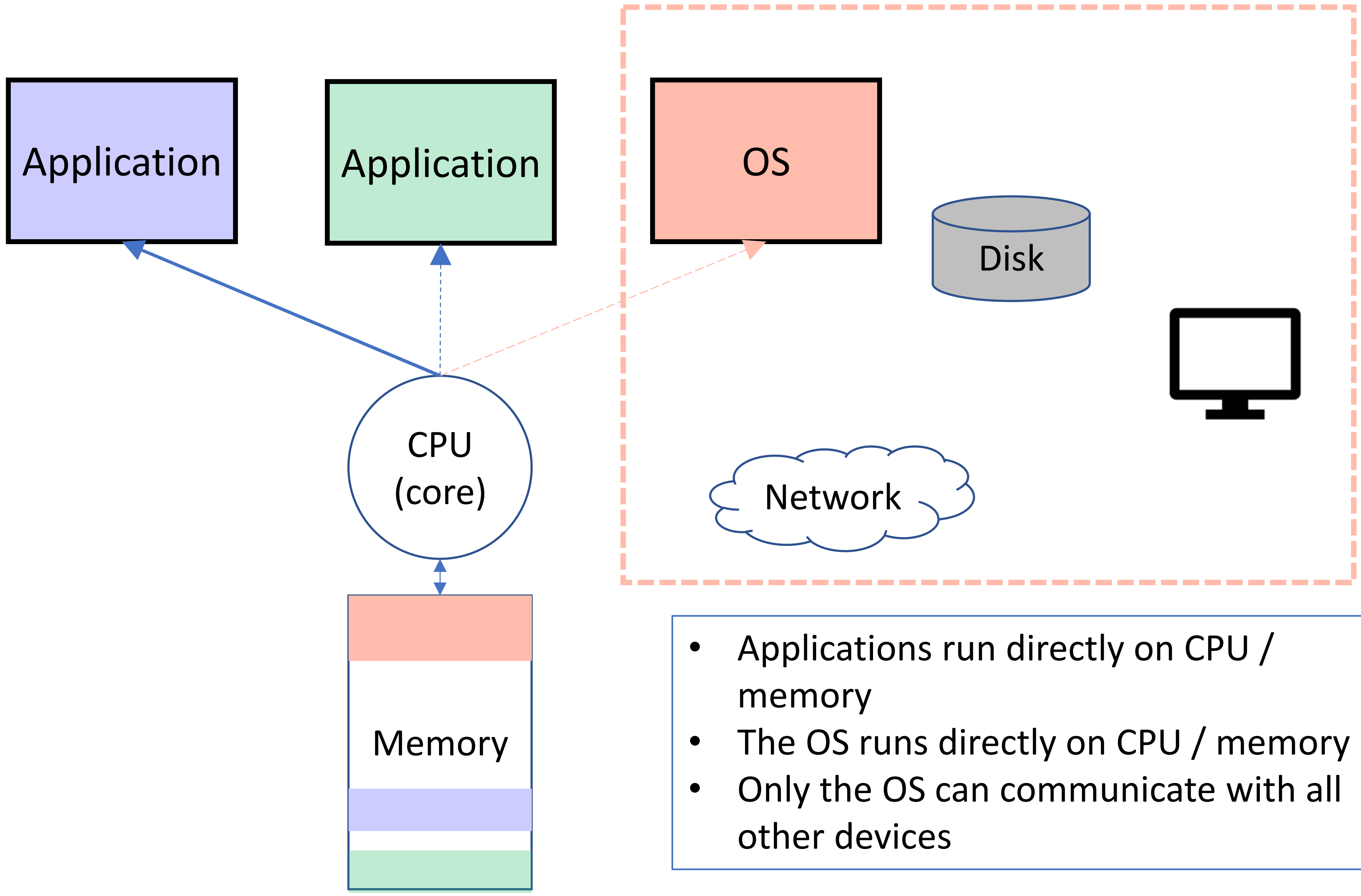
Roles of the Operating System: (1) Security



