# Operating Systems

(Sistemas Operativos)

## CPU Virtualization Mechanisms

University of Minho

2024-2025

# What will we learn?
## CPU Management

- Although a computer has a limited number of CPUs, users get the impression that several programs (a lot more than the available CPUs) are running simultaneously
  - How does the OS provide this illusion?
  - How can users use the OS APIs to start and stop programs?
  - How does the OS choose what programs will be running, be switched, …?

- Let us start with the **mechanisms**!

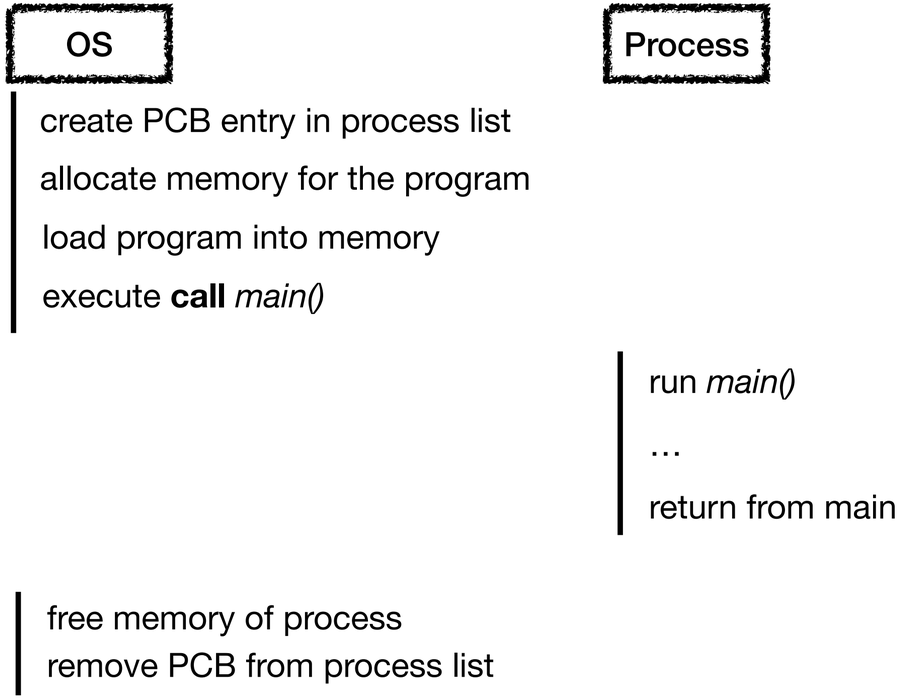
# CPU Virtualization
## Challenges

- To virtualize the CPU, the OS must share it among many programs running seemingly at the same time
  - Basic idea: **Time-sharing** the CPU
    i.e., run a process for a little while, then another one, …
- Challenges
  - **Performance:** OS mechanisms cannot add excessive overhead to the system
  - **Control:** The OS must retain control over the CPU, avoiding processes from
    - Taking control of the CPU (e.g., infinite loops)
    - Accessing/corrupting information from other programs in memory or at disk

# Limited Direct Execution

## Basic Technique

- **Goal:** Run a program as fast as possible in the CPU (Performance)
  - **Solution:** The program runs natively on the hardware CPU (*direct execution*)
- **Issues:** If the program has full control over the CPU, how does the OS…
  - Ensures the process does not perform any unwanted operations?
  - Stops the process from running and switches it with another one?
- One needs to *limit the execution* of programs *(i.e.,* limit what these can do*)*



