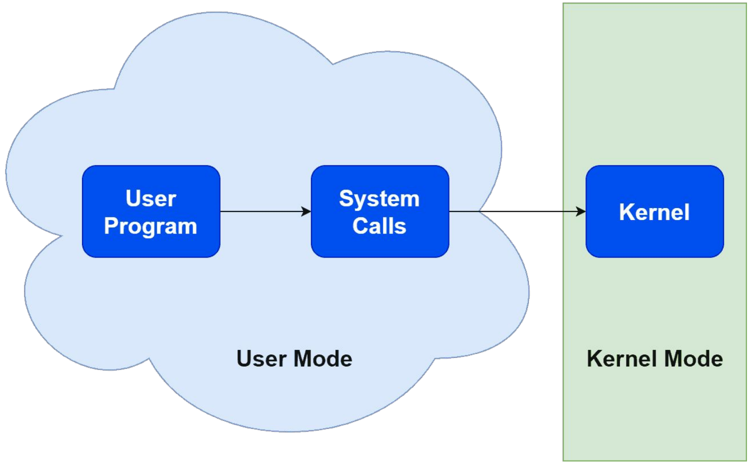
OPERATING SYSTEM

25 Aug 2025

System Call & Interrupt



Department of Computer Science and Engineering

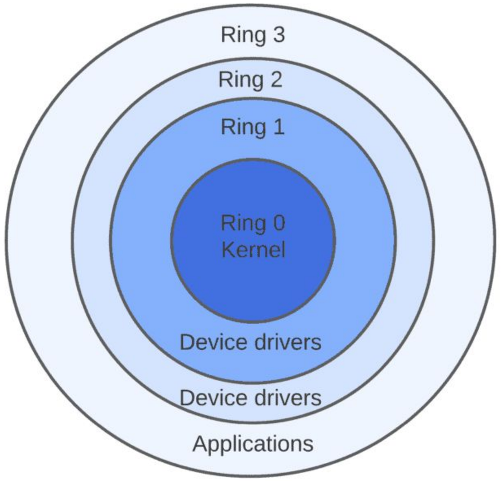
International Institute of Information Technology, Bhubaneswar

Utkalika Satapathy
utkalika@iiit-bh.ac.in

Topics to be covered

Recap

- How to protect concurrent processes from one another?
  - Can one process mess up the code or data of another process?
  - When we virtualize, how do we share safely?
- Modern CPUs have mechanisms for isolation
- **Privileged** and **unprivileged** instructions
  - Privileged instruction access (perform) sensitive information (actions)
  - Regular instructions (e.g., add) are unprivileged
- CPU has multiple modes of operation (Intel x86 CPUs run in 4 rings)
  - Low privilege level (e.g., ring 3) only allows unprivileged instructions
  - High privilege level (e.g., ring 0) allows privileged instructions also



User Mode and Kernel Mode

- User programs runs in user (unprivileged) mode
  - CPU is in unprivileged mode, executes only unprivileged instructions
  - permits only a subset of the instructions to be executed and a subset of the features to be accessed
- OS runs in kernel (privileged) mode
  - CPU is in privileged mode, can execute both privileged and unprivileged instructions
  - When running in kernel mode, the CPU can execute every instruction in its instruction set and use every feature of the hardware.

- CPU shifts from user mode to kernel mode and executes OS code when following events occur (Trap Instructions):
  - **System calls:** user request for OS services
  - **Interrupts:** external events that require attention of OS
  - **Program faults:** errors that need OS attention

- After performing required actions in kernel mode, OS returns back to user program, CPU shifts back to user mode

User Mode and Kernel Mode

- Process Status Word (PSW), the register contains the **condition code bits**, which are set by comparison instructions, the CPU priority, **the mode (user or kernel)**, and various other control bits.
- User programs may normally read the entire PSW but typically may write only some of its fields.
- The PSW plays an important role in system calls and I/O
- Example: Setting the PSW mode bit to enter kernel mode - Privileged or Unprivileged?

