Virtualizing the CPU

2023-24 COMP3230A

Contents

- System Calls
- Process switch
- Interrupt

Related Learning Outcome

- ILO 2a - explain how OS manages processes/threads and **discuss the mechanisms** and policies in efficiently sharing of CPU resources.

Reading & Reference

- Required Reading
  - Chapter 6, Mechanism: Limited Direct Execution, Operating Systems: Three Easy Pieces by Arpaci-Dusseau et. Al
    - http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-mechanisms.pdf

- Reference
  - Chapter 3 of Operating Systems, 3rd edition by Deitel et. al

Process Control

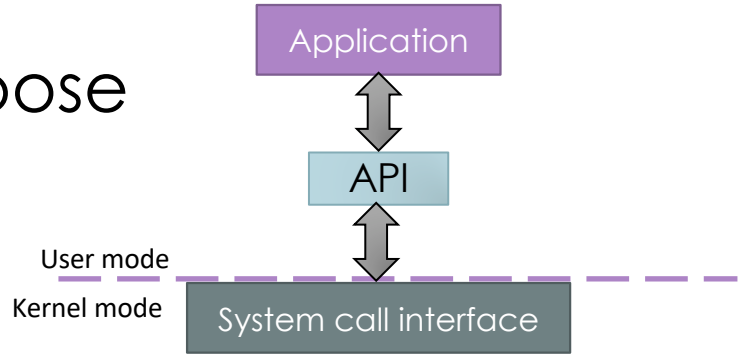
- How to perform restricted operations?
  - Mechanism: System Calls
- How to provide illusion of having many CPUs?
  - Virtualizing the CPU
  - Mechanism: Context Switching
- How to regain control of the CPU?
  - Voluntary release of CPU
  - Mechanism: System Calls
  - Involuntary release of CPU
  - Mechanism: Interrupt

Transparent:

- Process does not know when it is running and when it is not
- Programmer does not need to worry about this situation

To request OS services

- **System calls** allow the kernel to carefully expose certain key services to applications
  - Restricted operations in user mode
  - Most OSs expose a few hundreds such functions