# Modes of Execution

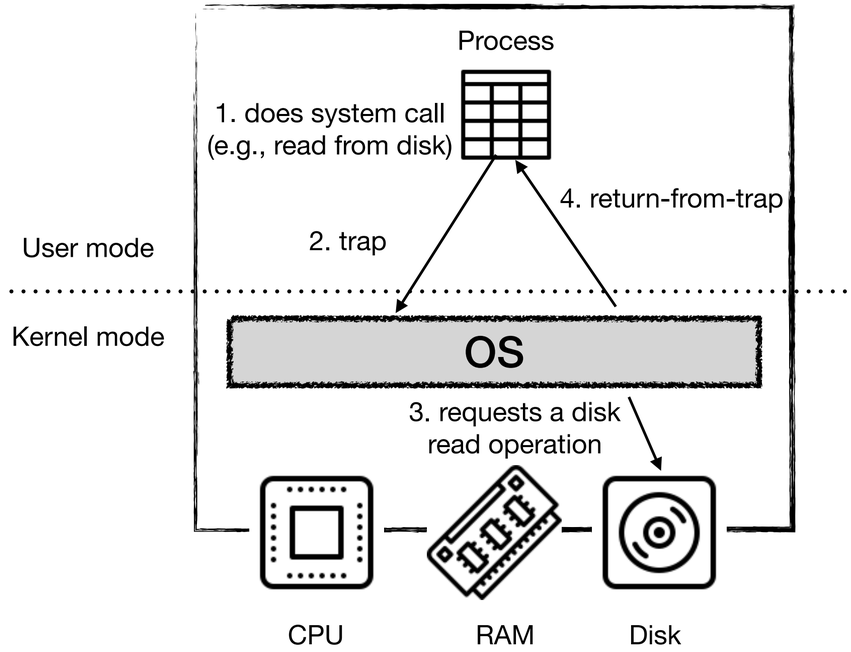
## Restricted Operations

- User processes (*i.e.*, running programs) cannot freely do restricted operations (*e.g.*, I/O requests to disk, gain access to resources such as CPU or memory)
- The OS must mediate this interaction (with the aid of hardware)!
  The CPU provides two modes of execution[1]
  - **User mode:** Code running in the CPU cannot do restricted (privileged) operations. (e.g., a process in user mode cannot directly do disk I/O operations)
  - **Kernel mode:** Code executing in the CPU has full access to the hardware and can do any restricted operation. **The OS runs in this mode!**
- **Issues:** What if a user process wishes to do a privileged operation?
  - Remember the system calls?

[1] *A single bit, stored at the processor, indicates which mode the CPU is currently running in*

# System Calls
## Restricted Operations

- **System calls** allow the kernel (OS) to expose privileged operations to user processes (e.g., process creation, communication, memory allocation, I/O)
- To execute a system call, the process calls a special **trap** instruction, raising the privilege level to **kernel mode** and jumping into the OS code (**trap handler**)
- The OS can now perform privileged operations (on behalf of the user process)
- When finished the OS calls a **return-from-trap** instruction that reduces the privilege level to **user mode** and resumes the execution of the process



# Limited Direct Execution (Updated)

OS (Kernel Mode) | Hardware | Process (User Mode)

# Limited Direct Execution (Updated)

| OS (Kernel Mode) | Hardware | Process (User Mode) |
| --- | --- | --- |
| create PCB entry / allocate and load memory<br>**return-from-trap** | | |

# Limited Direct Execution (Updated)

| OS (Kernel Mode) | Hardware | Process (User Mode) |
| --- | --- | --- |
| create PCB entry / allocate and load memory<br>**return-from-trap** | | |
| | move to user mode<br>jump to main | |

# Limited Direct Execution (Updated)

| OS (Kernel Mode) | Hardware | Process (User Mode) |
| --- | --- | --- |
| create PCB entry / allocate and load memory<br>**return-from-trap** | | |
| | move to user mode<br>jump to main | |
| | | run *main() …*<br>*call system call*<br>***trap** into OS* |

# Limited Direct Execution (Updated)

| OS (Kernel Mode) | Hardware | Process (User Mode) |
| --- | --- | --- |
| create PCB entry / allocate and load memory<br>**return-from-trap** | | |
| | move to user mode<br>jump to main | |
| | | run *main() …*<br>*call system call*<br>***trap** into OS* |
| | move to kernel mode<br>jump to trap handler | |

# Limited Direct Execution (Updated)

| OS (Kernel Mode) | Hardware | Process (User Mode) |
| --- | --- | --- |
| create PCB entry / allocate and load memory<br>**return-from-trap** | | |
| | move to user mode<br>jump to main | |
| | | run *main() …*<br>*call system call*<br>***trap** into OS* |
| | move to kernel mode<br>jump to trap handler | |
| handle trap<br>do work of system call<br>**return-from-trap** | | |

# Limited Direct Execution (Updated)

| OS (Kernel Mode) | Hardware | Process (User Mode) |
| --- | --- | --- |
| create PCB entry / allocate and load memory<br>**return-from-trap** | | |
| | move to user mode<br>jump to main | |
| | | run *main() …*<br>*call system call*<br>***trap** into OS* |
| | move to kernel mode<br>jump to trap handler | |
| handle trap<br>do work of system call<br>**return-from-trap** | | |
| | move to user mode<br>jump to PC after trap | |