The Usual Picture
of Runtime

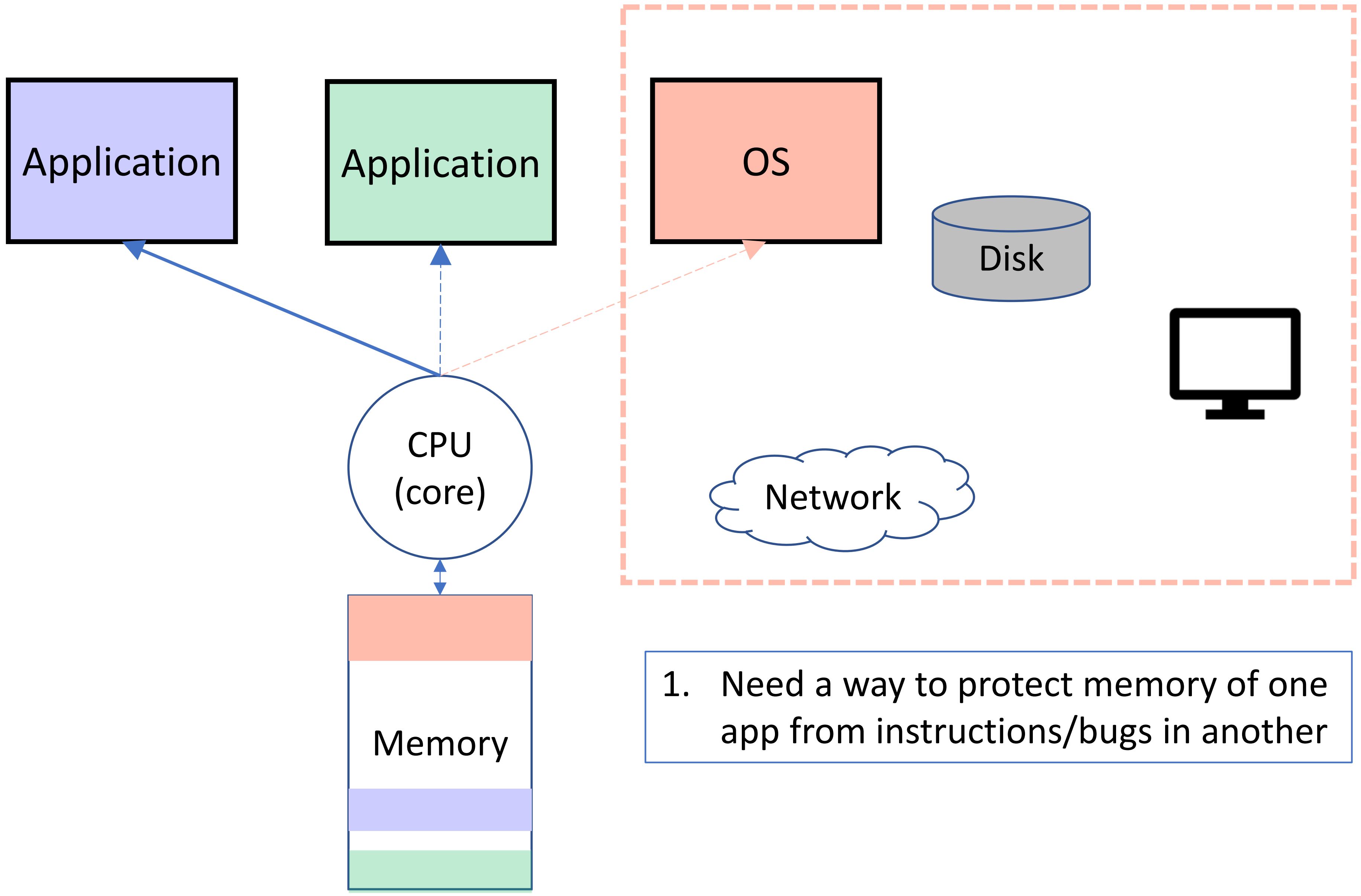
- *"The OS sits between the application and the hardware. The application can't touch the hardware, it can only ask the operating system to touch the hardware for it."*
  - **True**: Application programs can't directly manipulate any hardware and need to ask the OS to do it for them, except for
  - **False**: Application instructions run directly on the CPU and directly read and write main memory

