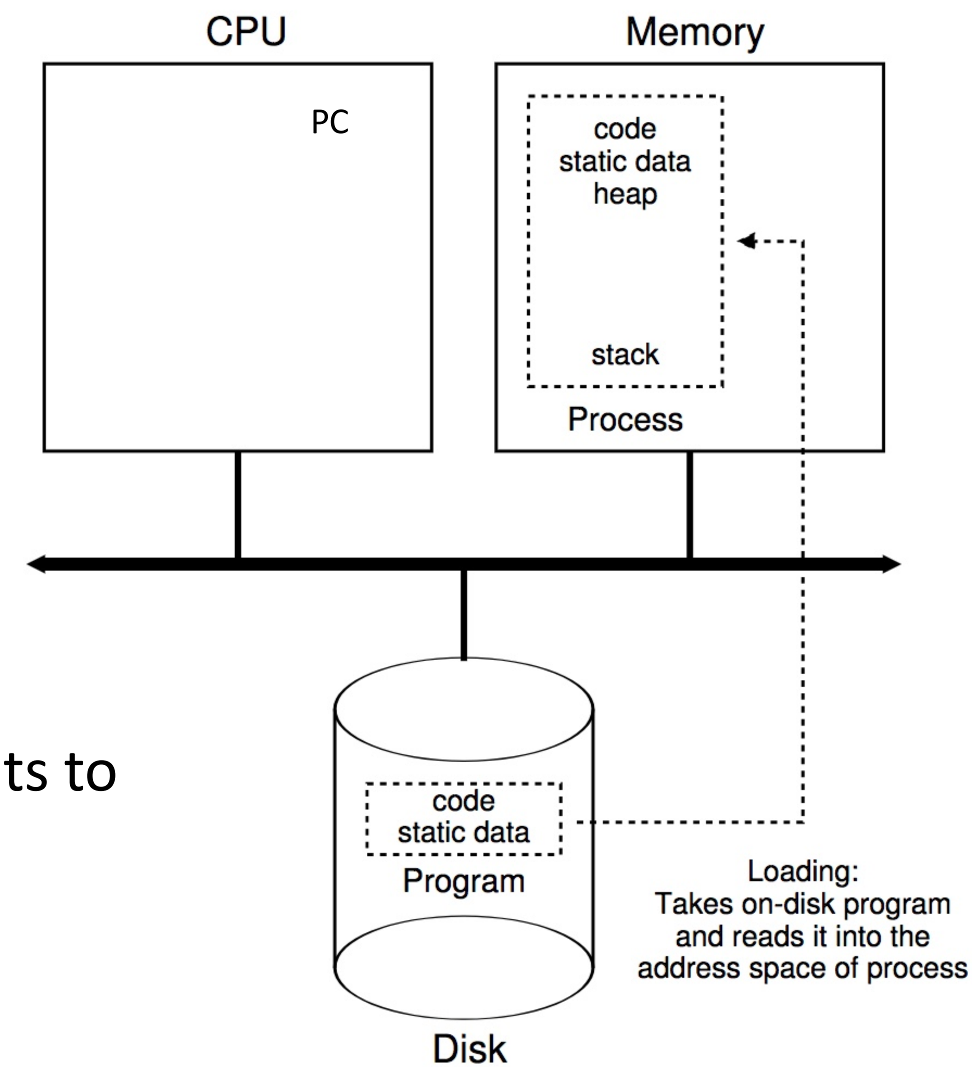
CPU Virtualization: Limited Direct Execution (LDE)

CS 571: Operating Systems (Spring 2021)

Lecture 2a

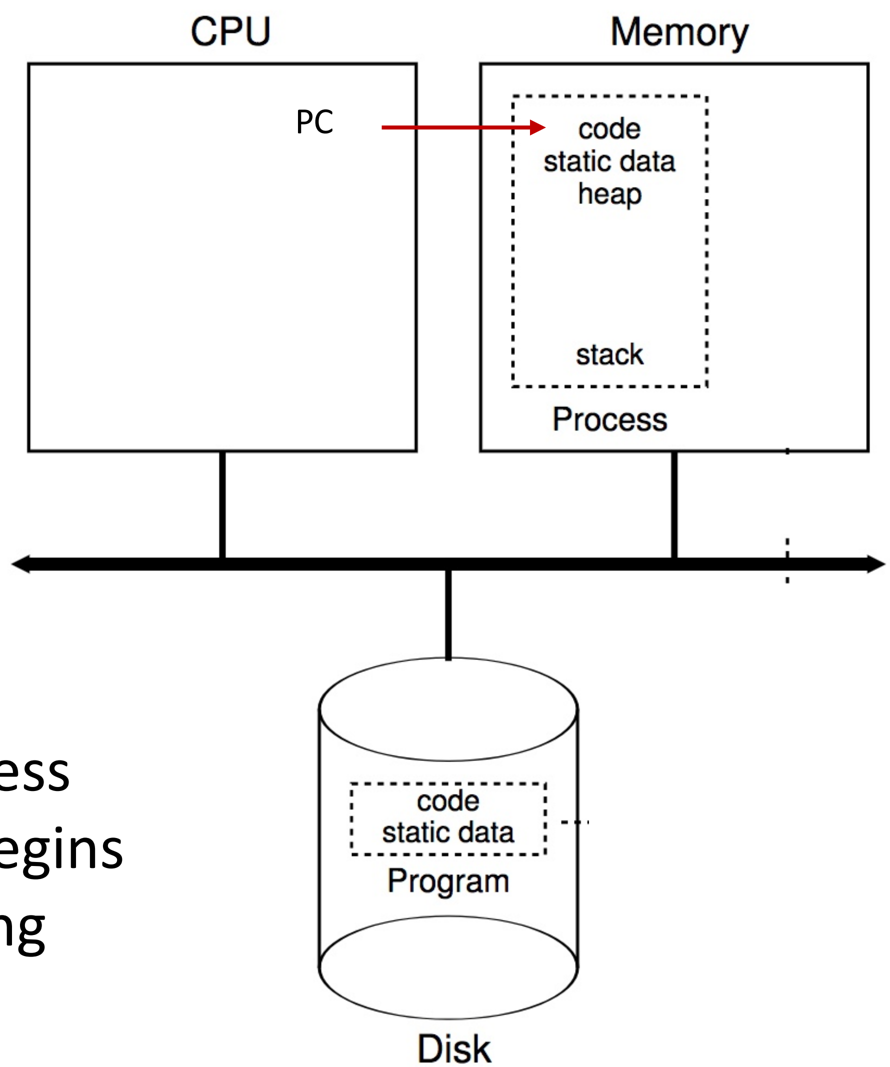
Yue Cheng

Process Creation



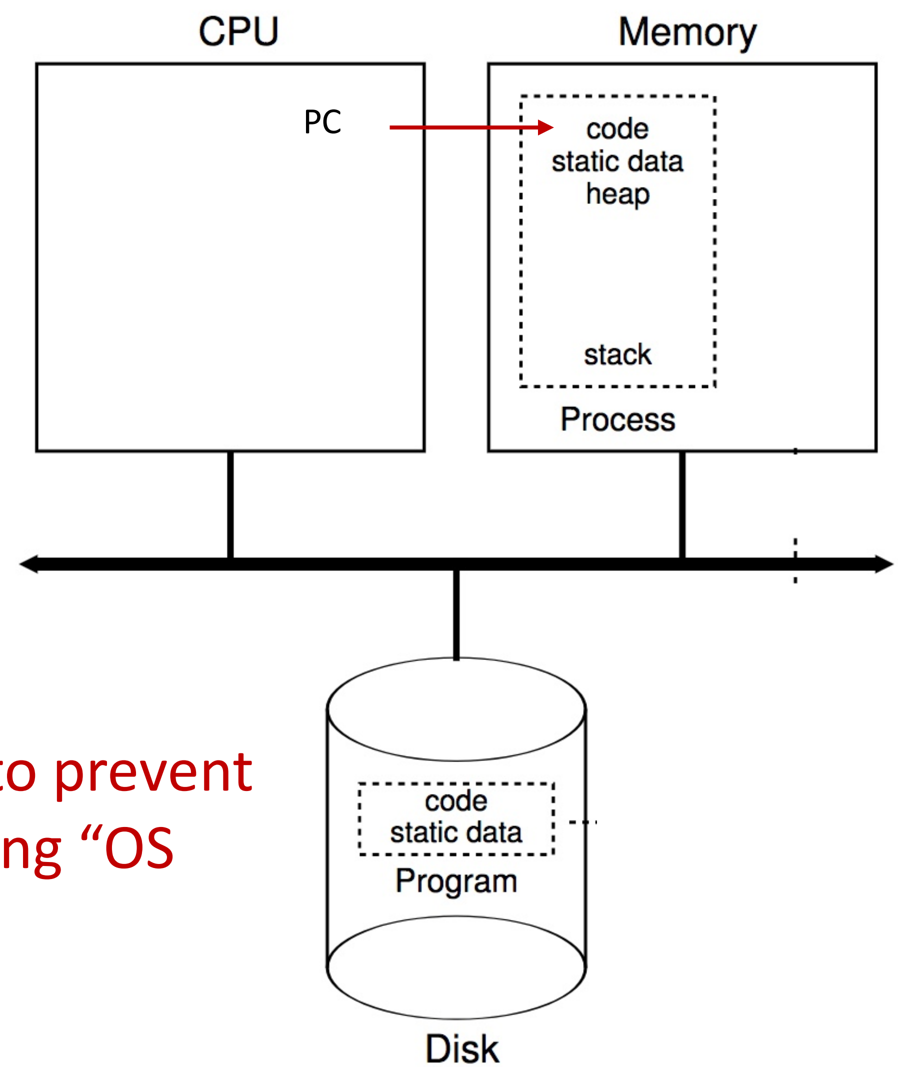
Before, PC points to kernel code

Process Creation



Now, after process creation, CPU begins directly executing process code

Process Creation



Challenge: how to prevent process from doing "OS kernel stuff"?

Limited Direct Execution (LDE)

Limited Direct Execution (LDE)

- Low-level mechanism that implements the user-kernel space separation

- Usually let processes run with no OS involvement
- Limit what processes can do
- Offer privileged operations through well-defined channels with help of OS

Limited Direct Execution (LDE)

Limited Direct Execution (LDE)

User-level process

LDE
mechanism

OS

What to limit?

- General memory access
- Disk I/O
- Certain x86 instructions

How to limit?

- Need hardware support
- Add additional execution mode to CPU

- User mode: restricted, limited capabilities
- Kernel mode: privileged, not restricted

- Processes start in user mode
- OS starts in kernel mode

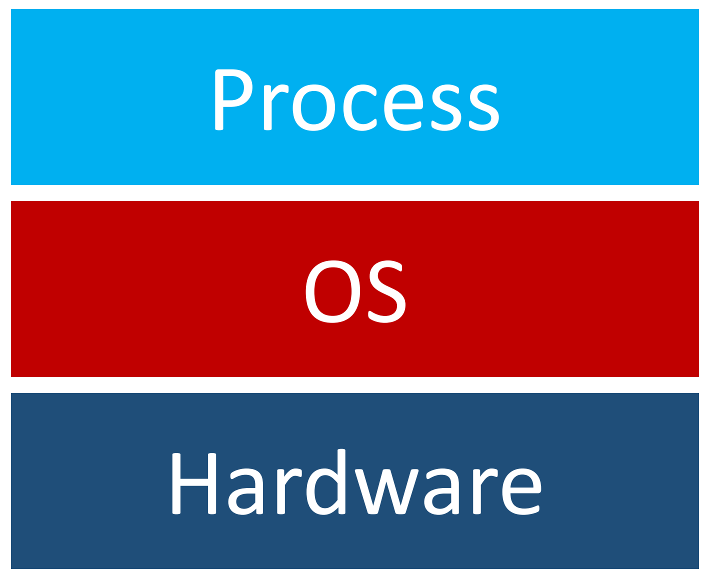
LDE: Remaining Challenges

1. What if process wants to do something privileged?
2. How can OS switch processes (or do anything) if it's not running?