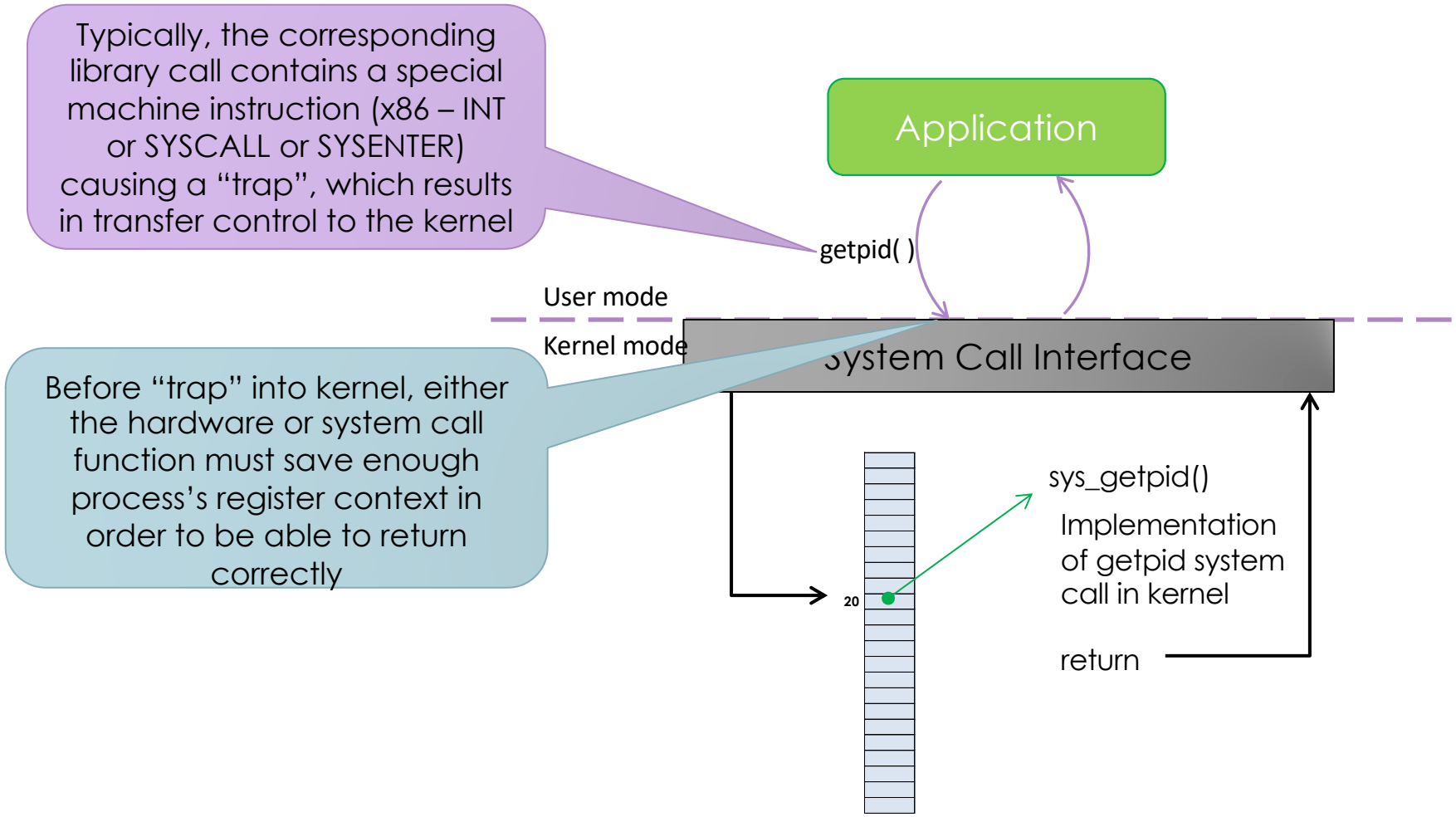


- Mostly accessed by programs via a high-level **Application Program Interface** (API) rather than directly invoke the specific system call
  - Most common APIs are Win32 API for Windows, and POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X)
  - Usually, in Unix system, the API is included in the run-time support library (e.g. C library)