# Limited Direct Execution (Updated)

| OS (Kernel Mode) | Hardware | Process (User Mode) |
|---|---|---|
| create PCB entry / allocate and load memory<br>**return-from-trap** | | |
| | move to user mode<br>jump to main | |
| | | run *main() …*<br>*call system call*<br>***trap*** *into OS* |
| | move to kernel mode<br>jump to trap handler | |
| handle trap<br>do work of system call<br>**return-from-trap** | | |
| | move to user mode<br>jump to PC after trap | |
| | | *…*<br>*return from main*<br>***trap*** *(via* exit*())* |

# Limited Direct Execution (Updated)

| OS (Kernel Mode) | Hardware | Process (User Mode) |
| --- | --- | --- |
| create PCB entry / allocate and load memory<br>**return-from-trap** | | |
| | move to user mode<br>jump to main | |
| | | run *main() …*<br>*call system call*<br>***trap** into OS* |
| | move to kernel mode<br>jump to trap handler | |
| handle trap<br>do work of system call<br>**return-from-trap** | | |
| | move to user mode<br>jump to PC after trap | |
| | | *…*<br>*return from main*<br>***trap** (via* exit*())* |
| free process memory / remove PCB | | |

# Mode Switching

## Traps and Interrupts

- **Two types of instructions** lead to transferring control from user into kernel mode
- A **trap** is generated by the current process running on a processor
  - The user program asks for an OS service (as we saw before)
    e.g., when the process issues a system call (in UNIX)
  - The user program performs an illegal action (an exception)
    e.g., division by zero, illegal memory access, …
- **Interrupt** are caused by devices and may not be related with the currently running process
  - When a device (disk, keyboard, network card) requires the OS attention
    e.g., a disk read is ready, there is keyboard input to be read

**Note:** *Terminology changes depending on the system (x86, xv6). Some call trap to all instructions, other use the terms interchangeably…*

# Interrupts

## Why are interrupts useful?

- **Example:** When a process issues a read for a file (system call), the OS will ask the disk device for that specific content
  - The disk will require some time to bring that information to memory…
- The OS can periodically check (**polling**) the disk device for that request completion
  - If events take long time to complete, then the OS is waisting precious CPU cycles that could be used to run other processes!
- **Interrupts** are generated by the devices asynchronously
  - **Benefit:** The OS is notified when events occur, leaving the CPU free for other processes
  - **Caution:** Interrupts are issued concurrently, when other processes or the OS are running at the CPU

# Limited Direct Execution (Interrupts)

| OS (Kernel Mode) | Hardware | Process (User Mode) |
| --- | --- | --- |
| create PCB entry / allocate and load memory<br>**return-from-trap** | | |
| | move to user mode<br>jump to main | |
| | | run *main()*<br>*…* |

# Limited Direct Execution (Interrupts)

| OS (Kernel Mode) | Hardware | Process (User Mode) |
| --- | --- | --- |
| create PCB entry / allocate and load memory<br>**return-from-trap** | | |
| | move to user mode<br>jump to main | |
| | | run *main()*<br>*…* |
| | **interrupt event** (e.g., from disk)<br>move to kernel mode<br>jump to trap handler | |

# Limited Direct Execution (Interrupts)

| OS (Kernel Mode) | Hardware | Process (User Mode) |
|---|---|---|
| create PCB entry / allocate and load memory<br>**return-from-trap** | | |
| | move to user mode<br>jump to main | |
| | | run *main()*<br>*…* |
| | **interrupt event** (e.g., from disk)<br>move to kernel mode<br>jump to trap handler | |
| handle interruption<br>do work<br>**return-from-interrupt** | | |
| | move to user mode<br>jump to PC after interruption | |
| | | *…*<br>*return from main*<br>***trap*** *(via* exit*())* |
| free process memory / remove PCB | | |

# Traps and Interrupts

## Process state

- Both **trap** and **interrupt** instructions are **hardware-assisted**
  - The OS cannot take the CPU from a process without aid
  - The CPU must interrupt the process, change to kernel mode, save some information and jump to the OS code
- When a process is interrupted, the CPU must save a set of registers
  - Registers contain information (e.g., program counter) required by the CPU to resume the process execution when the OS calls a **return-from-trap/interrupt** instruction
  - Registers are stored, for example, in a per-process kernel stack