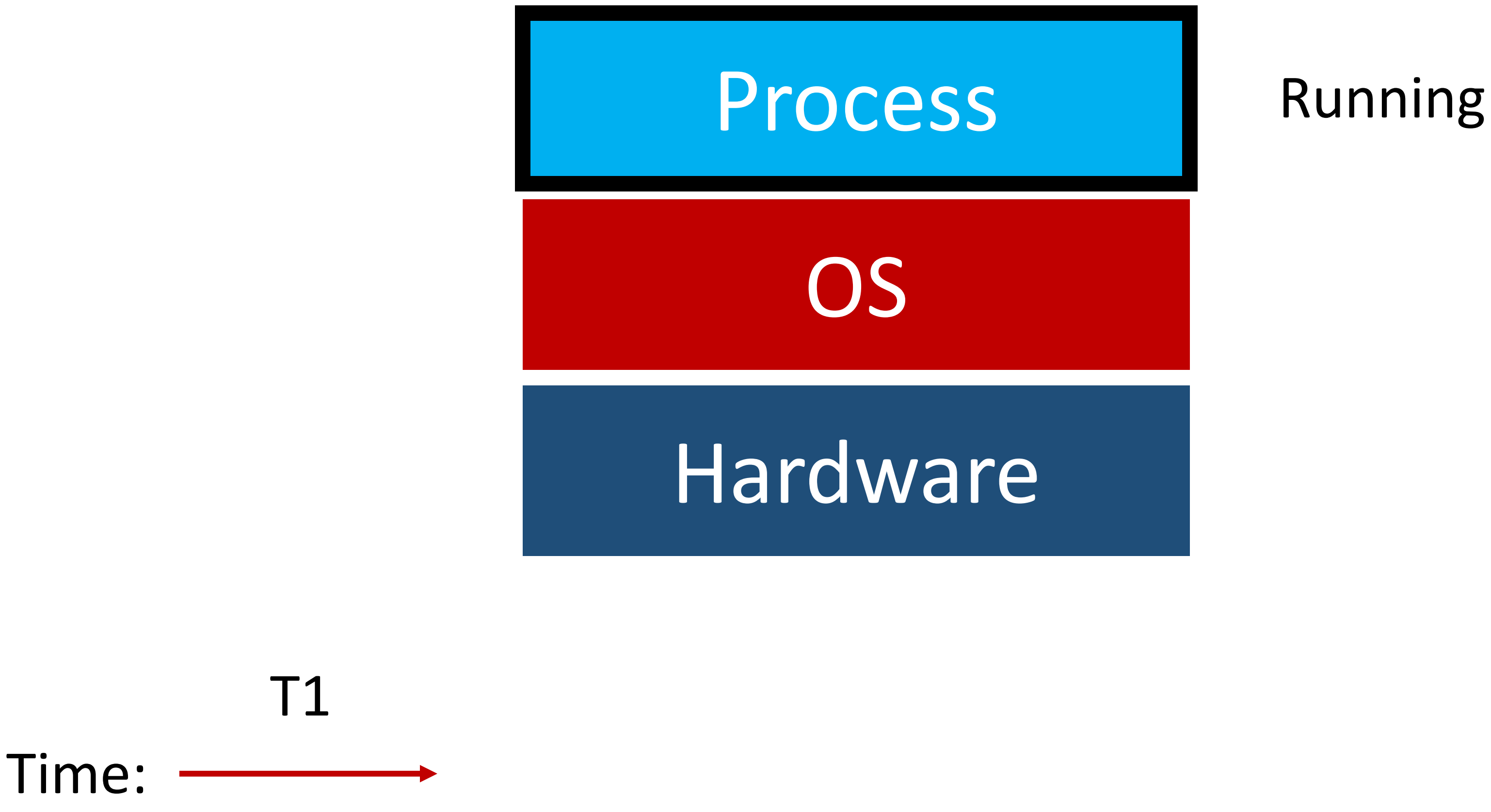
LDE: Remaining Challenges

1. What if process wants to do something privileged?
2. How can OS switch processes (or do anything) if it’s not running?