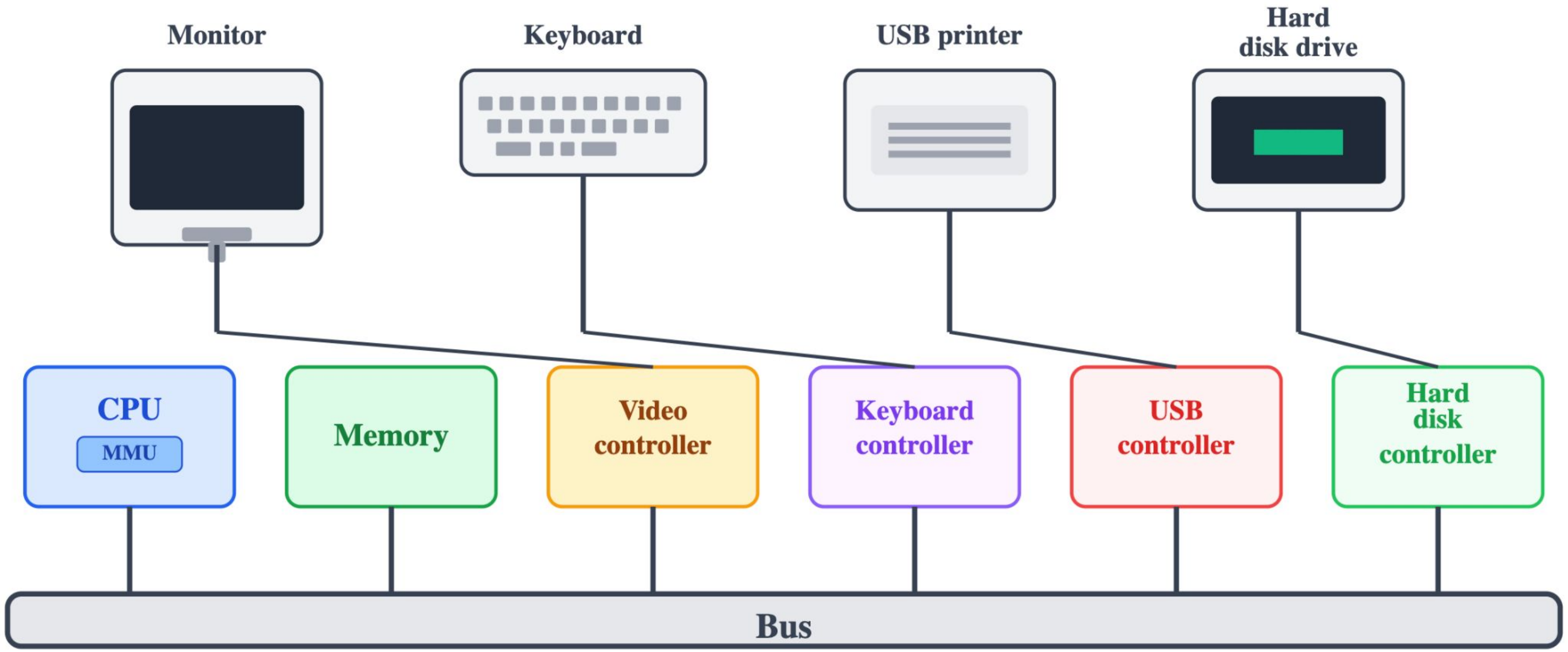
System Calls

- When user program requires a service from OS, it makes a **system call (syscall)**
  - Example: Process makes system call to read data from hard disk
  - Why? User process cannot run privileged instructions that access hardware, to prevent one user from harming another
  - CPU jumps to OS code that implements system call, and returns back to user code after system call completes
- Hence, to obtain services from the OS, a user program must make a system call, which **traps** into the kernel and invokes the operating system.
- The **TRAP instruction** switches from user mode to kernel mode and starts the operating system.
- When the work has been completed, control is returned to the user program at the instruction following the system call.