Roles of the Operating System: Security



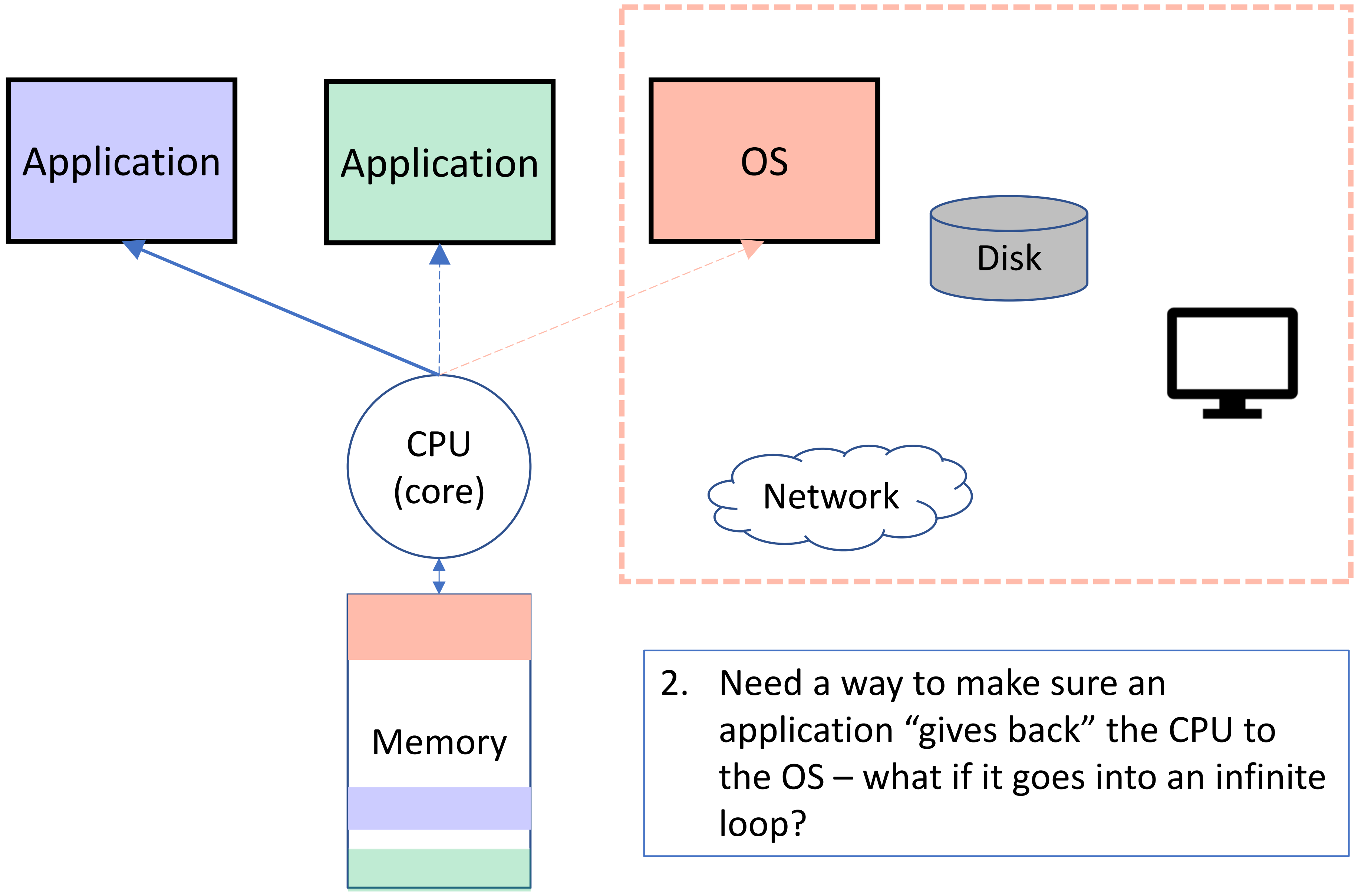
- Applications run directly on CPU / memory
- The OS runs directly on CPU / memory
- Only the OS can communicate with all other devices