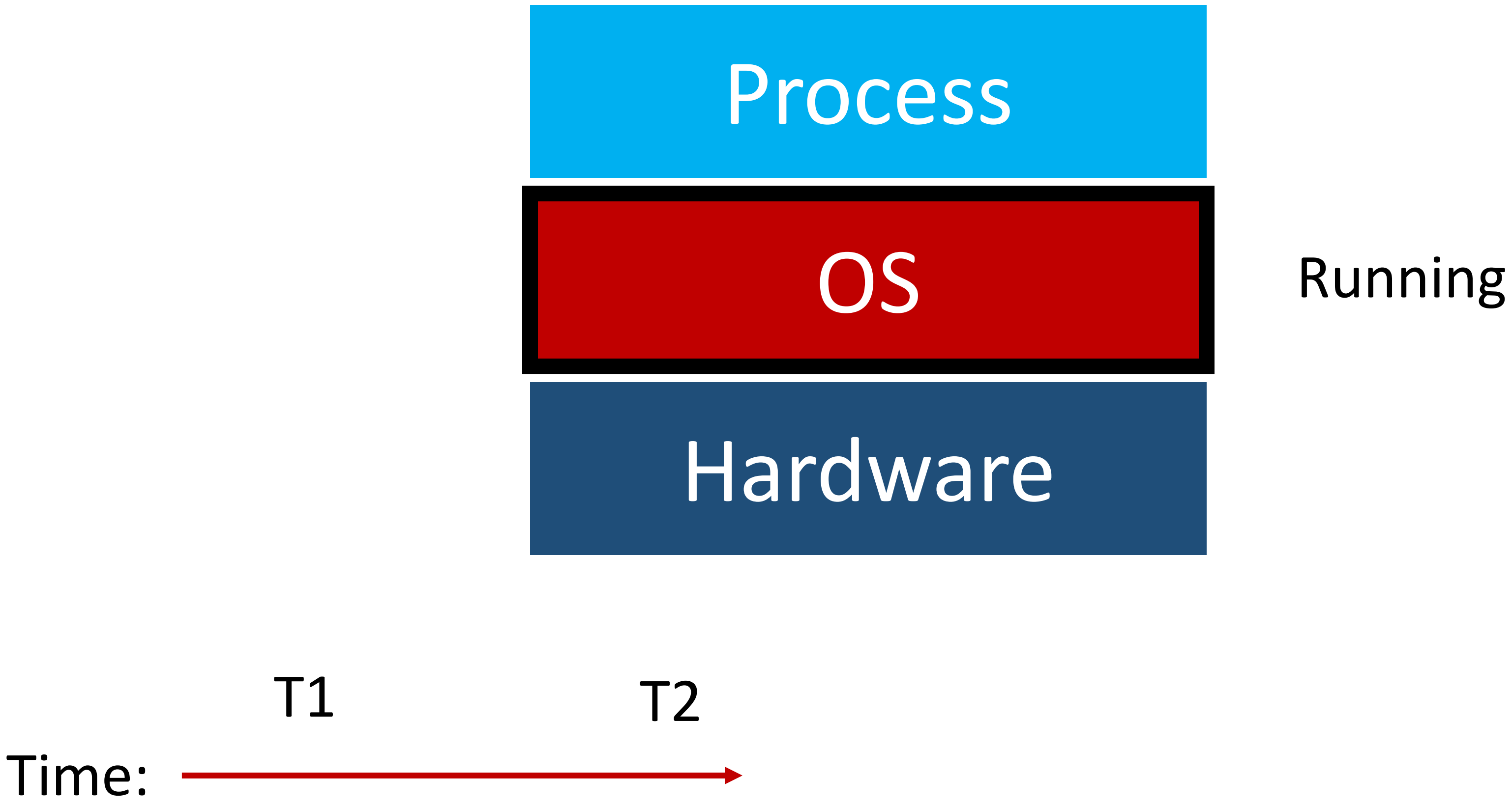
Taking Turns



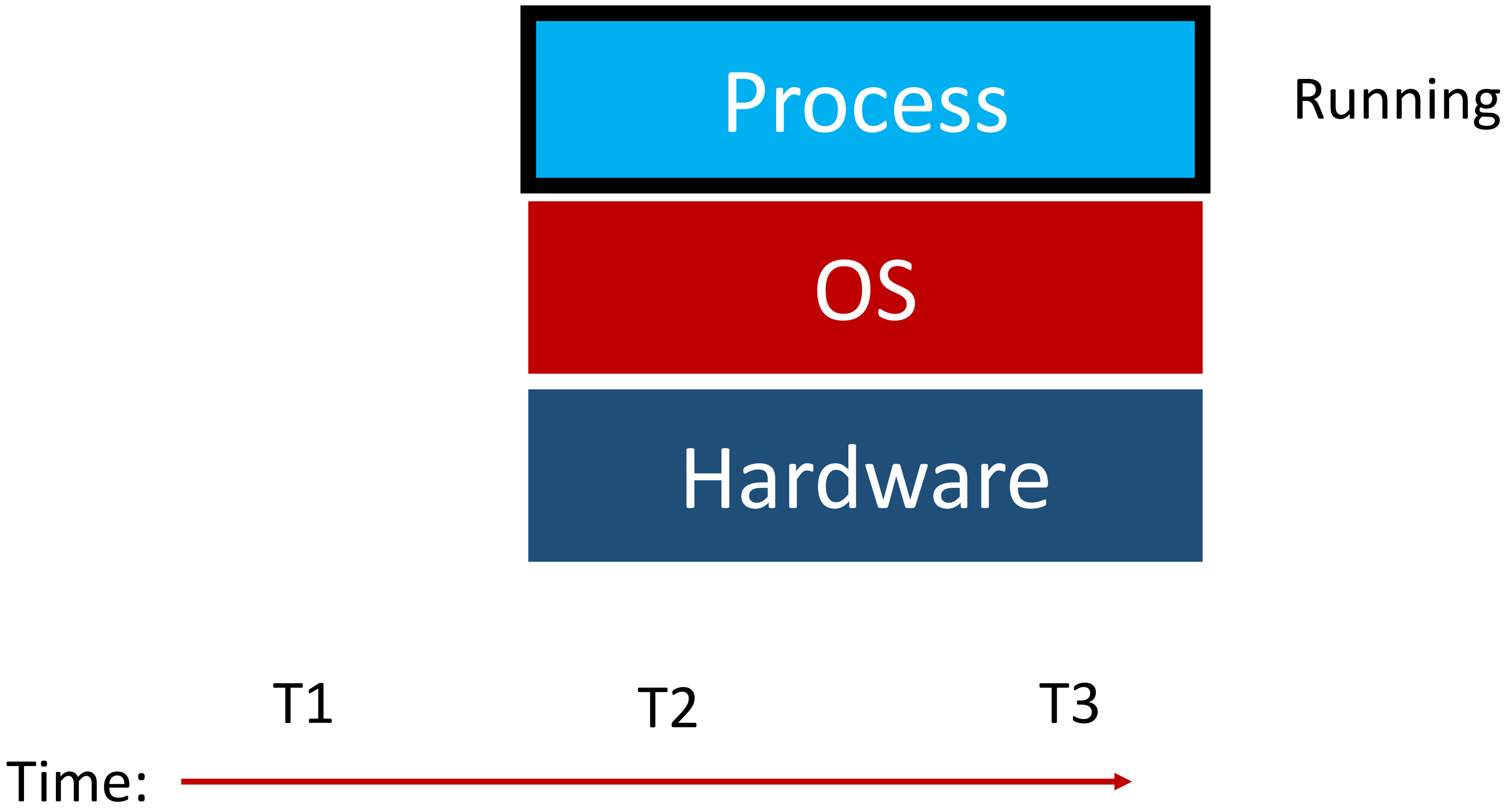
Taking Turns



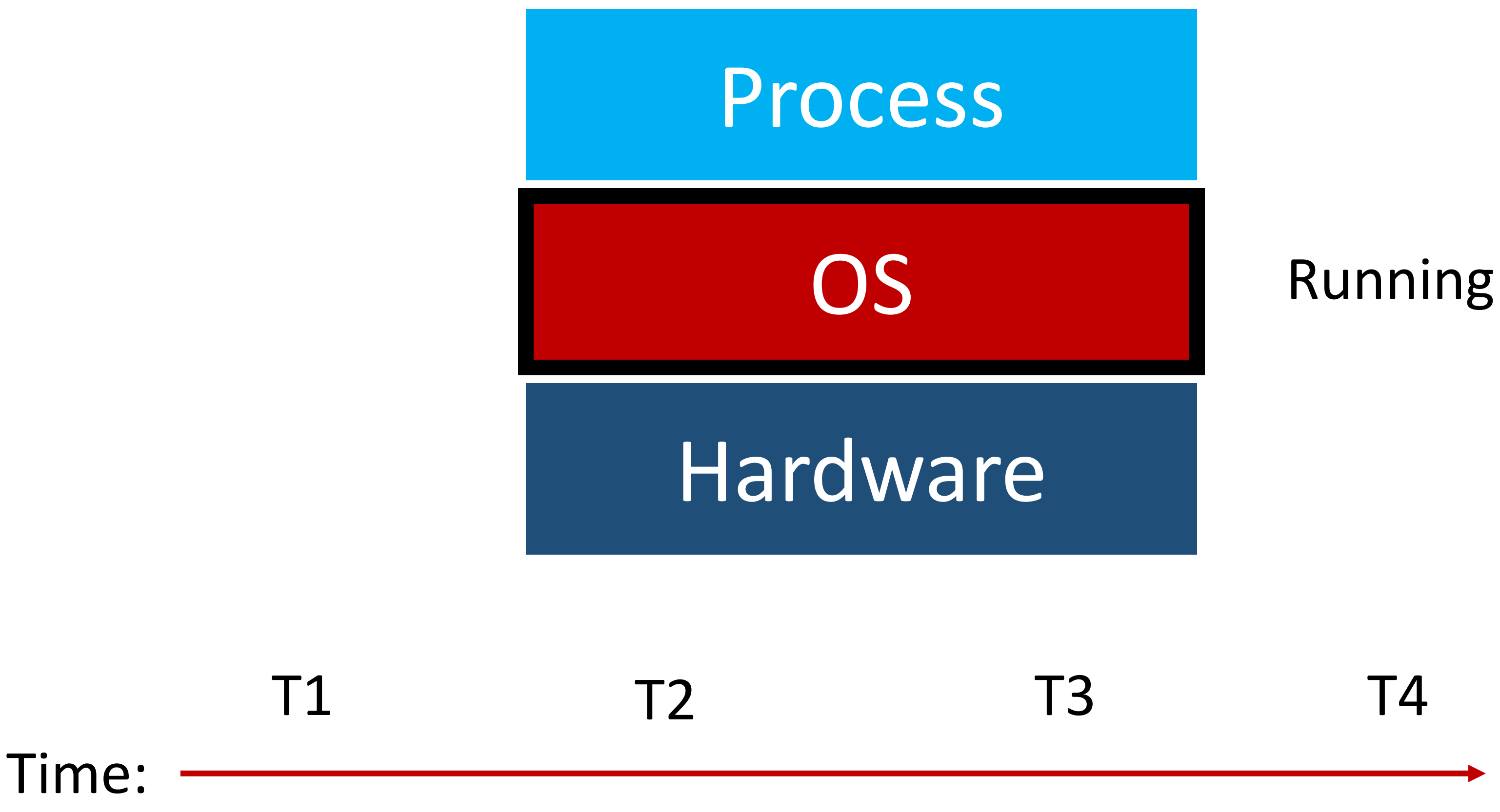
Taking Turns



Taking Turns

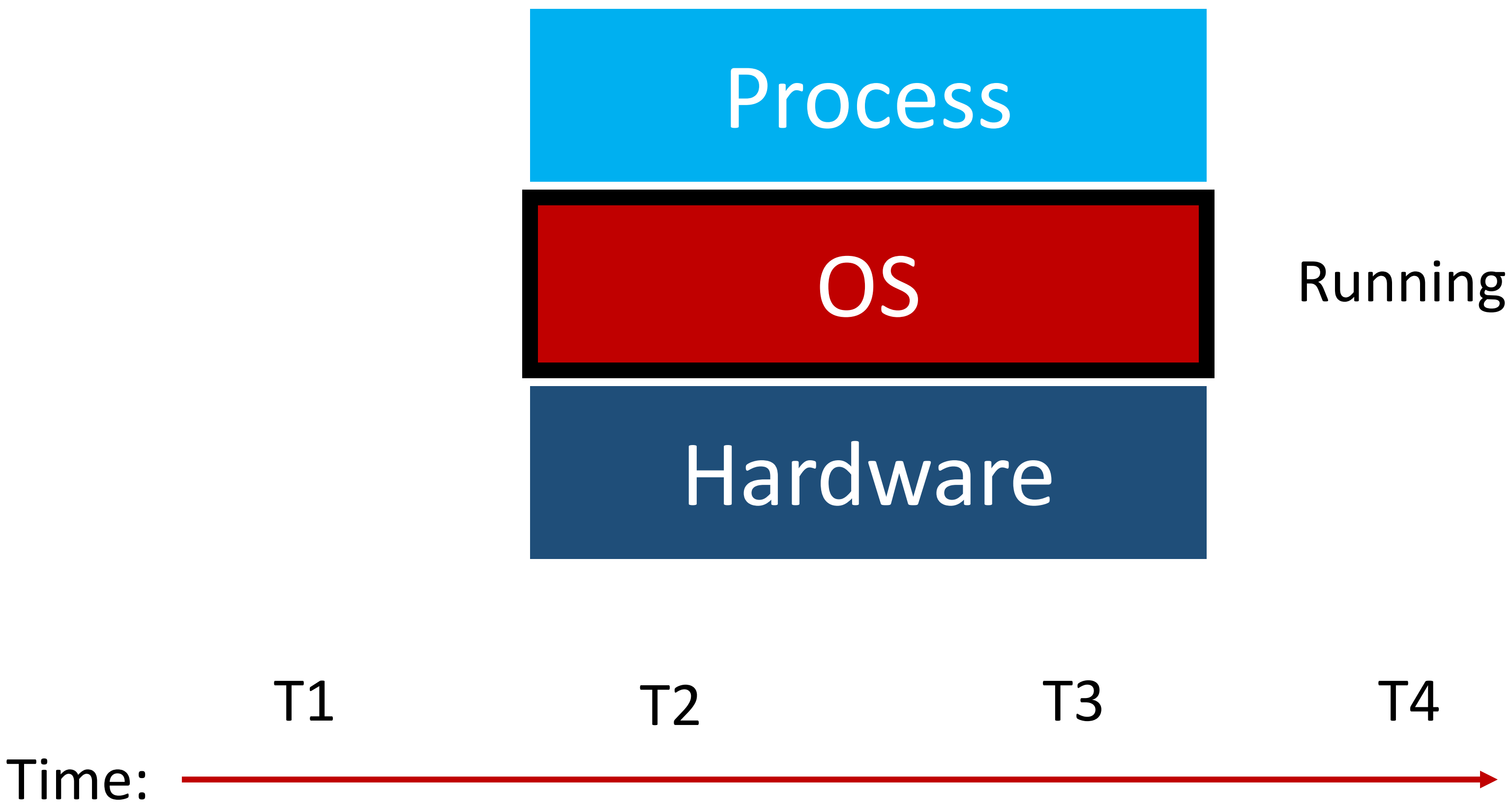


Taking Turns



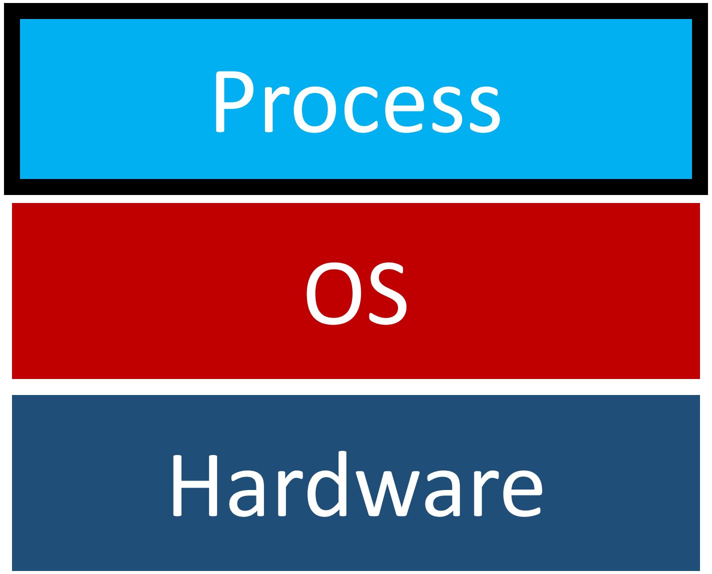
Taking Turns

Question: when/how do we switch to OS?