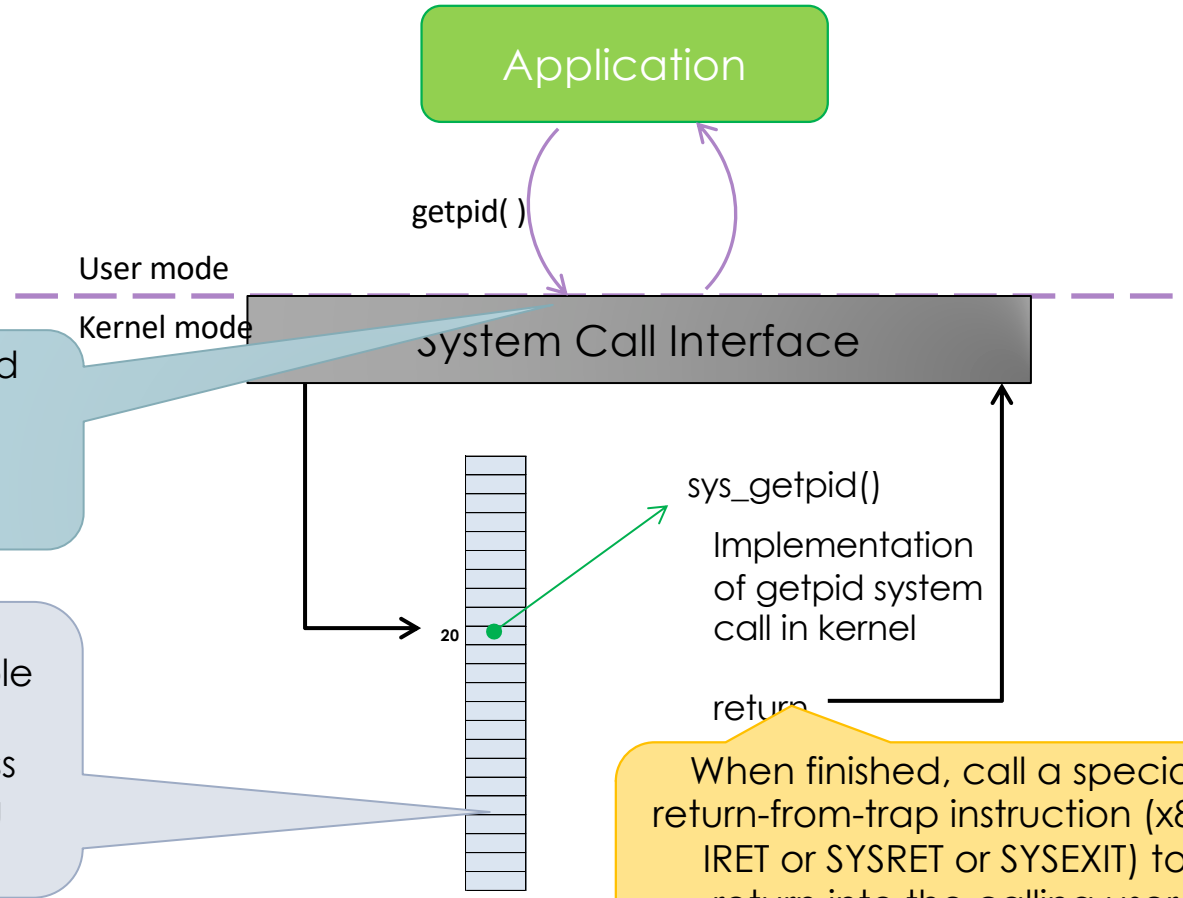
To request OS services

- Why use APIs rather than directly invoke system calls?
  - Caller does not need to know how the system call is implemented and how to invoke a specific system call in that particular OS
  - Just needs to know how to use the **common or standard** API and understand what OS will do / offer
    - Most details of OS interface hidden from programmer by API
    - The system call interface invokes intended system call in kernel and returns status and results of the system call and pass back to calling program via API

System Call Implementation



System Call Implementation



- Each system call is associated with a number.
- This number is passed by the function call to the kernel.

- Within kernel, the system-call implementation maintains a table indexed by these numbers.
- Each entry consists of an address that points to the corresponding system function in the kernel

When finished, call a special return-from-trap instruction (x86 - IRET or SYSRET or SYSEXIT) to return into the calling user program

System Call Protocol

OS @ run (kernel mode)	Hardware	Program (user mode)
Initialize trap table		
	Remember address of syscall handler	

OS @ run (kernel mode)	Hardware	Program (user mode)
Create entry for process list; Allocate memory for program; Load program into memory; Setup user stack with argv; Fill kernel stack with reg/PC **return-from-trap**		
	Restores regs (from kernel stack) Move to user mode Jump to main	
		Run main()… Call system call **trap** into OS
	Save regs (to kernel stack) Move to kernel mode Jump to trap handler	
Handle trap Do work of system call **return-from-trap**		
	Restore regs (from kernel stack) Move to user mode Jump to PC **after** trap	
		… Return from main **trap** (via exit())
Free memory of process Remove from process list		

Virtualizing the CPU

- To provide illusion that each process has its own CPU
- By virtualizing, does not mean giving a virtual CPU
  - The program is directly running on the real CPU
  - Mechanism:
    - **Time sharing** the CPU: By running one process, then stopping it and running another, and so forth
- The Crux
  - How to **transparently** and **temporarily stop** a process and **resume** it?
  - With direct execution of application process on CPU, how can OS **regain control** of the CPU?

Switching from one process to another process

- To stop a process, one just needs to remove it from running in CPU (to ready state)
- How can OS make sure that the process can be resumed **without affecting** its execution logic?
  - OS needs to **save the register context** (the contents of CPU registers) of current running process before stopping it
  - To resume, OS needs to **restore the register context** of the soon-to-be running process back to CPU registers before resuming it
  - Question: Where to store the register context?
- Context Switch
  - Switching of the register context of one process to another

Context Switch

- Action
  - Save the "current" process's register context to the process's kernel stack
    - *save the general purpose registers, PC, as well as the kernel stack pointer of the currently-running process, and then restore said registers, PC, and switch to the kernel stack for the soon-to-be-executing process.*
  - Other Actions
    - Change the process's state in PCB to appropriate status
    - Move the process to appropriate queue
    - Select a ready process (according to **scheduling policy**)
    - Load the "to be dispatched" ready process's register context from its PCB
    - Update the process's state of that going to be run process
    - Resume that process by turning control over to that process

Context Switch Overhead

- The CPU is considered as not performing any "useful" computation for the application process
  - OS must minimize context-switching time
  - There are hardware instructions in some architectures that facilitates the switching
- Has some indirect cost (induced overheads)
  - The newly scheduled process may not have data and/or instructions in the physical memory
  - The newly scheduled process does not have any part of its address space in cache

Mode Switch and Context Switch

- Mode switch
  - Switching from user mode to kernel mode
  - the Kernel is said to be "executing on behalf of the process"
  - the process's context remains accessible (e.g., address space)
  - upon exiting kernel, the process *may* resume and return to execute in user space
- Context switch
  - Switching from one process to another
  - is an essential feature of multiprogramming or time-sharing systems
  - Kernel must
    - (1) suspend the progression of current process and stores its context
    - (2) retrieve the context of another process (B) and restores it to the CPU
    - (3) resume execution of process B