CPU/Memory Security: Implications



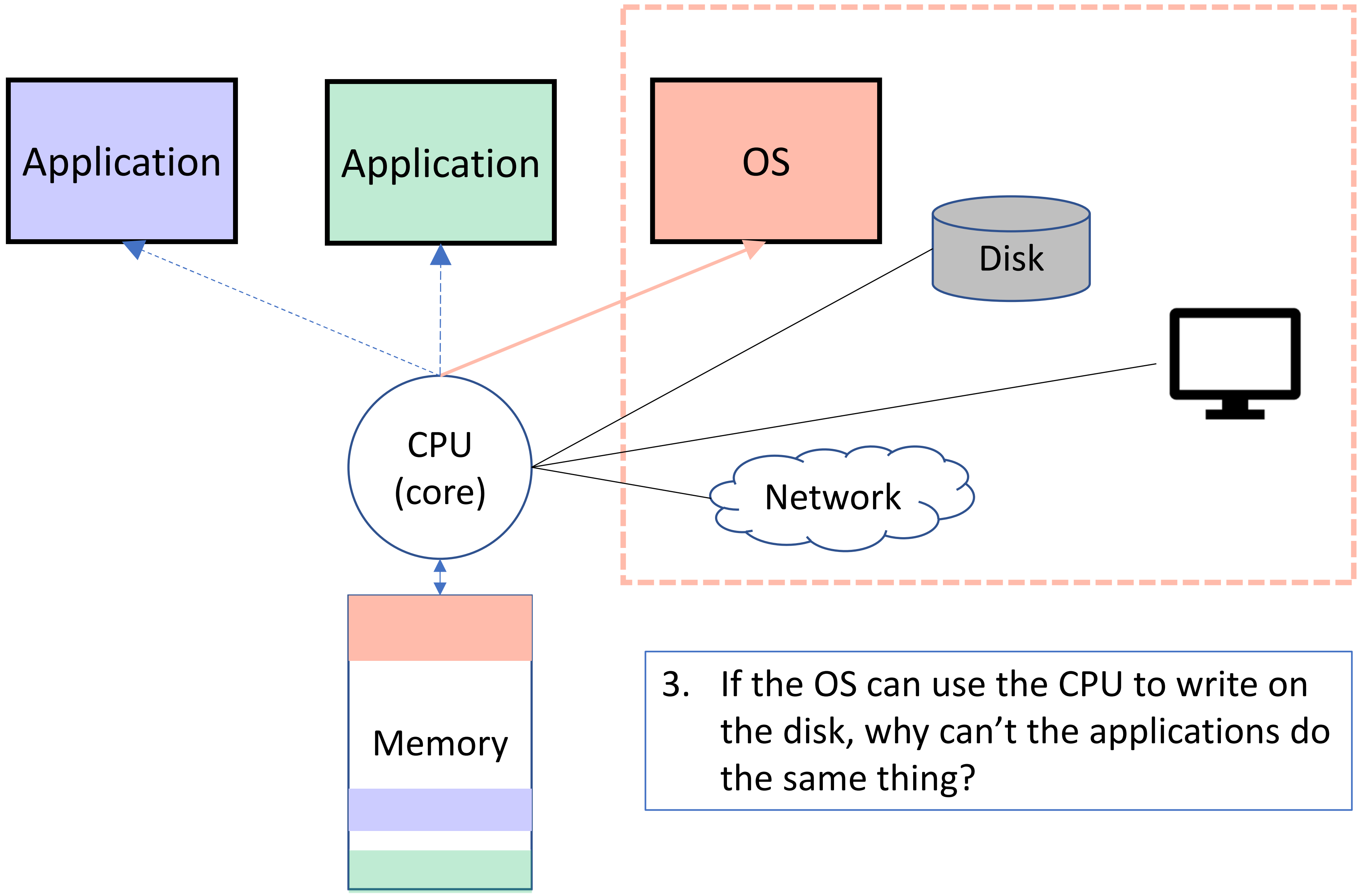
1. Need a way to protect memory of one app from instructions/bugs in another

CPU/Memory Security: Implications



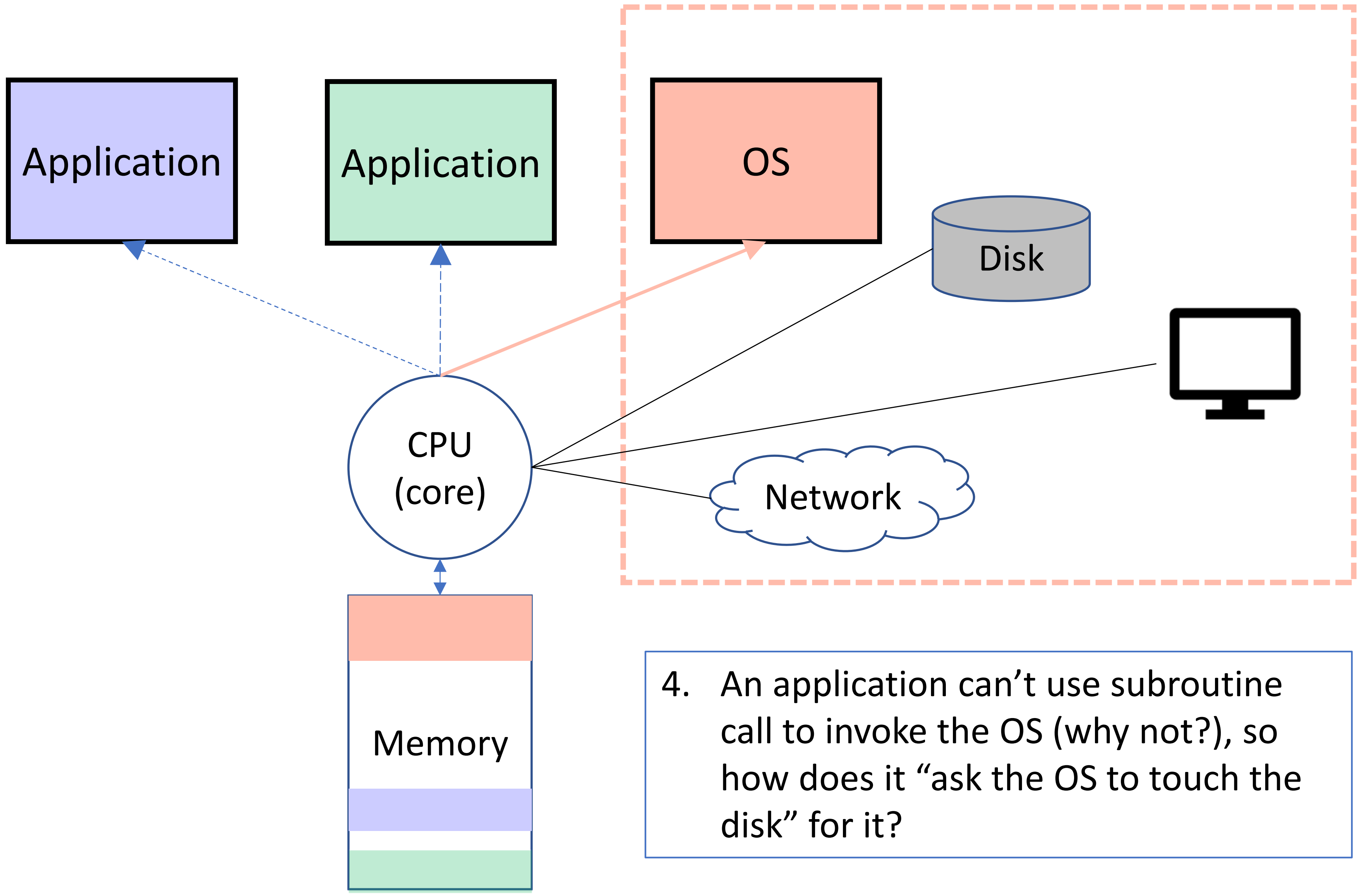
2. Need a way to make sure an application "gives back" the CPU to the OS – what if it goes into an infinite loop?