System Calls

- Normally, user program does not call system call directly, but uses language library functions
    - Example: printf is a function in the C library, which in turn invokes the system call to write to screen
    - https://man7.org/linux/man-pages/man2/write.2.html

Some of the components of a simple personal computer



I/O Devices and Device Drivers

- Apart from the CPU and memory, I/O devices also interact heavily with the OS.
- Every I/O devices generally consist of two parts: a controller and the device itself (shown in previous slide).
- The controller is a chip or a set of chips that physically controls the device.
- It accepts commands from the OS, for example, to read data from the device, and carries them out.
- Because each type of controller is different, different software is needed to control each one.
    - The software that talks to a controller, giving it commands and accepting responses, is called a **device driver.**
    - Each controller manufacturer has to supply a driver for each operating system it supports.
- Example - A scanner may come with drivers for OS X, Windows X, and Linux etc.
- To be used, the driver has to be put into the operating system so it can run kernel mode.

I/O Devices and Device Drivers

- Every controller has a small number of registers that are used to communicate with it.
- For example, a minimal disk controller might have registers for specifying the disk address, memory address, sector count, and direction (read or write).
- To activate the controller, the driver gets a command from the operating system, then translates it into the appropriate values to write into the device registers.
- The collection of all the device registers forms the I/O port space (Details will cover later)

I/O Processing

01 Busy Waiting (Polling)

- User program makes system call ⟶ kernel translates into a procedure call for the respective driver
- Driver starts I/O and continuously polls device status (indicated by some bit)
- CPU remains tied up until I/O completes
- Disadvantage: Wastes CPU cycles

02 Interrupt Driven

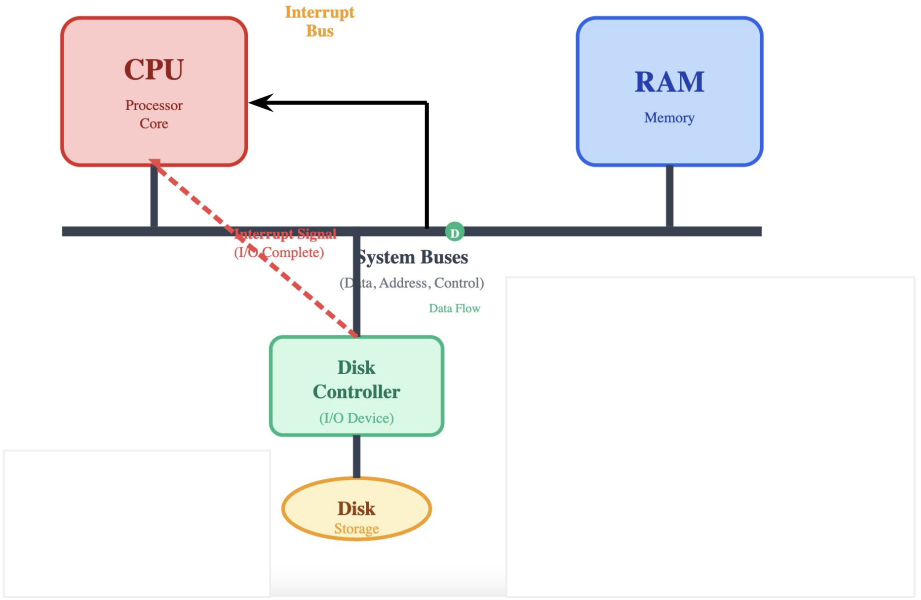
- Driver starts device and requests interrupt when done
- Driver returns immediately, OS blocks caller
- Controller generates interrupt upon completion
- Advantage: CPU free for other tasks while waiting

