Chapter 1

Input/Output Systems

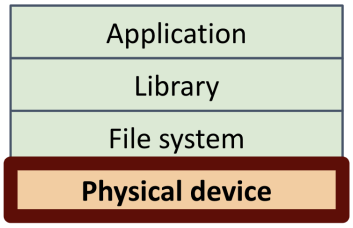
In this lecture, we will discuss how computers communicate with I/O devices, the layered structure of I/O operations, and the various mechanisms used to optimize I/O performance.
This was mostly covered in Computer Architecture.

1.1 I/O System Architecture

1.1.1 Layered Approach to I/O Operations

I/O operations in modern computing systems follow a layered approach, with each layer providing a specific set of responsibilities:

- **Application Layer:** Programs that need to read or write data
- **Library Layer:** Standard libraries that provide I/O functions to applications
- **Operating System Layer:** File systems and device drivers that translate generic operations into device-specific commands
- **Hardware Layer:** Physical devices that perform the actual I/O operations



This layered architecture provides abstraction, allowing applications to perform I/O operations without understanding the underlying hardware details.

1.1.2 Core I/O Services in Operating Systems

Operating systems provide several essential I/O services:

- Loading programs and data from storage devices
- Writing data to display devices (terminals, screens)
- Reading and writing network packets
- Capturing input from various input devices (keyboard, mouse, sensors)

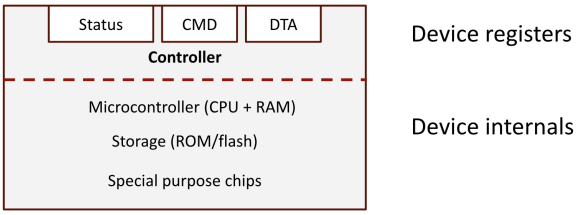
These services form the foundation of how users and applications interact with the computer system.

1.2 Device Interaction Models

1.2.1 Canonical Device Structure

Device communication follows a standardized model that abstracts hardware complexity:

Definition (Canonical Device). *A **canonical device** refers to a standardized model of I/O device interaction where the CPU communicates with a device controller through designated registers, while the internal workings of the device remain hidden from the system.*



In this interaction, we have the following components:

- **Device Controller:** Hardware that interfaces between the device and the system
- **Device Registers:** Control, status, and data registers used for communication
- **Device Driver:** Software component that knows how to communicate with specific devices
- **Signaling Mechanisms:** Methods (polling or interrupts) for the device to signal its status to the CPU

Question: Why do we setup data before setting up the CMD register?

Answer: Race condition if data is not present

Definition (Race Condition).
*A **race condition** occurs when the timing or ordering of events affects the correctness of a program. In device interactions, setting up data before the command prevents race conditions where the device might begin execution before all necessary data is available.*

Thus the following steps are taken to communicate with the device:

1. Wait until the device is ready (check status register)
2. Set data in the data register
3. Issue command by writing to the command register
4. Wait for command completion (check status register)

```
// 1. Wait until device is ready
while (STATUS == BUSY) ;

// 2. Write data to data register
*dtaRegister = DATA;

// 3. Write command to command register
*cmdRegister = COMMAND;

// 4. Wait until command completes
while (STATUS == BUSY) ;
```

1.3 Parameters of I/O Operations

When designing or analyzing I/O systems, five fundamental parameters must be considered:

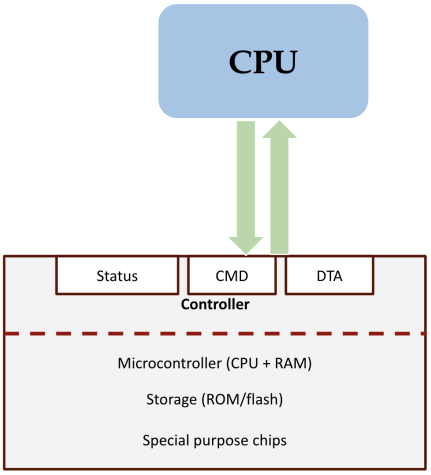
1. **Communication Method:** How does the CPU communicate with the device?
2. **Data Granularity:** What is the size of data transfers (byte, block, etc.)?
3. **Access Pattern:** How is data accessed (sequentially or randomly)?
4. **Notification Mechanism:** How does the device signal the CPU?
5. **Transfer Mechanism:** How is data moved between memory and device?