CPU/Memory Security: Implications



3. If the OS can use the CPU to write on the disk, why can't the applications do the same thing?

CPU/Memory Security: Implications



4. An application can't use subroutine call to invoke the OS (why not?), so how does it "ask the OS to touch the disk" for it?

Hardware (ISA) support for OS Implementation

- The security issues just outlined cannot be solved by software alone (that is, without adding features to the ISA we've seen so far)

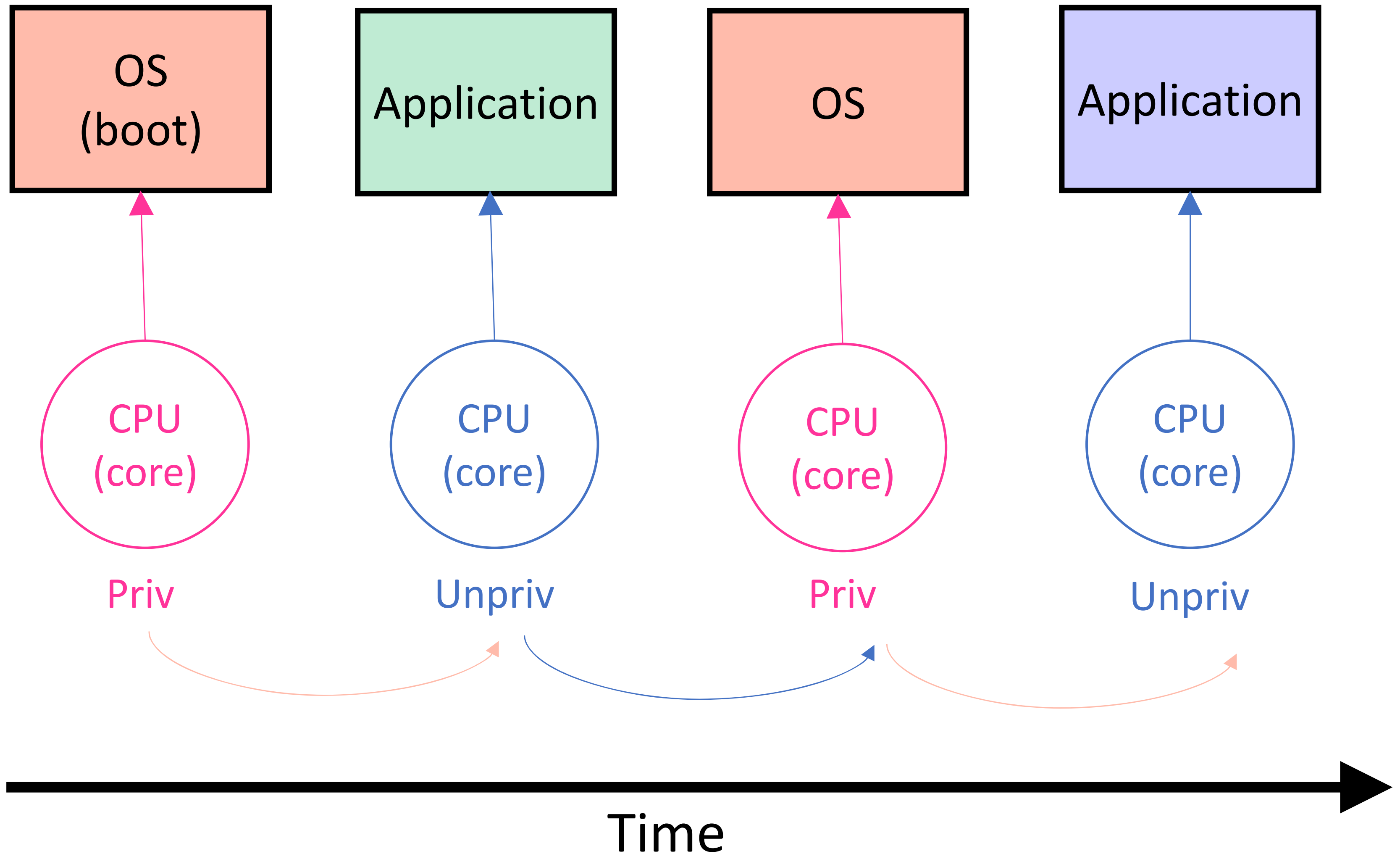
- The first thing we need is a way for the CPU to be able to determine whether the code it is currently running is the OS or just an application
  - If the PC points to an instruction that will write the disk, the CPU should execute it IF the CPU is currently running the OS and it should refuse to execute it otherwise