03 Direct Memory Access (DMA)

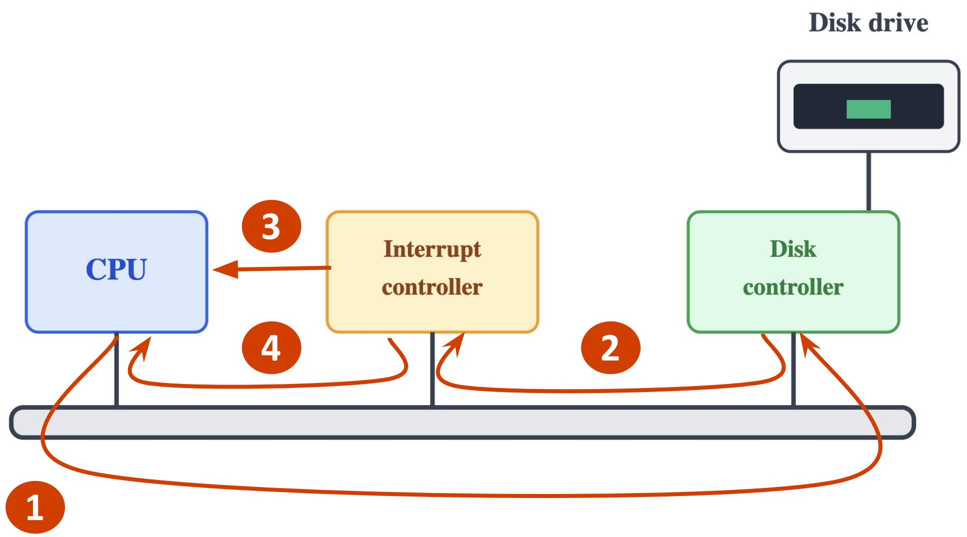
- Special DMA chip controls data flow between memory and device
- CPU sets up DMA with: Number of bytes to transfer, Device and memory addresses, Transfer direction
- DMA operates independently without CPU intervention
- DMA generates interrupt when transfer complete
- Advantage: Frees CPU from data transfer overhead

Interrupts

- In addition to running user programs, CPU also has to handle external events (e.g., mouse click,keyboard input)
- Interrupt = external signal from I/O device asking for CPU's attention
- Example: program issues request to read data from disk, and disk raises interrupt when data isavailable (instead of program waiting for data)



Interrupt Handling



1 Driver Commands Controller

- The driver tells the controller what to do by writing into its device registers
- Controller starts the device operation

2 Controller Signals Completion

- Controller finishes data transfer (reading or writing the number of bytes)
- Signals interrupt controller via bus lines

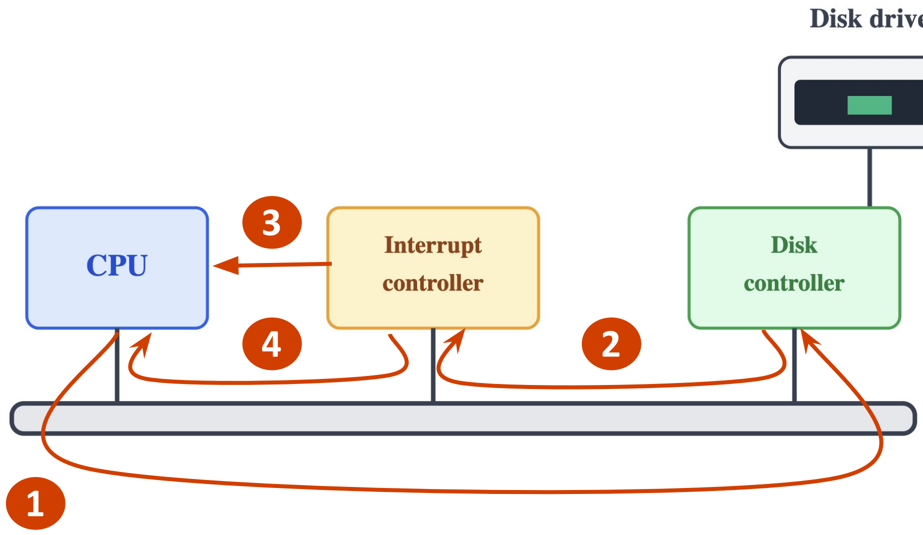
3 Interrupt Controller Notifies CPU

- Interrupt controller checks priority and availability
- Asserts CPU pin if ready to handle interrupt