*On x86, for example, the processor will push the **program counter, flags, and a few other registers** onto its **kernel stack**; the return-from-trap will pop these values off the stack and resume execution of the user-mode program*

# Limited Direct Execution (Registers)

OS (Kernel Mode)

Hardware

Process (User Mode)

# Limited Direct Execution (Registers)

OS (Kernel Mode) | Hardware | Process (User Mode)

create PCB / allocate and load memory
**fill kernel stack with registers / PC**
**return-from-trap**

# Limited Direct Execution (Registers)

| OS (Kernel Mode) | Hardware | Process (User Mode) |
| --- | --- | --- |
| create PCB / allocate and load memory<br>**fill kernel stack with registers / PC**<br>**return-from-trap** | | |
| | **restore regs (from kernel stack)**<br>move to user mode<br>jump to main | |

# Limited Direct Execution (Registers)

| OS (Kernel Mode) | Hardware | Process (User Mode) |
| --- | --- | --- |
| create PCB / allocate and load memory<br>**fill kernel stack with registers / PC**<br>**return-from-trap** | | |
| | **restore regs (from kernel stack)**<br>move to user mode<br>jump to main | |
| | | run *main() …*<br>*call system call / **trap** into OS* |

# Limited Direct Execution (Registers)

| OS (Kernel Mode) | Hardware | Process (User Mode) |
| --- | --- | --- |
| create PCB / allocate and load memory<br>**fill kernel stack with registers / PC**<br>**return-from-trap** | | |
| | **restore regs (from kernel stack)**<br>move to user mode<br>jump to main | |
| | | run *main() …*<br>*call system call / **trap** into OS* |
| | **save regs (to kernel stack)**<br>move to kernel mode<br>jump to trap handler | |

# Limited Direct Execution (Registers)

| OS (Kernel Mode) | Hardware | Process (User Mode) |
| --- | --- | --- |
| create PCB / allocate and load memory<br>**fill kernel stack with registers / PC**<br>**return-from-trap** | | |
| | **restore regs (from kernel stack)**<br>move to user mode<br>jump to main | |
| | | run *main() …*<br>*call system call / **trap** into OS* |
| | **save regs (to kernel stack)**<br>move to kernel mode<br>jump to trap handler | |
| handle trap<br>do work of system call<br>**return-from-trap** | | |

# Limited Direct Execution (Registers)

| OS (Kernel Mode) | Hardware | Process (User Mode) |
|---|---|---|
| create PCB / allocate and load memory<br>**fill kernel stack with registers / PC**<br>**return-from-trap** | | |
| | **restore regs (from kernel stack)**<br>move to user mode<br>jump to main | |
| | | run *main() …*<br>*call system call / **trap** into OS* |
| | **save regs (to kernel stack)**<br>move to kernel mode<br>jump to trap handler | |
| handle trap<br>do work of system call<br>**return-from-trap** | | |
| | **restore regs (from kernel stack)**<br>move to user mode<br>jump to PC after trap | |

# Limited Direct Execution (Registers)

| OS (Kernel Mode) | Hardware | Process (User Mode) |
|---|---|---|
| create PCB / allocate and load memory<br>**fill kernel stack with registers / PC**<br>**return-from-trap** | | |
| | **restore regs (from kernel stack)**<br>move to user mode<br>jump to main | |
| | | run *main() …*<br>*call system call / **trap** into OS* |
| | **save regs (to kernel stack)**<br>move to kernel mode<br>jump to trap handler | |
| handle trap<br>do work of system call<br>**return-from-trap** | | |
| | **restore regs (from kernel stack)**<br>move to user mode<br>jump to PC after trap | |
| | | *…*<br>*return from main / **trap** (via* exit()*)* |