- **Privileged Instructions**
  - Instructions the CPU will execute if and only if they are issued by the OS code, and not by any other code

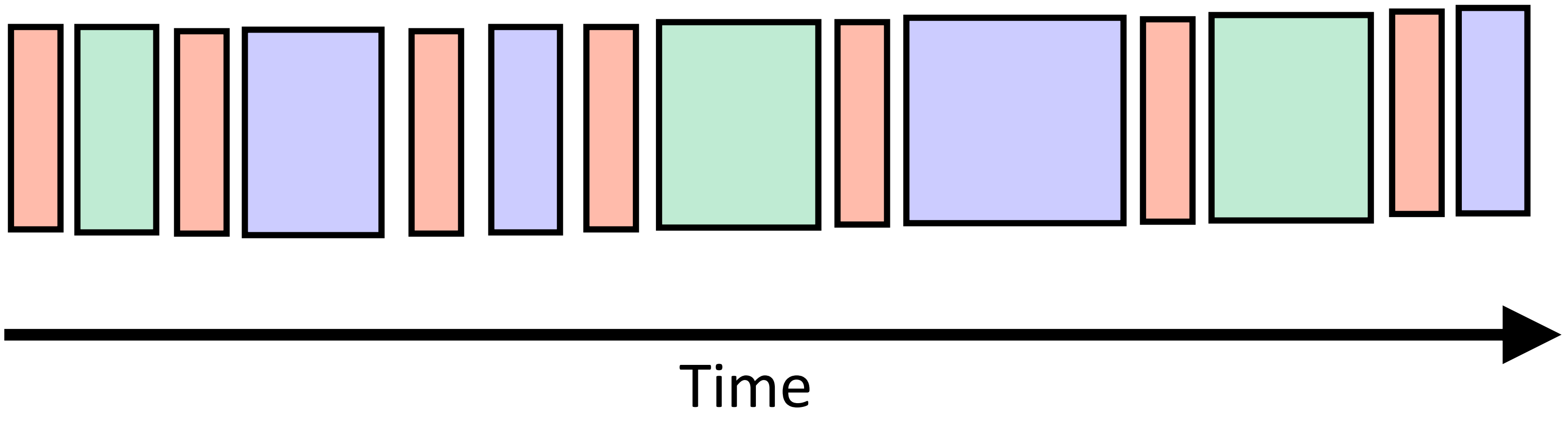
ISA Support for OS: Privileged Mode and Privileged Instructions

- The CPU has additional state (beyond the registers we have tal,mked about) that lets it distinguish between "privileged mode" and "unprivileged mode"
  - Depending on the ISA, there might be a special register that contains a bit indicating the current privilege mode
  - Depending on the ISA, there might be more than just two levels of privilege
- There are special operations (beyond the ones we've talked about) that the CPU is willing to execute only if is currently in privileged mode
  - If not, it raises an error using a mechanism we'll see shortly
- Privileged instructions are a solution to "how can the OS do I/O but applications can't"
  - and other very similar operations