4 Device Identification

- Interrupt controller puts device number on bus
- CPU reads device ID to identify which device finished
- Enables handling of multiple concurrent devices

Interrupt Handling



- Once the CPU has decided to take the interrupt, the **program counter** and **PSW** are typically then pushed onto the current stack and the CPU switched into kernel mode.
- The device number may be used as an index into part of memory to find the address of the interrupt handler for this device.
- This part of memory is called the **interrupt vector.**

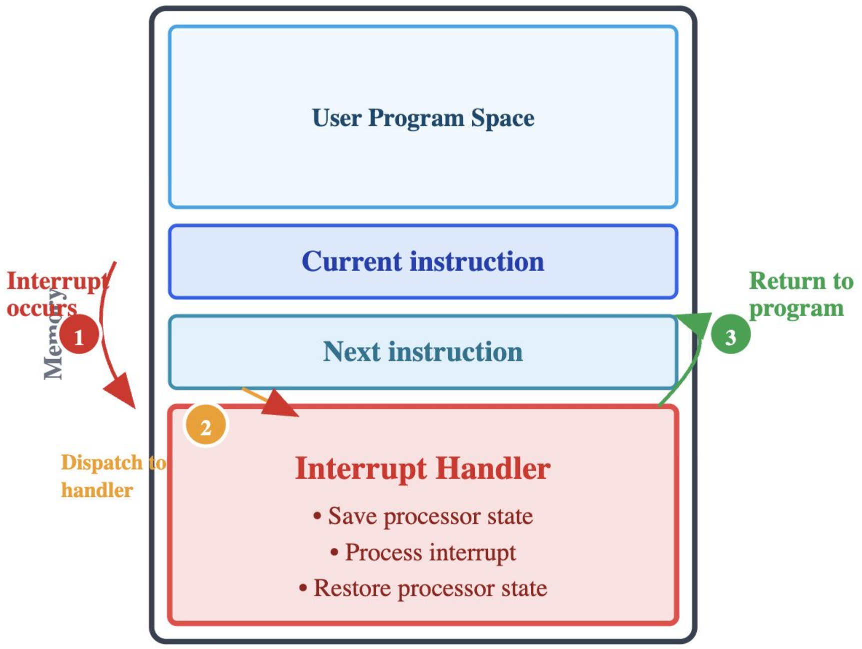
1 **Driver Commands Controller**
- The driver tells the controller what to do by writing into its device registers
- Controller starts the device operation

2 **Controller Signals Completion**
- Controller finishes data transfer (reading or writing the number of bytes)
- Signals interrupt controller via bus lines

3 **Interrupt Controller Notifies CPU**
- Interrupt controller checks priority and availability
- Asserts CPU pin if ready to handle interrupt

4 **Device Identification**
- Interrupt controller puts device number on bus
- CPU reads device ID to identify which device finished
- Enables handling of multiple concurrent devices