1.3.1 CPU-Device Communication: Memory-Mapped I/O

Definition (Memory-Mapped I/O (MMIO)). ***Memory-Mapped I/O (MMIO)*** *is a method where device control registers are mapped into the physical address space of the processor, allowing the CPU to access devices using standard memory access instructions (load/store).*

In Memory-Mapped I/O, we have the following characteristics:

- Device registers appear as memory locations to the CPU
- Standard load/store instructions are used for device communication
- Applicable to a wide range of devices
- Particularly effective for high-performance devices (fast storage, network interfaces, displays)



1.3.2 Data Granularity and Access Patterns

Different I/O devices operate with different data sizes and access patterns:

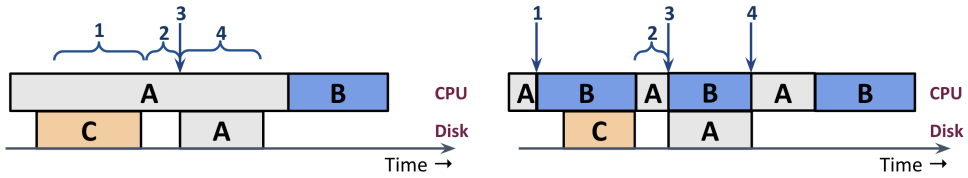
- **Data Granularity:**
  - *Byte-oriented devices:* Transfer one byte at a time (e.g., keyboards, serial ports)
  - *Block-oriented devices:* Transfer blocks of data (e.g., disk drives, network cards)
- **Access Patterns:**
  - *Sequential access:* Data must be accessed in order (e.g., tape drives)
  - *Random access:* Data can be accessed in any order (e.g., disk drives, SSDs)

These characteristics significantly impact the design of device drivers and the overhead involved in data transfers.

1.4 Notification Mechanisms

1.4.1 From Polling to Interrupts

Waiting for device operations to complete poses a challenge for system efficiency. Two main approaches address this:



Definition (Polling). ***Polling*** *(or busy-waiting) is a notification technique where the CPU periodically checks a device's status register to determine if an operation has completed.*

Definition (Interrupt). *An* ***interrupt*** *is a hardware signal sent from a device to the CPU that causes the CPU to temporarily suspend its current execution, save its state, and execute an interrupt handler routine.*

1.4.2 Comparing Polling and Interrupts

Aspect	**Polling**	**Interrupts**
Mechanism	CPU periodically checks device status	Device signals CPU when attention needed
CPU Utilization	Wastes CPU cycles when events are infrequent	Efficient for unpredictable or infrequent events
Overhead	Low per-check overhead	Higher overhead for context switching
Predictability	Predictable timing	Less predictable
Best Use Cases	High-frequency events, short wait times	Unpredictable events, long wait times

Definition (Livelock). ***Livelock*** *is a situation where a system is continuously responding to interrupts and cannot make progress on its main tasks, similar to deadlock but with processes actively running rather than blocked.*

1.4.3 Optimizing Notification Mechanisms

Modern systems use sophisticated approaches to balance efficiency:

- **Hybrid Approaches:** Using both polling and interrupts depending on workload characteristics
- **Interrupt Coalescing:** Delaying and batching multiple interrupts to reduce overhead
- **Adaptive Strategies:** Dynamically switching between polling and interrupts based on system load and device activity

1.4.4 Data Transfer Mechanisms

The final aspect of I/O operations concerns how data is transferred between main memory and device controllers.

Definition (Programmed I/O (PIO)). ***Programmed I/O (PIO)*** *is a data transfer technique where the CPU directly controls data movement between memory and a device, executing an instruction for each data unit transferred.*

Definition (Direct Memory Access (DMA)). ***Direct Memory Access (DMA)*** *is a data transfer technique that allows a device controller to directly transfer data to or from main memory without CPU intervention, after initial setup by the CPU.*

Aspect	Programmed I/O	Direct Memory Access
CPU Involvement	CPU handles each data transfer	CPU only sets up transfer, then free for other tasks
Efficiency	Efficient for small transfers	Efficient for large transfers
Complexity	Simpler implementation	More complex hardware needed
CPU Overhead	High, proportional to data size	Low, independent of data size
Best Use Cases	Small data transfers, simple devices	Large data transfers, high-performance devices