# Limited Direct Execution (Registers)

| OS (Kernel Mode) | Hardware | Process (User Mode) |
|---|---|---|
| create PCB / allocate and load memory<br>**fill kernel stack with registers / PC**<br>**return-from-trap** | | |
| | **restore regs (from kernel stack)**<br>move to user mode<br>jump to main | |
| | | run *main() …*<br>*call system call / **trap** into OS* |
| | **save regs (to kernel stack)**<br>move to kernel mode<br>jump to trap handler | |
| handle trap<br>do work of system call<br>**return-from-trap** | | |
| | **restore regs (from kernel stack)**<br>move to user mode<br>jump to PC after trap | |
| | | *…*<br>*return from main / **trap** (via* exit()*)* |
| Free process memory / remove PCB | | |

# Limited Direct Execution (Registers)

| OS (Kernel Mode) | Hardware | Process (User Mode) |
|---|---|---|
| create PCB / allocate and load memory<br>**fill kernel stack with registers / PC**<br>**return-from-trap** | | |
| | **restore regs (from kernel stack)**<br>move to user mode<br>jump to main | |
| | | run *main() …*<br>*call system call / **trap** into OS* |
| | **save regs (to kernel stack)**<br>move to kernel mode<br>jump to trap handler | |
| handle trap<br>do work of system call<br>**return-from-trap** | | |

How does the hardware knows what OS code it should run (it should jump into)?

# Traps and Interrupts

## Interrupt/Trap Table

- At boot time the kernel (OS) sets up an **interrupt/trap table**
- The table tells the hardware what code to run (**interrupt/trap handler**) under exceptional events
  - what code should run when an hard disk interrupt happens,
  - or when a keyboard interrupt occurs,
  - or when a program makes a system call
- The table maps interrupts and traps (e.g., identified by a number) to their corresponding handlers
  - It distinguishes interrupts from different **devices** (e.g., disk, keyboard) and **requests** (e.g, disk read, write)
  - It distinguishes between different **trap types** (e.g., exception, system call)

| | |
|---|---|
| 0 | HANDLER SYSTEM CALL |
| 1 | HANDLER KEYBOARD… |
| 2 | HANDLER DISK… |
| … | … |

Illustrative example
(not a real one!)

# System Calls in Practice

## Integration with the C library

- When a program calls a system call (*e.g.*, *fork()*, *open()*), a procedure call into the C library is done
- The library uses an agreed-upon calling convention with the OS kernel
  - **System call arguments** and **number** (identifier) are placed into well-known locations (e.g., stack, special registers)
- The library executes the corresponding trap instruction (system call)
  - The **interrupt/trap table** is used to check for the **system call trap handler**
  - When running the trap handler code, the OS can now validate the system call **number** and run the corresponding code
- This indirection provides **security**
  (safer than jumping directly to a memory address to run the code)

# Switching Processes

## The Conundrum

- Let us go back to our initial goal
  - **Time-sharing** the CPU, i.e., run a process for a little while, then another one…
  - This requires pausing one process and running another one
  - But we saw that if a process is running at the CPU, the OS is not running
- **Question:** Given the mechanisms studied before, do you have any ideas on how the OS can share the CPU across several processes?
- We are getting closer to revealing the illusion created by the OS!

# Cooperative Approach

## Wait for System Calls

- When a process does a system call, the OS kernel regains control to
  - Handle the system call (e.g., ask the disk to read some content)
  - Do other tasks such as scheduling another process!
- Actually the OS can also regain control under device interruptions and other traps (exceptions)
- In a cooperative approach the OS assumes the processes will
  - Issue system calls (e.g., block for doing disk I/O)
  - or voluntarily yield the CPU to the OS (the *yield* system call serves exactly this purpose)
- **Question:** What if we have malicious or buggy programs (e.g., infinite loops)…? Any ideas on how to solve it?

# Non-Cooperative Approach

## Timer Interrupt

- We need help from the hardware once again, namely from a **timer interrupt**
  - *E.g., e*very X milliseconds an interrupt is generated by the hardware
  - The process is halted, a specific interrupt handler runs and the OS gains control!
- The OS is responsible for
  - Defining the code for the timer interrupt handler
  - Setting the timer's interval (at boot time)
- **Remember:** when the interrupt occurs process state must be saved so the process can be resumed later…