Interrupt Processing Flow



🔴 Normal program execution is interrupted

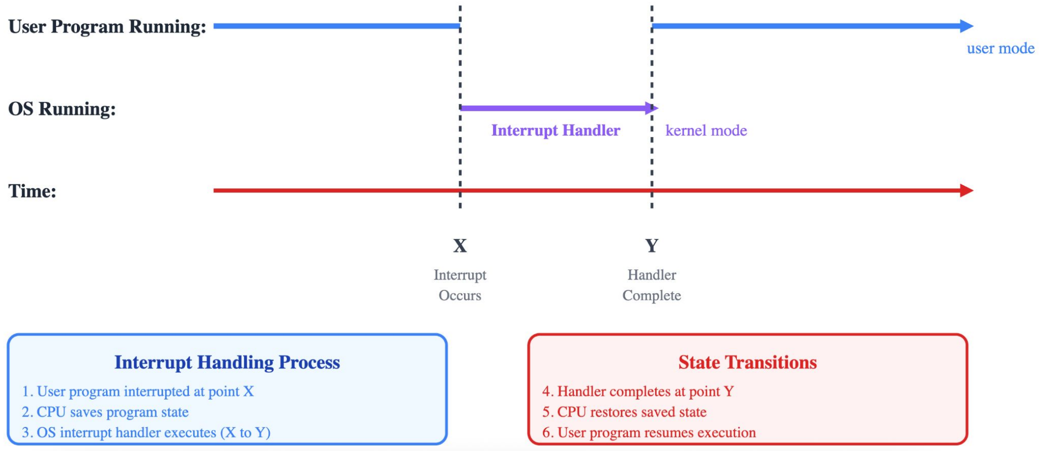
🟠 Control transfers to interrupt handler routine

🟢 Handler completes, execution resumes at next instr

The processor state is saved and restored automatically

Interrupt Handling Process

1. How are interrupts handled?
   a. CPU is running process P and interrupt arrives
   b. CPU saves context of P, runs OS code to handle interrupt (e.g., read keyboard character) in kernel mode
   c. Restore context of P, resume P in user mode
2. Interrupt handling code is part of OS
   a. CPU runs interrupt handler of OS and returns back to user code



Interrupt Handling Process

1. User program interrupted at point X
2. CPU saves program state
3. OS interrupt handler executes (X to Y)

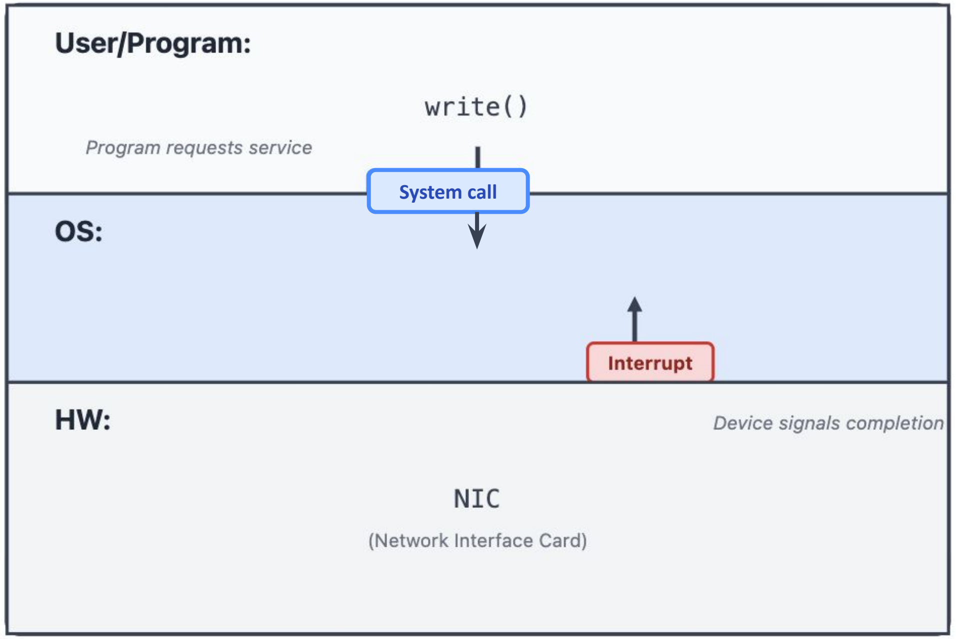
State Transitions

4. Handler completes at point Y
5. CPU restores saved state
6. User program resumes execution

Interrupt Handling Process

1. Device completes its I/O operation
2. Controller sends interrupt signal
3. CPU Save current state
4. CPU jumps to the Interrupt handler
5. Handler processes the interrupt
6. CPU Restores save data
7. Resume normal execution

System calls vs. interrupts



System Calls

- Initiated by user programs
- Synchronous (program waits)
- Request OS services (I/O, memory, etc.)