- **Programmed I/O (PIO):**
  - CPU directly controls data transfer, telling the device what data is
  - Requires one instruction for each byte/word transferred
  - Efficient for small transfers but consumes CPU cycles proportional to data size
- **Direct Memory Access (DMA):**
  - CPU only tells the device where data is located in memory
  - Controller is granted access to the memory bus
  - Device transfers data to/from memory without CPU intervention
  - Highly efficient for large data transfers

1.5 Direct Memory Access (DMA)

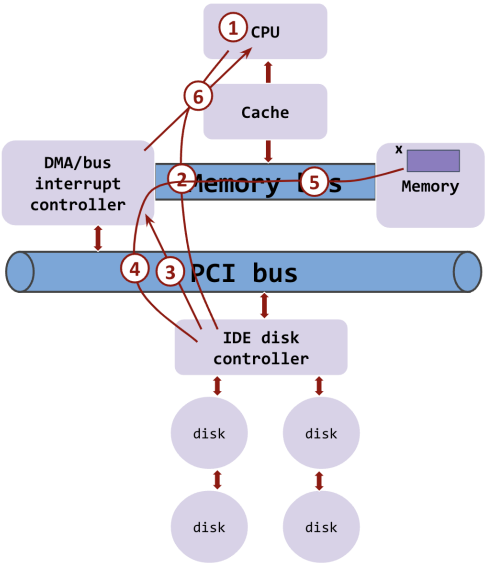
Definition (Direct Memory Access (DMA)). *Direct Memory Access (DMA) is a method that allows hardware subsystems to access main system memory independently of the CPU, enabling efficient data transfer between I/O devices and memory.*

1.5.1 DMA Controller Operation

The DMA controller facilitates direct data transfer between peripheral devices and memory without constant CPU intervention, significantly improving system efficiency for data-intensive operations.

The following steps illustrate a typical DMA transfer sequence:

1. The device driver receives an instruction to transfer disk data to a buffer at address X.
2. The driver commands the disk controller to transfer C bytes from disk to the buffer at address X.
3. The disk controller initiates the DMA transfer operation.
4. The disk controller sends each byte to the DMA controller.
5. The DMA controller transfers bytes to buffer X, incrementing the memory address and decrementing C until C = 0.
6. When C = 0, the DMA controller interrupts the CPU to signal completion of the transfer.



1.6 Device Management in Operating Systems

1.6.1 The Device Driver Challenge

Modern computing systems must interface with numerous devices, each with unique protocols and requirements. This diversity creates significant challenges for operating system design.

Definition (Device Driver). *A device driver is a specialized component of the operating system kernel that directly interfaces with hardware devices, translating between the standardized OS interfaces and device-specific protocols.*

Device drivers solve the challenge of hardware diversity by:

- Supporting a standard, internal interface within the OS
- Enabling the same kernel I/O system calls to interact with different physical devices
- Providing device-specific implementations of standard operations

1.6.2 Principles of Device Driver Design

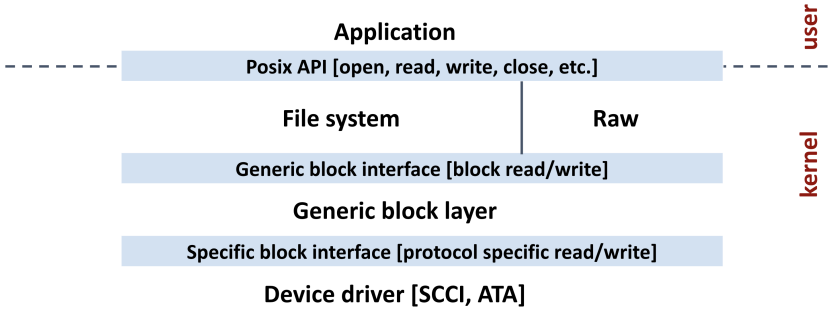
Device drivers demonstrate encapsulation in system design. Different drivers adhere to the same API, allowing the OS to interact with diverse hardware through consistent interfaces. The OS implements support for APIs based on device class rather than specific hardware models.