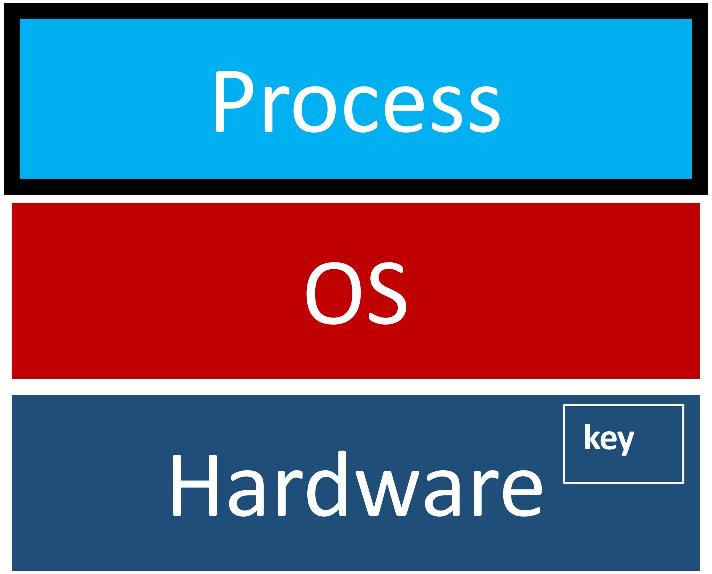


Exceptions

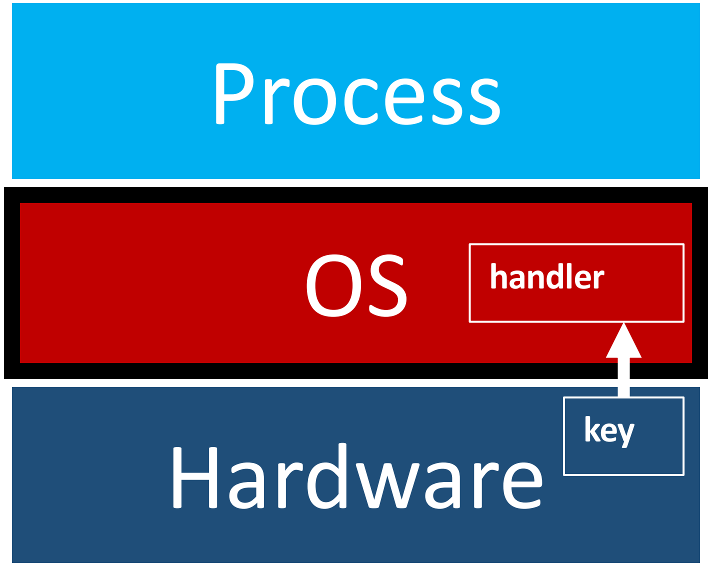
Interrupt



Interrupt

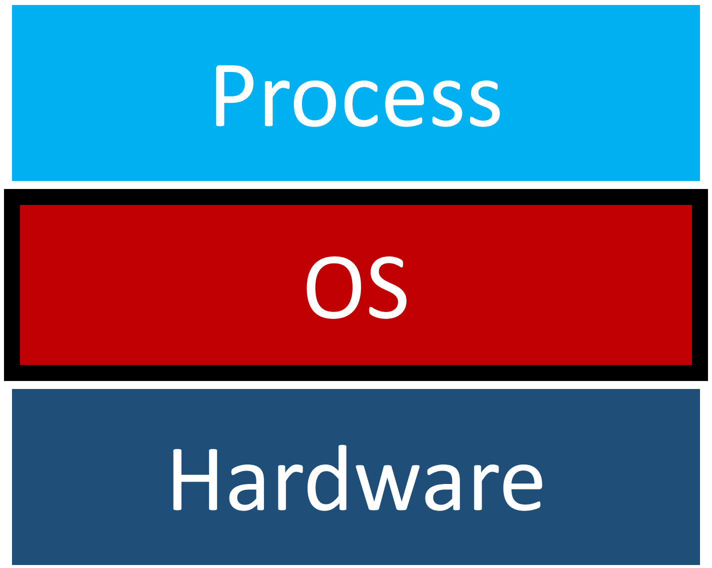


Interrupt

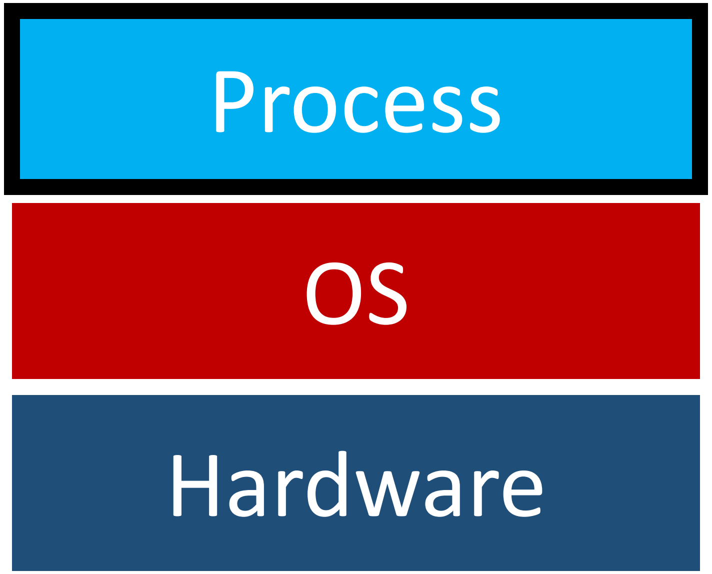


Hardware interrupt

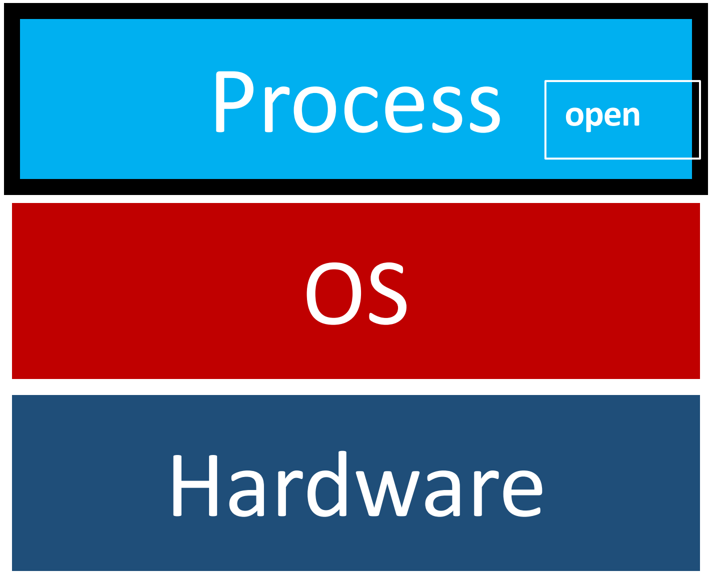
Interrupt



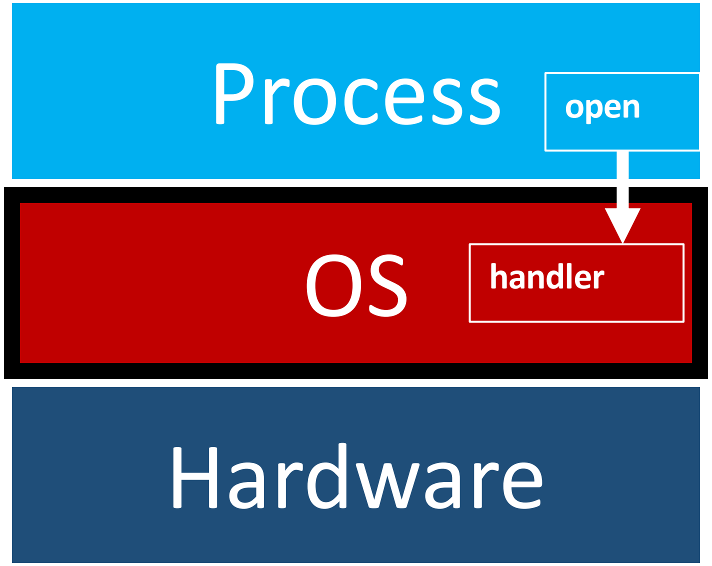
System Call



System Call

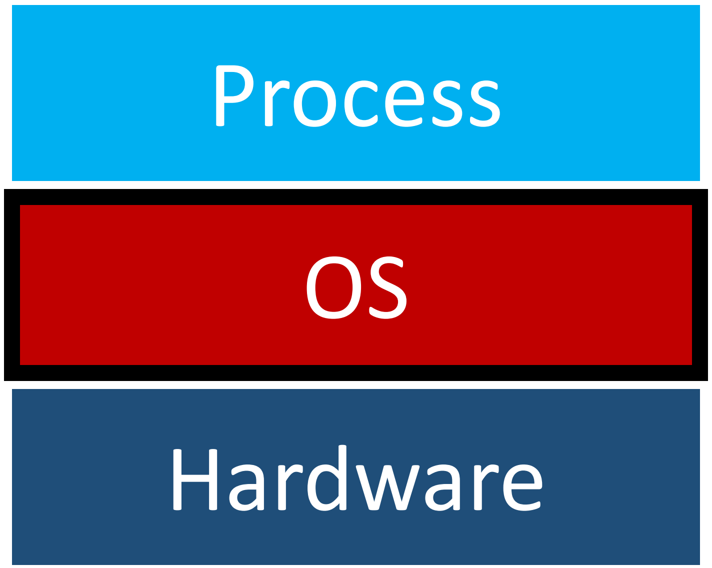


System Call



System call "trap"