Hardware Interrupts

- Initiated by hardware devices
- Asynchronous (unexpected timing)
- Signal completion or need attention

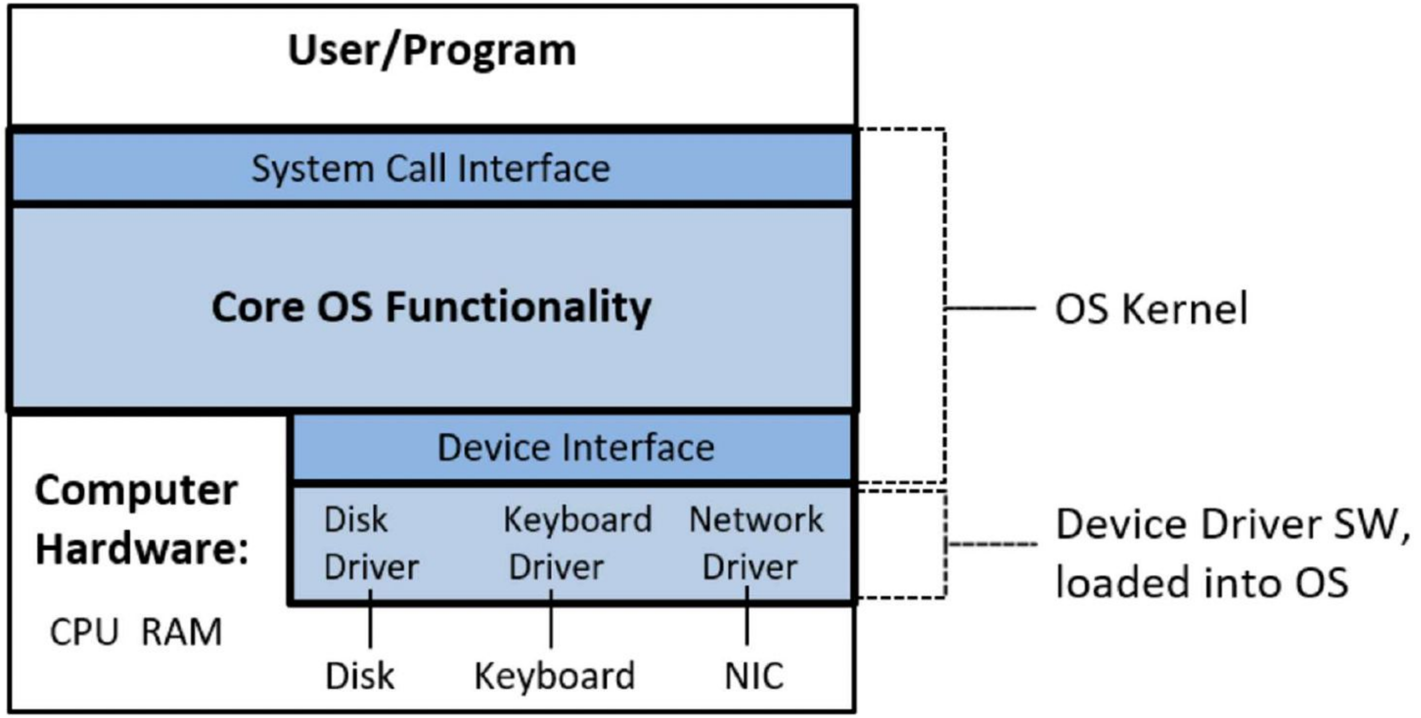


Figure 2. The OS kernel: core OS functionality necessary to use the system and facilitate cooperation between I/O devices and users of the system

Next Class We Will Talk About

- Process States
- Operations with examples from UNIX (fork, exec) and/or Windows.
- Process scheduling

Happy Learning !