Privileged / Unprivileged Mode



Multiprogramming



- Each core alternates between running the OS and running some application
  - This is called *multiprogramming*
- Each transition involves a change in privilege level
  - Either from privileged to unprivileged (OS to app) or vice versa
- These transitions happen 100’s to 1000’s of times per second
- Gives the illusion that many programs are running at the same time
  - Well, they are, in the way shown
- Each transition is called a *context switch*

Context Switching: Execution State

- When an application’s execution is suspended, all of the core’s registers and the PC are saved
- When the OS decides to resume that application’s execution, it reloads the registers with their saved values and branches to the saved PC
- Because the memory used by the application isn’t changed while it’s suspended, when it resumes execution the application executes exactly as if it hadn’t been suspended
    - (unless it looks at a real time clock...)

Context Switching: Privilege Mode

- The return from the OS to resumption of the application code involves a transition from privileged to unprivileged mode
- **Easy** – the OS executes an instruction that turns off privilege

- The transition from the application to the OS involves a transition from unprivileged to privileged mode
- How can that happen? If unprivileged code can cause a transition to privileged mode, what keeps it from doing so whenever it wants for whatever reason it wants?

Context Switching: The Trap Mechanism

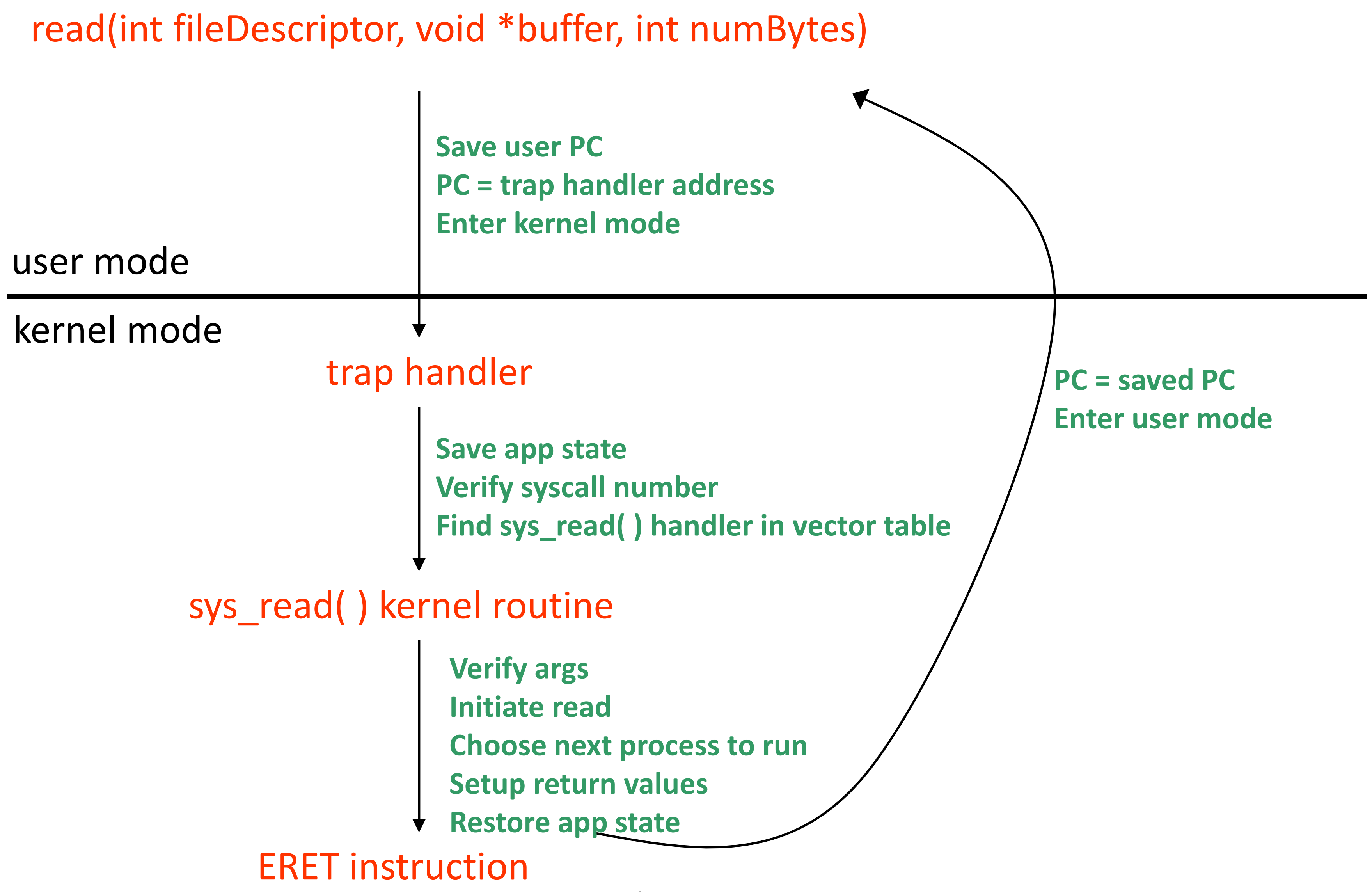
- Unprivileged code **can** cause a transition to privileged mode whenever it wants
  - There's an instruction for that
- **BUT** the unprivileged code cannot control what instruction is executed in the first cycle after privilege is acquired
  - The OS determines that
- The "trap mechanism" does these, indivisibly in one cycle:
  - Saves the current PC somewhere well known, then
  - Loads the PC with an address stored in somewhere only the OS can write (e.g., a "privileged register")
  - Sets the CPU in privileged mode
- The OS sets the "trap handler address" during boot. After that, any transition from unprivileged to privileged mode causes a branch to the address of a handler routine established by the OS