System Call

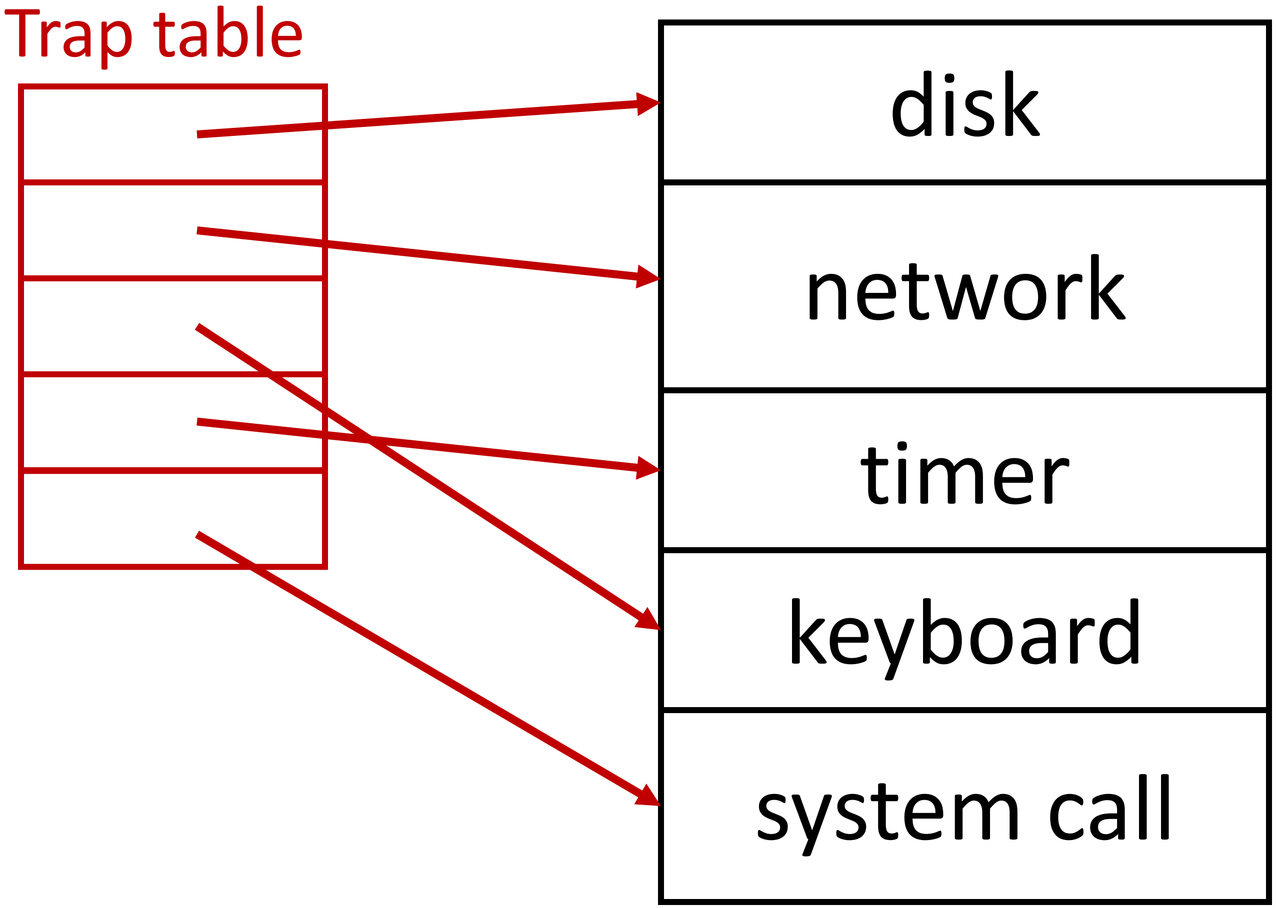


Exception Handling

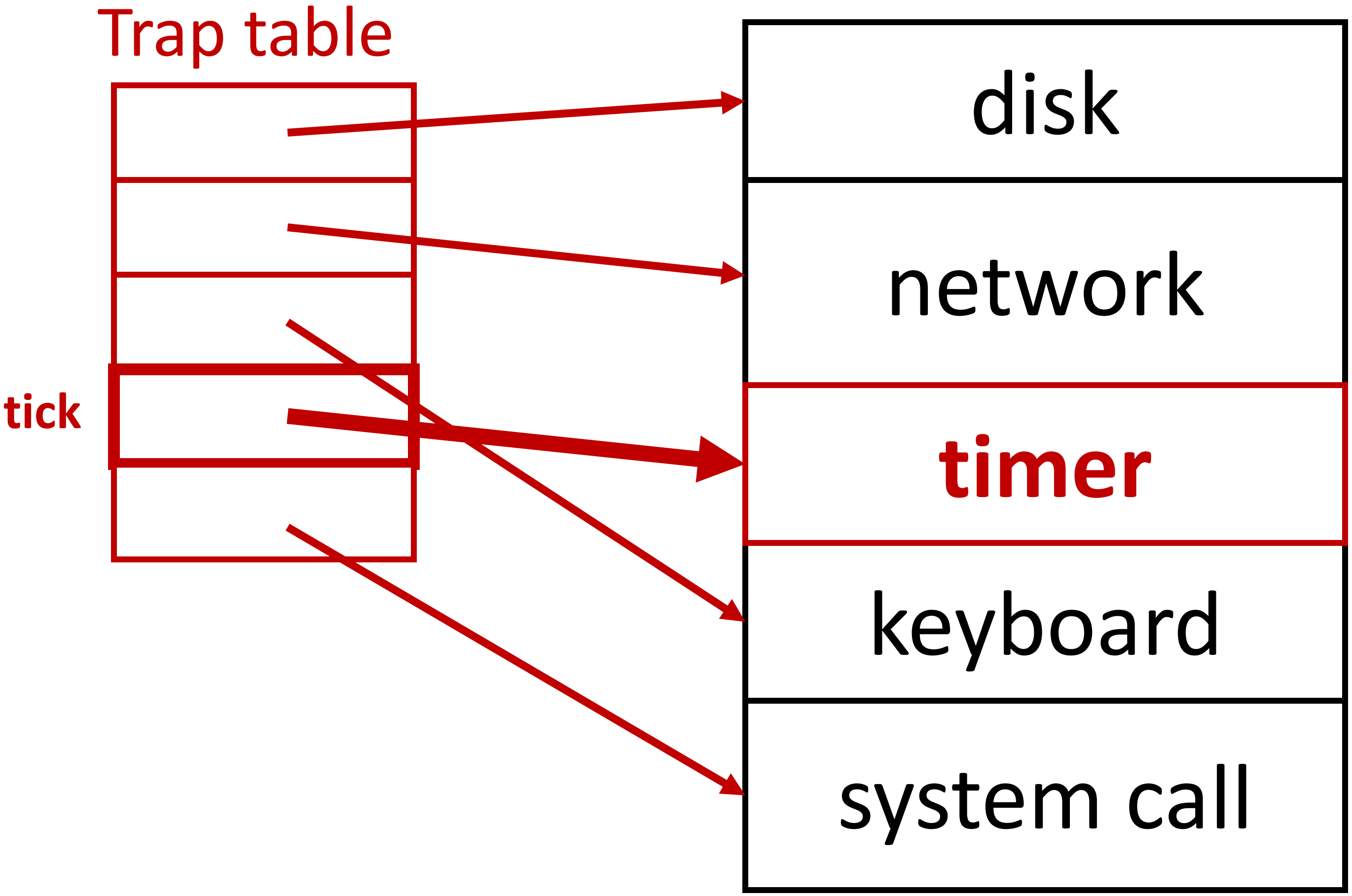
Exception Handling: Implementation

- Goal: Processes and hardware should be able to call functions in the OS

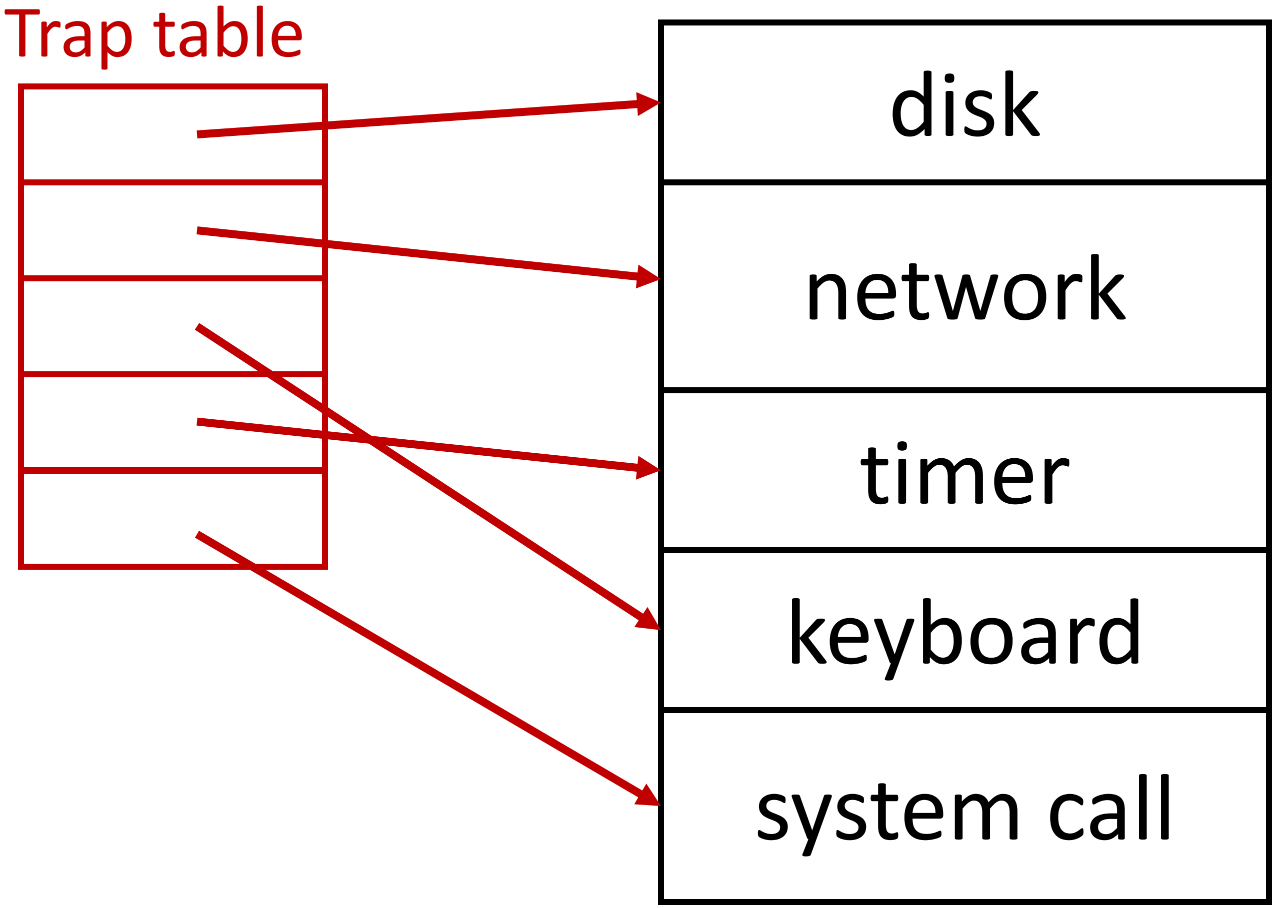
- Corresponding OS functions should be:
  - At well-known locations
  - Safe from processes



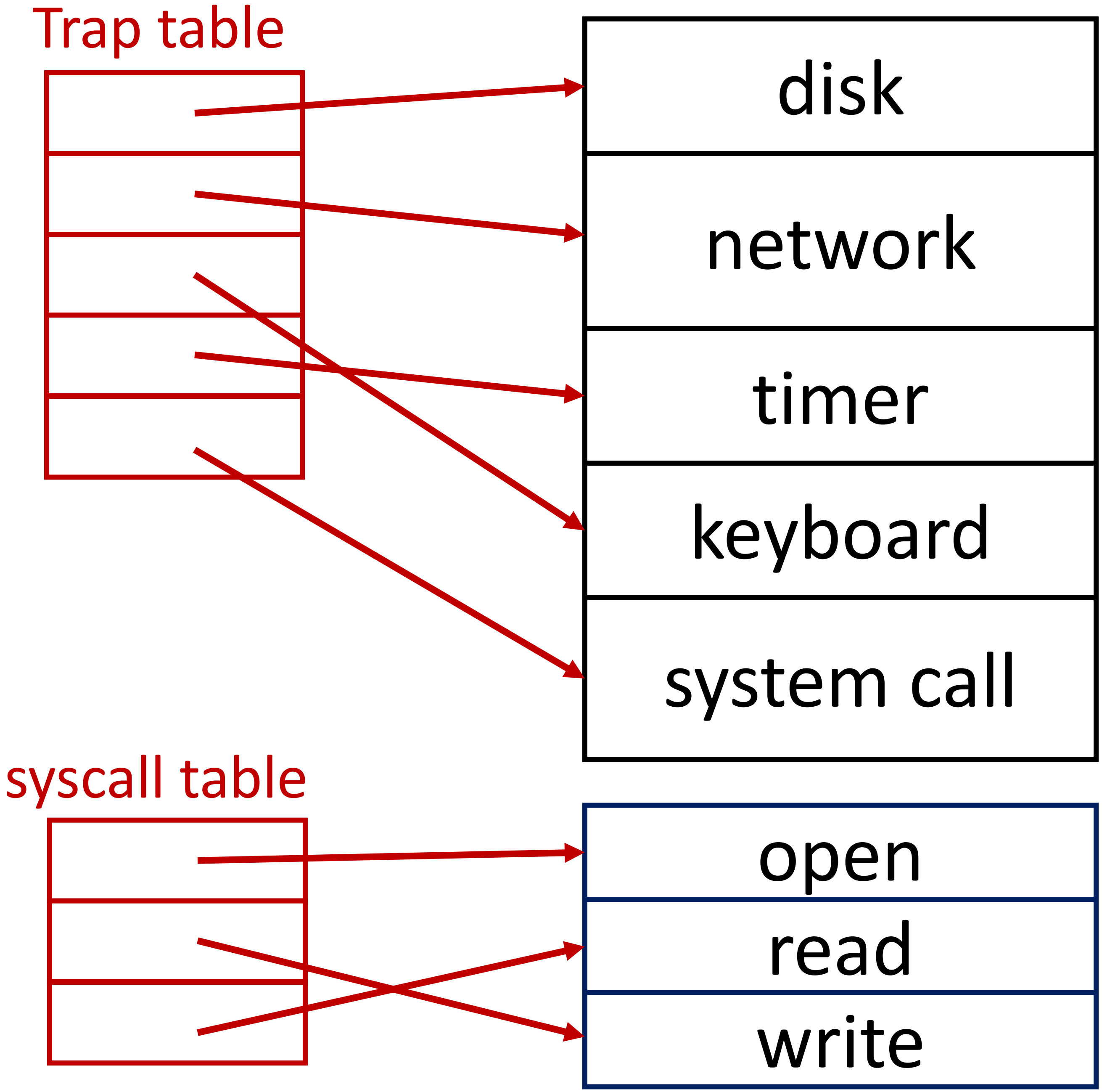
Use array of function pointers to locate OS functions
(**Hardware knows where this is**)