To regain control of CPU

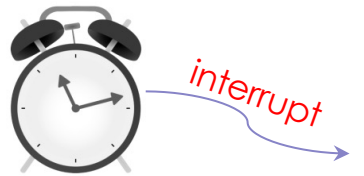
- A Passive Approach
  - OS just waits for a process to make **system calls** or an illegal operation
  - Indeed, this works most of the time as processes quite frequently invoke system calls
  - However, in worst case, a process gets into an infinite loop and never makes a system call, then what can OS do!!
- An Active Approach (assume non-cooperative processes)
  - Needs hardware support: **Timer Interrupt**
  - A timer device periodically generates an interrupt, e.g. every few milliseconds
  - When the interrupt is raised, OS will be called in to handle the interrupt, and thus can do what it wants

Interrupts

- Interrupt-request line triggered by hardware device
  - Interrupts enable OS to respond to "alerts" from **hardware**
- May be triggered by a running process
  - Interrupt in this case is called an exception
  - **Synchronous** with the operation of the **current** process
  - Unexpected event from within the processor
  - e.g., dividing by zero or referencing protected memory
- May be initiated by some event that may or may not be related to the running process
  - **Asynchronous** with the operation of the **current** process
  - Unexpected event from outside the processor
  - e.g., a key is pressed or the mouse is moved, timer events
- Software-generated interrupt
  - the INT instruction

Interrupt Handling

The processor responds to interrupts and exceptions in essentially the same way.



Timer device

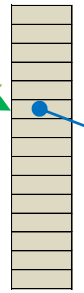
interrupt

CPU control

P1

After receiving an interrupt, the processor completes execution of the current instruction

Interrupt vector



- OS has to save remainder of process state in somewhere
- Jump to interrupt handler and process the interrupt