Trap Handling: System Calls

- A “system call” is a lot like a subroutine call to the operating system, except that privilege is acquired on the way
- Because the “call” cannot specify the address in the operating system at which to execute (because it must execute starting at the trap handler address), “what method to call” must be an explicit argument

- To system call, do this:
  - Put a system call number in a register (say, a7). That indicates what you are trying to do (e.g., open a file)
  - Put additional arguments, depending on the call, in registers a0-a6
  - Execute a trap instruction
  - When the call returns, look for return value in, say, a0

A kernel crossing illustrated



Security Recap to This Point

1. Need a way to protect memory of one app from instructions/bugs in another
   We don’t yet know how to do this

2. Need a way to make sure an application “gives back” the CPU to the OS – what if it goes into an infinite loop?
   We don’t yet know how to do this

3. If the OS can use the CPU to write on the disk, why can’t the applications do the same thing?
   Privilege vs. unprivileged mode and privileged instructions

4. An application can’t use subroutine call to invoke the OS (why not?), so how does it “ask the OS to touch the disk” for it?
   It uses the trap mechanism

Getting the CPU Back

- Each core alternates between running CPU code and running application code
  - The OS chooses which application to “dispatch” next
  - Making that decision is called “(CPU) scheduling”
- When an application is dispatched, it is in control of the CPU
- IF it makes a system call, then the OS gets a chance to execute and make a new scheduling decision
- What if the application code doesn’t make a system call? What if it instead does something compute intensive for a few seconds/hours/days?

Getting the CPU Back: count-down timer

- We can’t rely on the applications to cause a transition back to the operating system while they’re running
- So, we need a way to guarantee that transition happens without relying on code
  - So, need hardware support
- Besides the registers, PC, privilege mode bit(s), and trap handler address information, CPU’s also have a (count-down) **timer**

Getting the CPU Back: count-down timer

- When the OS dispatches application code, it sets the timer to some value (say, 10 msec.)

- If the application makes a system call before 10 msec. has expired, great
  - The OS is entered, gets a chance to make a scheduling decision, and resets the timer (to 10 msec., say)

- Otherwise, when 10 msec. has gone by, the timer will have counted down to 0
- When it does, it “raises an interrupt”
  - That invokes the trap handler mechanism, which
  - Causes the cpu to transition to privileged mode and to start executing the OS trap handler

More Vocabulary Than You Need

- The trap handler mechanism is invoked in three circumstances, and each has a specific name (although people are often not careful about which they use)
  - A **trap** is an intentional invocation of the mechanism, to make a system call
  - An **interrupt** is an "asynchronous" execution of the trap handling mechanism, because some I/O device wants attention for some reason
    - E.g., the timer
    - E.g., data has arrived on the network
  - An **exception** is a "synchronous" execution of the trap handler mechanism because the hardware has detected some problem with the execution of a particular instruction
    - E.g., trying to execute a privileged instruction when in unprivileged mode

Key Ideas

- One role of the operating system is security
  - Protecting one application from bugs or intentional violation by another
- For performance reasons, the OS must allow applications to run directly on the CPU and memory hardware
- We need a memory protection mechanism (comes later in the course)
- CPU protection is provided by:
  - having privileged and unprivileged CPU modes
  - having privileged instructions that raise exceptions when executed unless the CPU is in privileged mode
  - a trap handling mechanism that allows the CPU to (re)gain control of the CPU, to save the state of the application that was running, and to late restore the state of the application and start it running back where it was, as though it had never stopped
  - a countdown timer that can raise an interrupt