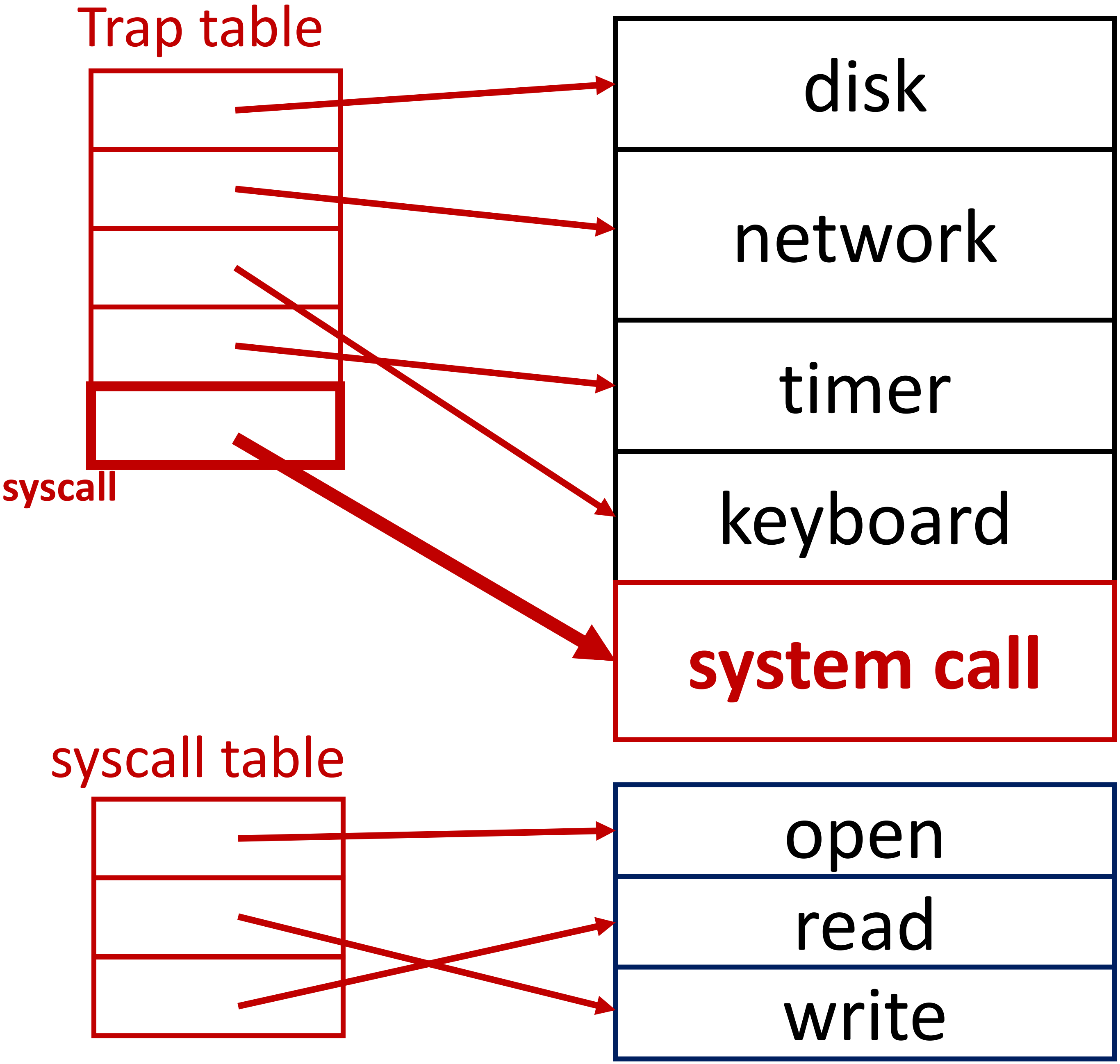


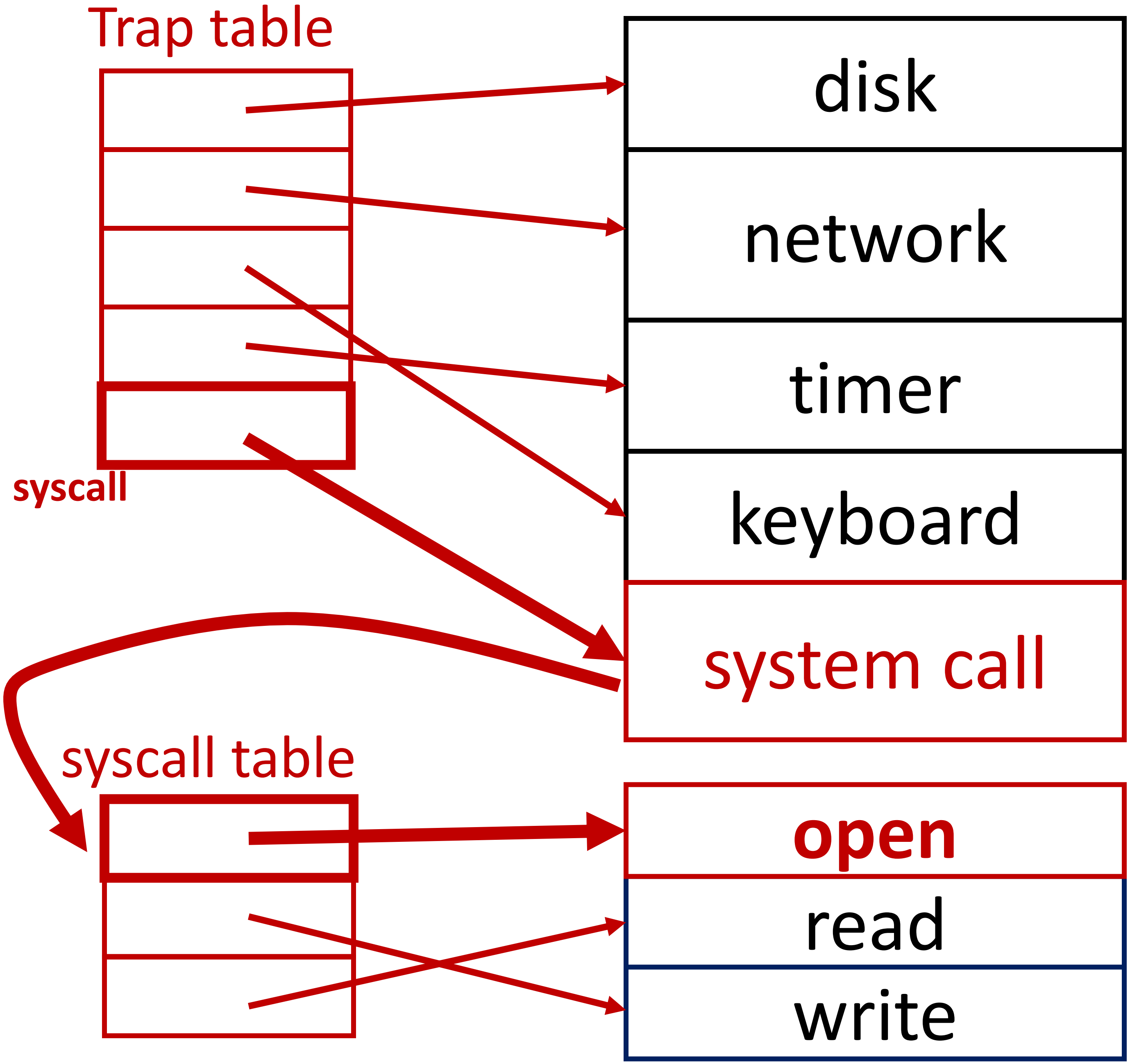
Use array of function pointers to locate OS functions
(Hardware knows this through **`lidt`** instruction)



How to handle variable number of system calls?


Trap table
disk
network
timer
keyboard
system call
syscall table
open
read
write


Trap table
disk
network
timer
keyboard
syscall
system call
syscall table
open
read
write


Trap table
disk
network
timer
keyboard
syscall
system call
syscall table
open
read
write

Safe Transfers

- Only certain kernel functions should be callable
- Privileges should escalate at the moment of the call
  - Read/write disk
  - Kill processes
  - Access all memory
  - …

LDE: Remaining Challenges

1. What if process wants to do something privileged?
2. How can OS switch processes (or do anything) if it's not running?

Sharing (virtualizing) the CPU

How does OS share…

- CPU?
- Memory?
- Disk?

How does OS share...

- CPU? (a: time sharing)
- Memory? (a: space sharing)
- Disk? (a: space sharing)

How does OS share…

- CPU? (a: time sharing)

Today

- Memory? (a: space sharing)
- Disk? (a: space sharing)

How does OS share…

- CPU? (a: time sharing)

Today

- Memory? (a: space sharing)
- Disk? (a: space sharing)

Goal: processes should **not** know they are sharing **(each process will get its own virtual CPU)**

What to do with processes that are not running?

- A: Store context in OS struct

What to do with processes that are not running?