# Saving and Restoring Context

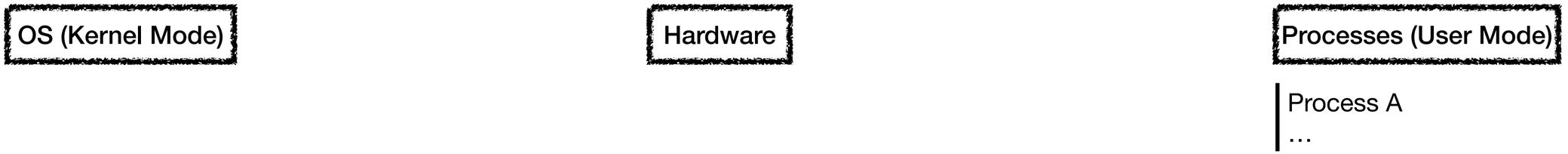
## Context Switching

- When the OS regains control (cooperatively or non-cooperatively)
  - it must decide to keep running the same process or to run another one
  - This decision is made by the OS **scheduler**
- When the scheduler decides to switch $process_A$ with $process_B$, a **context switch** is made
  - The OS must save information of $process_A$ (e.g., program counter, memory limits, kernel stack pointer[1]) into its $PCB_A$ structure (**Process Control Block** - check the *Processes slides*!)
    - This information is part of the **process list** and it is managed by the OS!
      (Different from the per-process kernel stack managed by the hardware)
  - Switches contexts, by changing the CPU's stack pointer (a register) to the kernel stack of $process_B$

[1] Remember that the CPU stores essential information to resume a process in a per-process kernel stack. The PCB stores a pointer to the stack's address.

# Limited Direct Execution (Context Switching)

# Limited Direct Execution (Context Switching)



# Limited Direct Execution (Context Switching)

| OS (Kernel Mode) | Hardware | Processes (User Mode) |
| --- | --- | --- |
| | | Process A<br>… |
| | **timer interrupt**<br>**save regs (to kernel stack$_A$)**<br>move to kernel mode<br>jump to interrupt handler | |

# Limited Direct Execution (Context Switching)

| OS (Kernel Mode) | Hardware | Processes (User Mode) |
|---|---|---|
| | | Process A<br>… |
| | **timer interrupt**<br>**save regs (to kernel stack$_A$)**<br>move to kernel mode<br>jump to interrupt handler | |
| handle interrupt<br>**call *switch()* routine**<br>save PCB$_A$<br>restore regs from PCB$_B$<br>switch to kernel stack$_B$ | | |

# Limited Direct Execution (Context Switching)

| OS (Kernel Mode) | Hardware | Processes (User Mode) |
|---|---|---|
| | | Process A<br>… |
| | **timer interrupt**<br>**save regs (to kernel stack$_A$)**<br>move to kernel mode<br>jump to interrupt handler | |
| handle interrupt<br>**call *switch()* routine**<br>save PCB$_A$<br>restore regs from PCB$_B$<br>switch to kernel stack$_B$ | | |
| | **restore regs (from kernel stack$_B$)**<br>move to user mode<br>jump to Process$_B$ PC | |

# Limited Direct Execution (Context Switching)

| OS (Kernel Mode) | Hardware | Processes (User Mode) |
|---|---|---|
| | | Process A<br>… |
| | **timer interrupt**<br>**save regs (to kernel stack$_A$)**<br>move to kernel mode<br>jump to interrupt handler | |
| handle interrupt<br>**call *switch()* routine**<br>save PCB$_A$<br>restore regs from PCB$_B$<br>switch to kernel stack$_B$ | | |
| | **restore regs (from kernel stack$_B$)**<br>move to user mode<br>jump to Process$_B$ PC | |
| | | *Process B*<br>*…* |

# Performance
## Context Switching

- In 1996, when running Linux 1.3.37 on a 200-MHz P6 CPU, **system calls took roughly 4 microseconds**, and a **context switch roughly 6 microseconds**
- Modern systems perform almost an order of magnitude better, with **sub-microsecond** results on systems with 2- or 3-GHz processors
- Whenever a system call to disk is made, for example, the OS will probably schedule another process to run
  - **Question:** Why?
  - **Question:** Do you see an impact of implementing a program that does repetitive system calls that take few microseconds?

# More Information

- **Chapters 6** - Remzi H. Arpaci-Dusseau, Andrea C. Arpaci-Dusseau. **Operating Systems: Three Easy Pieces.** Arpaci-Dusseau Books, 2018.
- Avi Silberschatz, Peter Baer Galvin, Greg Gagne. **Operating System Concepts (10. ed)**. John Wiley & Sons, 2018.

# Questions?