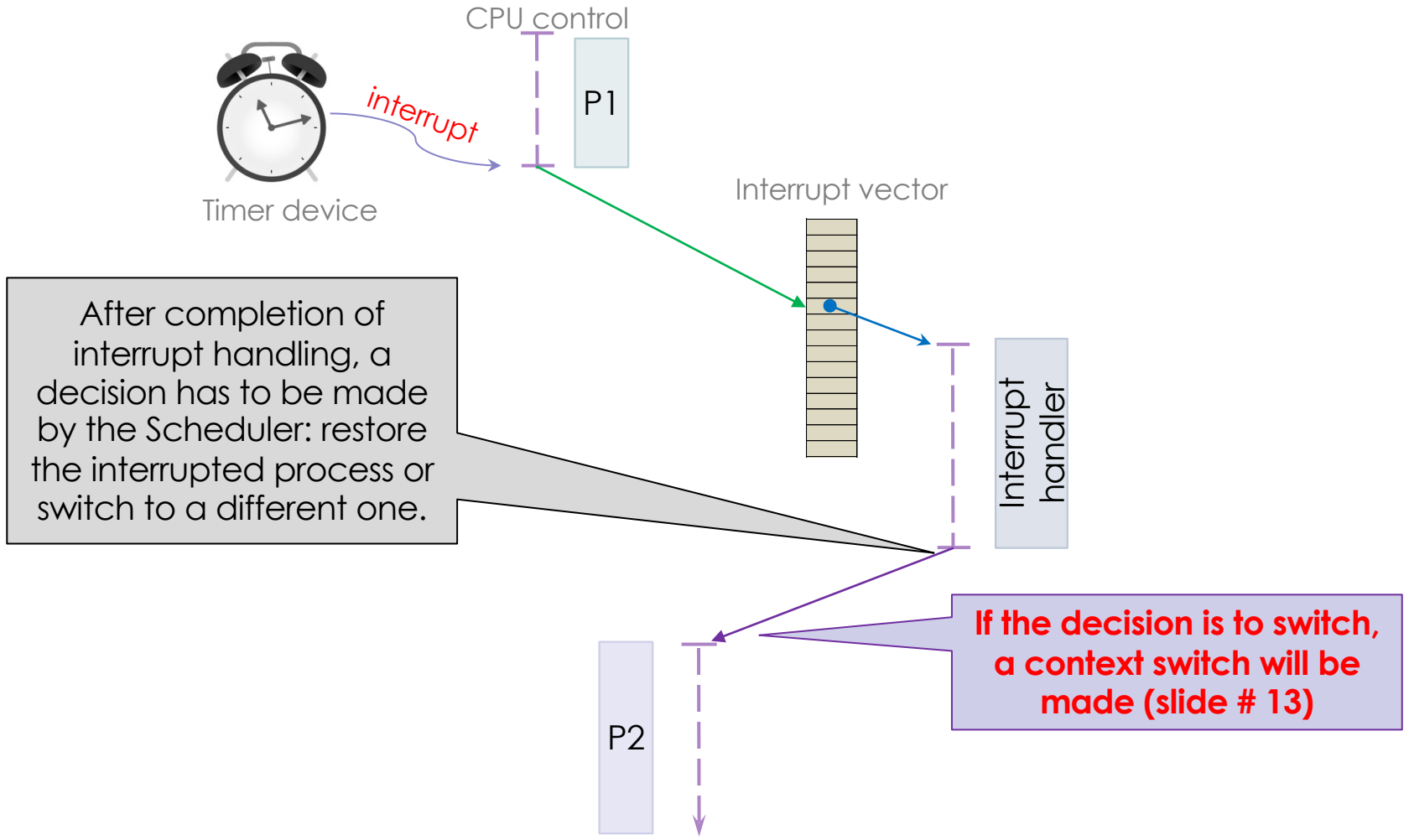
Interrupt handler

CPU has to save some register contents when interrupt occurred. It could be the whole set or just a few registers.

- Interrupt handlers are stored in an array of function pointers called the interrupt vector
- OS sets up the vector at boot time
- CPU reads from the system bus the interrupt vector number provided by an interrupt controller
- This number is used as an index to locate the interrupt handler

The interrupt handler determines how the system should respond

Interrupt Handling



Timer Interrupt

OS @ run (kernel mode)	Hardware	Program (user mode)
Initialize trap table		
	Remember address of trap handler	
Start interrupt timer		
	Start timer Interrupt CPU every x ms	
OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A…
	timer interrupt Save regs(A) to kernel stack(A) Move to kernel mode Jump to trap handler	
Handle trap Call switch() routine to switch from A to B save regs(A) to PCB(A) restore regs(B) from PCB(B) switch to kernel stack(B) **return-from-trap (into B)**		
	Restore regs(B) from kernel stack(B) Move to kernel mode Jump to B's PC	
		Process B…

Summary

- Three mechanisms
  - System call
    - Each call represent a service provided by the OS to user application
    - System mode will switch from user mode to kernel mode, and after the service, will switch back from kernel mode to user mode
  - Context switch
    - The key action of virtualizing the CPU
    - The whole set of register context must be saved for the interrupted process
    - The register context of coming process are restored
    - Context switch involves certain amount of performance overhead
  - Interrupt processing
    - A mechanism for hardware devices to alert OS for handling high-priority events
    - This gives a chance to OS to regain control of CPUs

Operating Systems

- **Virtualization**
  - CPU Virtualization
    - Process Abstract
      - Address space
      - Process states
      - Process control block
      - Process operations API
      - Signals
    - Limited Direct Execution
      - System calls
      - Context switch
      - Interrupts
    - Scheduling
      - Scheduling metrics
      - FIFO, SJF, HRRN, STCF, RR, MLFQ
      - Multi-core scheduling, Linux CFS
  - Memory Virtualization
    - Address space
    - Address translation: dynamic relocation
    - Segmentation
    - Paging
    - TLB
    - Multi-level paging
    - Inverted page table
    - Swap space
    - Page replacement policy: FIFO, LFR, LRU, Clock
    - Thrashing
- **Concurrency**
  - Thread
    - POSIX threads (pthreads)
    - Race conditions, critical sections, mutual exclusion, atomic operations, synchronization
  - Locks
    - Atomic instructions: test-and-set, compare-and-swap
    - Mutex locks
  - Condition Variables
    - Pthread CVs
    - Producer-Consumer problem
  - Semaphores
    - Binary Semaphores
    - Counting Semaphores
    - Ordering
    - Readers-Writers problem
  - Deadlock
    - Dining philosophers' problem
    - Four necessary conditions
    - Deadlock prevention, avoidance, detection&recovery
- **Persistence**
  - I/O devices (HDD, SSD)
  - Files and Directories
    - Inode
    - File descriptor
    - Hard/Symbolic links
  - File System Implementation
    - On-disk data structure
      - Superblock, Bitmap, Inodes, Data blocks
    - Free space management
      - Bitmap, linked-list, block-list
    - Caching and buffering
    - Access control and protection
    - Journaling file system
      - Data journaling
      - Metadata journaling
- ***Advanced Topics***