- A: Store context in OS struct

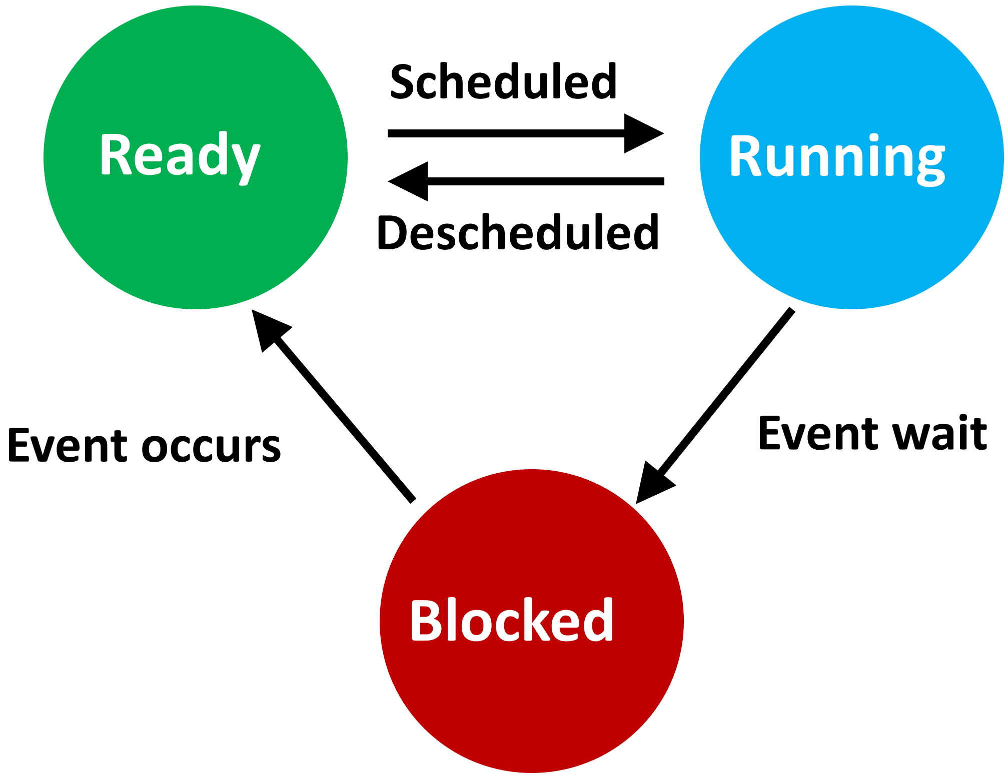
- Context:
    - CPU registers
    - Open file descriptors
    - State (sleeping, running, etc.)

What to do with processes that are not running?

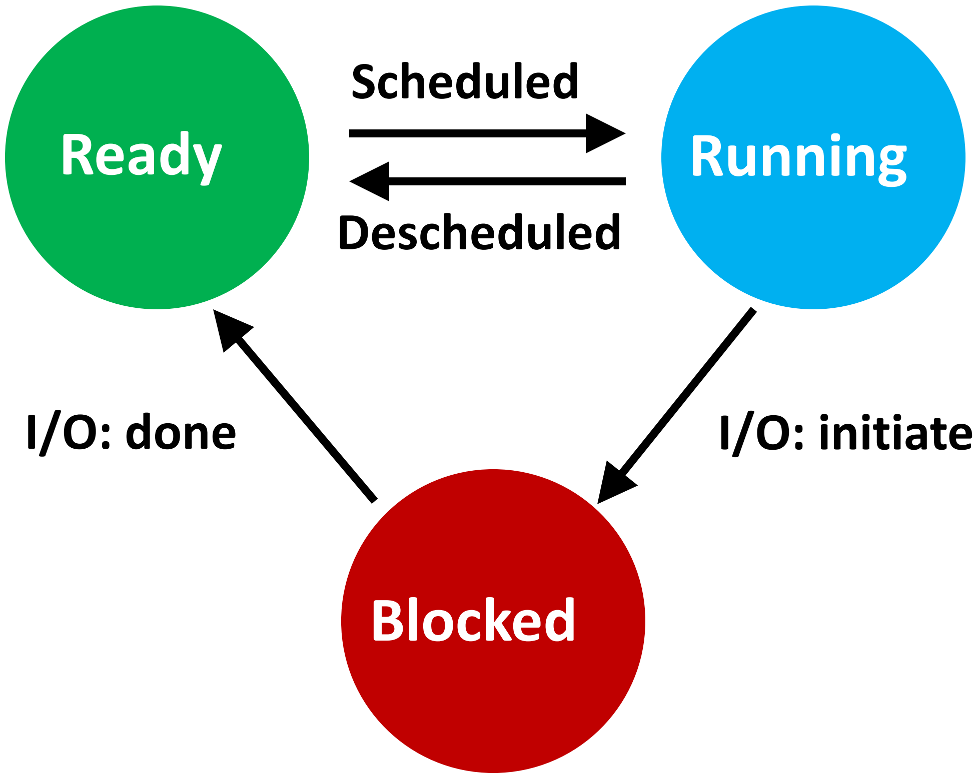
- A: Store context in OS struct

- Context:
  - CPU registers
  - Open file descriptors
  - State (sleeping, running, etc.)

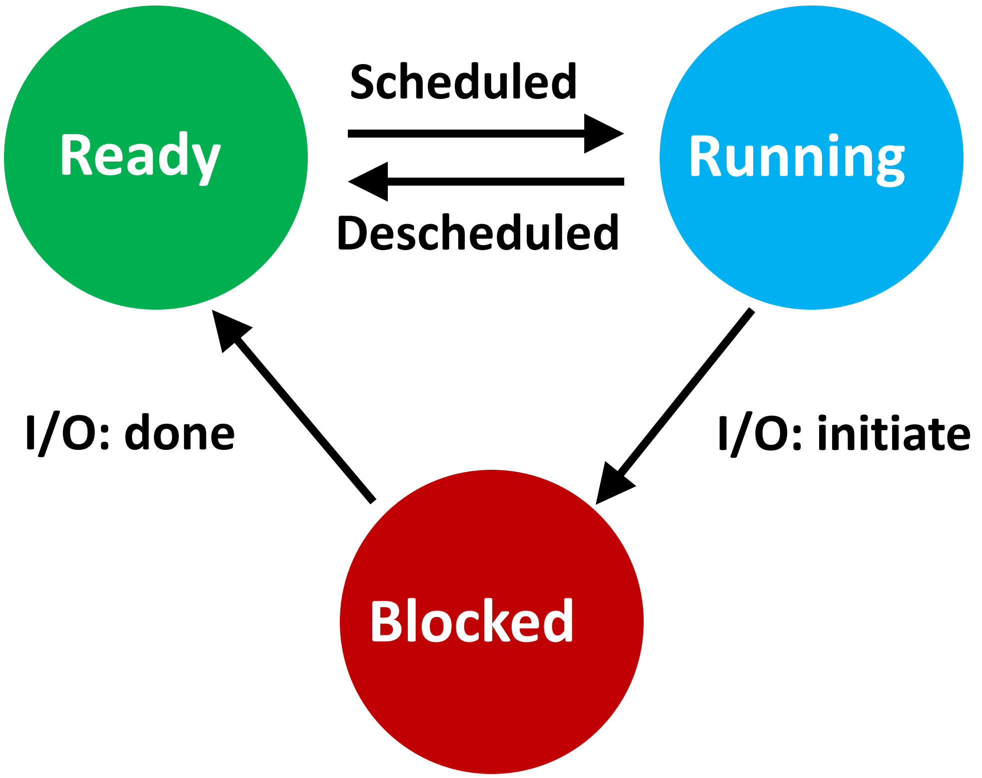
Process State Transitions



Process State Transitions

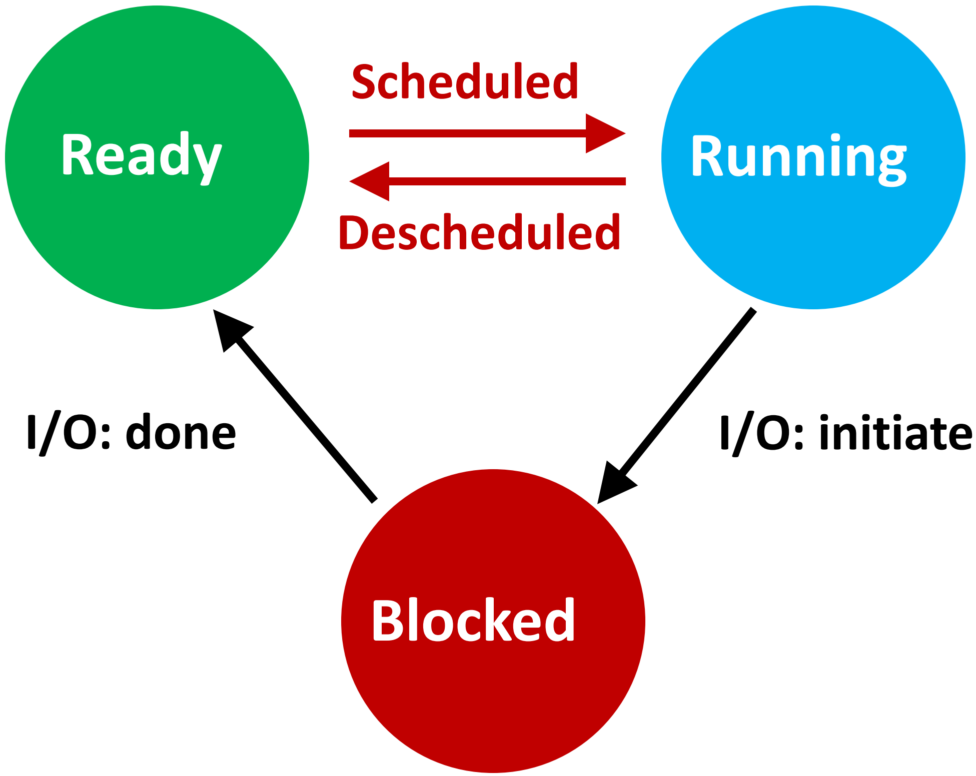


Process State Transitions



View process state with "`ps xa`"

How to transition? (mechanism)
When to transition? (policy)



Context Switch

- Problem: When to switch process contexts?
- Direct execution => OS can't run while process runs

- Can OS do anything while it's not running?

Context Switch

- Problem: When to switch process contexts?
- Direct execution => OS can't run while process runs

- Can OS do anything while it's not running?
- A: it can't

Context Switch

- Problem: When to switch process contexts?
- Direct execution => OS can't run while process runs

- Can OS do anything while it's not running?
- A: it can't

- Solution: Switch on interrupts
  - But what interrupt?

Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call

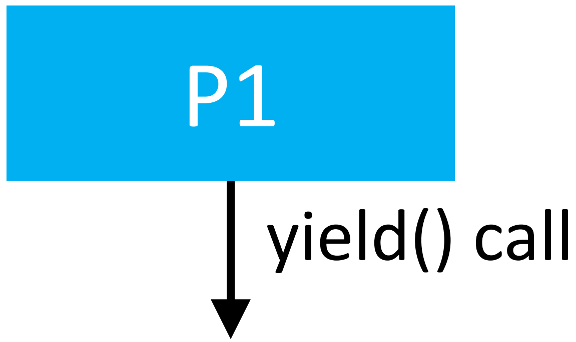
Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call

P1

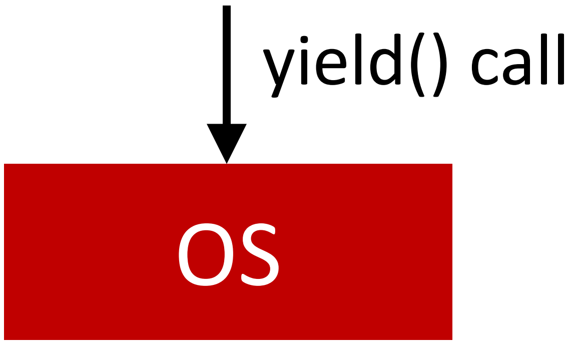
Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call



Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call

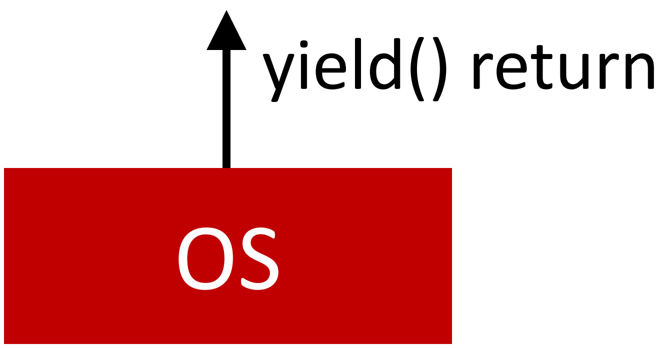


Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call

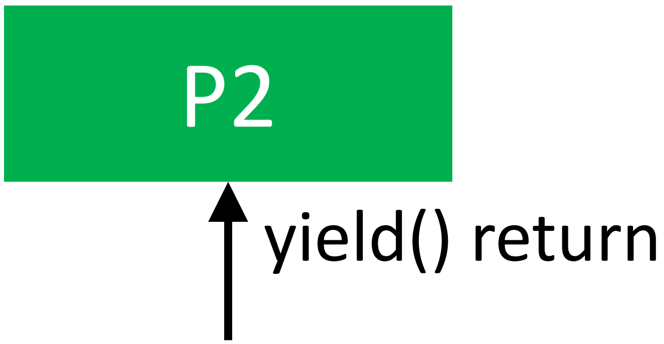
Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call



Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call



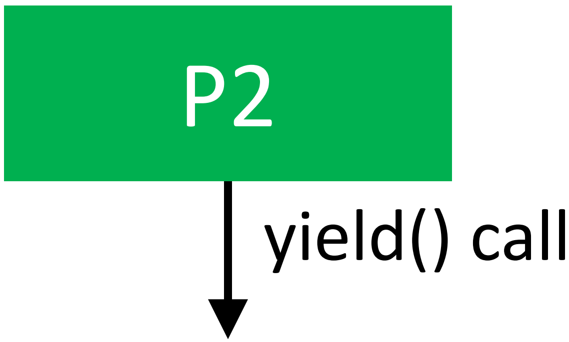
Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call

P2

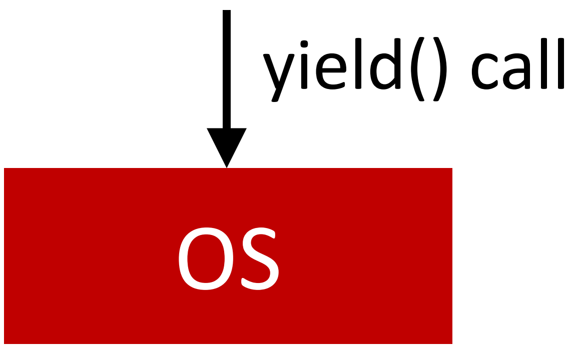
Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call



Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call



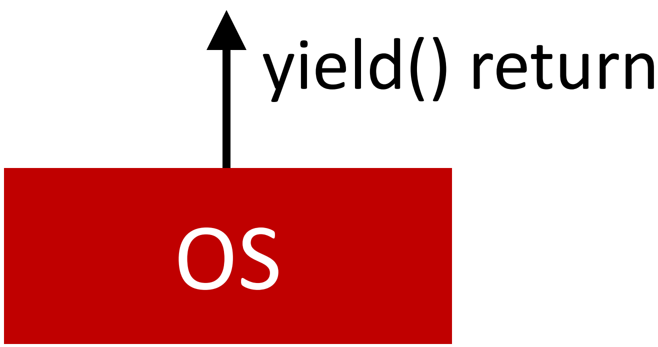
Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call

OS

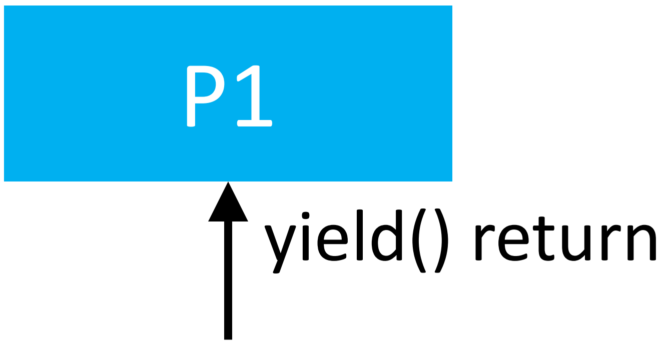
Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call



Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call



Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call

Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call

Critiques?

Cooperative Approach

- Switch contexts for syscall interrupt
  - Special `yield()` system call
- Cooperative approach is a passive approach

P1

Critiques?

What if P1 never calls `yield()`?

Non-Cooperative Approach

- Switch contexts on timer (hardware) interrupt
- Set up before running any processes
- Hardware does not let processes prevent this
  - Hardware/OS enforces process preemption

Non-Cooperative Approach

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A
		...

Non-Cooperative Approach

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A ...
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	

Non-Cooperative Approach

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A ...
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call `switch()` routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) **return-from-trap (into B)**		

Non-Cooperative Approach

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A …
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call `switch()` routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) **return-from-trap (into B)**		
	restore regs(B) from k-stack(B) move to user mode jump to B's PC	

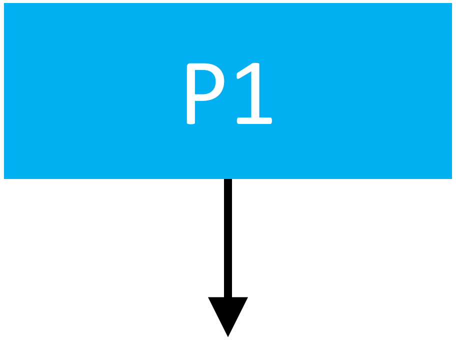
Non-Cooperative Approach

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A …
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call `switch()` routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) **return-from-trap (into B)**		
	restore regs(B) from k-stack(B) move to user mode jump to B's PC	
		Process B …

Preemptive Approach

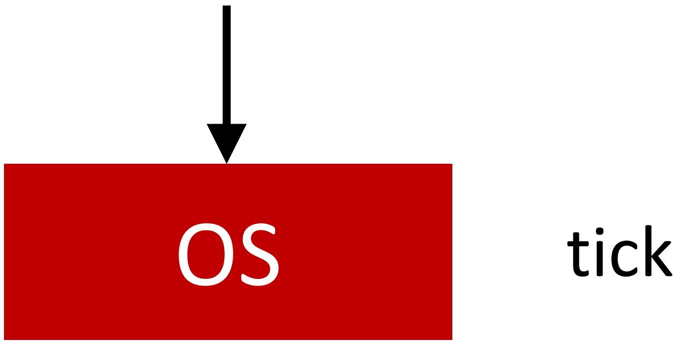
P1

Preemptive Approach



tick

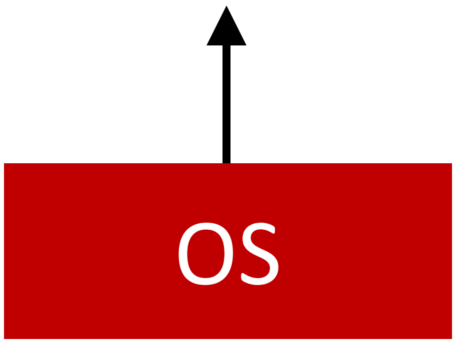
Preemptive Approach



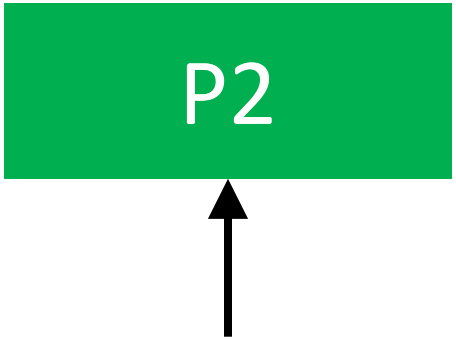
Preemptive Approach

OS

Preemptive Approach



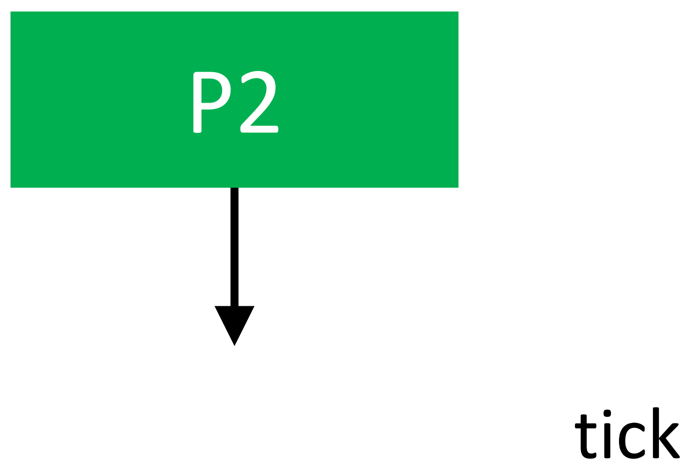
Preemptive Approach



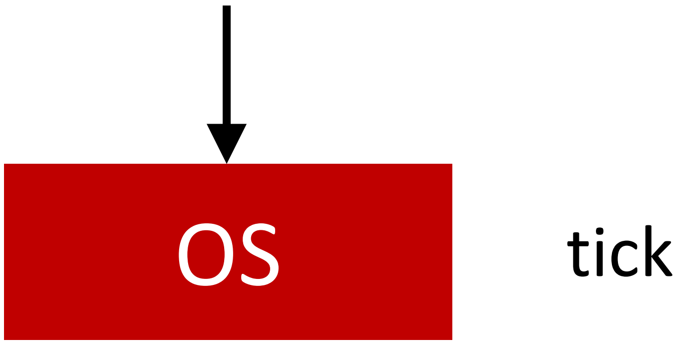
Preemptive Approach

P2

Preemptive Approach



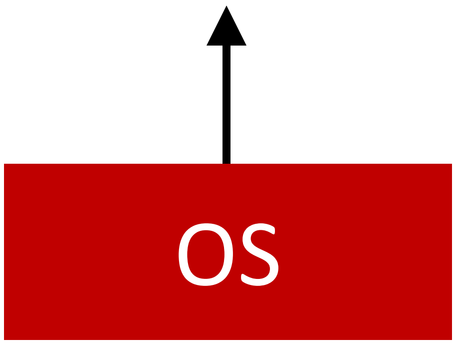
Preemptive Approach



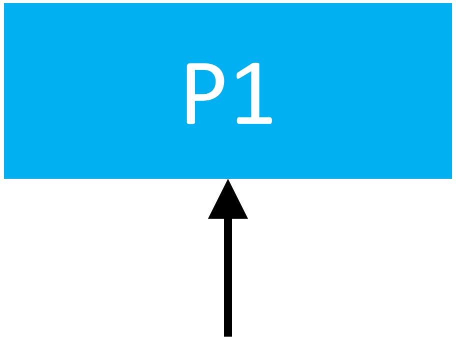
Preemptive Approach

OS

Preemptive Approach



Preemptive Approach



Preemptive Approach

P1

LDE Summary

- Smooth context switching makes each process think it has its own CPU (virtualization!)
- Limited direct execution makes processes fast
- Hardware provides a lot of OS support
  - Limited direct execution
  - Timer interrupt
  - Automatic register saving