This approach requires:

- Well-designed interfaces and APIs
- Careful balance between versatility and specialization
- Layered API structure to manage complexity

1.6.3 Complexity of API Layers

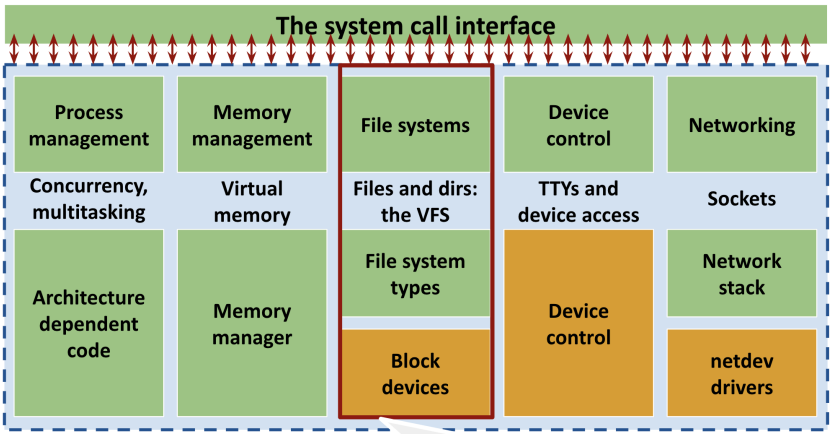
The file system stack exemplifies the layered approach to I/O management:



This layering allows for abstraction and modularity while providing necessary functionality at each level.

1.6.4 OS Device Structure

The overall device structure in an operating system organizes components into logical layers:



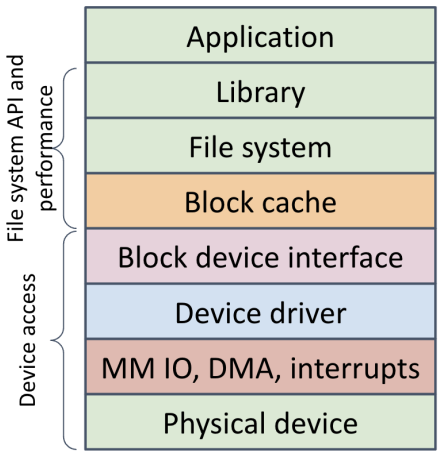
- **Process Management:** Handles process creation, scheduling, synchronization, and termination, allowing multiple programs to run concurrently.
- **Memory Management:** Controls allocation and deallocation of memory resources, implements virtual memory, and manages address translation.
- **File Systems:** Provides abstractions for persistent data storage, organizing files and directories while managing access permissions.
- **Device Control:** Interfaces with hardware peripherals through device drivers, translating generic commands into device-specific operations.
- **Networking:** Implements network protocols and provides interfaces for network communication, enabling data exchange between systems.

The system call interface serves as the boundary between user applications and these kernel components, providing a standardized way for programs to request services from the operating system. Through system calls, applications can interact with all these subsystems without needing to understand their internal implementation details.

1.6.5 General I/O Abstraction Stack

I/O systems are accessed through a series of layered abstractions that progressively translate between user-level operations and hardware-specific details:
These layers provide:

- Caching of recently read disk blocks
- Buffering of recently read blocks
- A unified interface to diverse devices
- Fixed-size block data operations
- Translation between OS abstractions and hardware-specific details
- Control of hardware registers, bulk data transfers, and OS notifications



1.7 Storage Systems

1.7.1 Storage Media Evolution

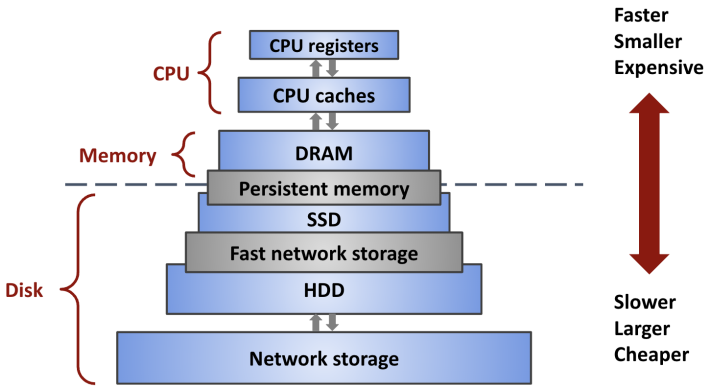
Definition (Persistent Storage). *Persistent storage refers to data storage technologies that retain information even when power is removed from the system.*

Computer systems have employed various storage media for persistent data:

- Punched paper cards
- Magnetic tapes
- Floppy disks
- Magnetic hard disks
- Optical disks
- USB flash drives
- Solid-state drives

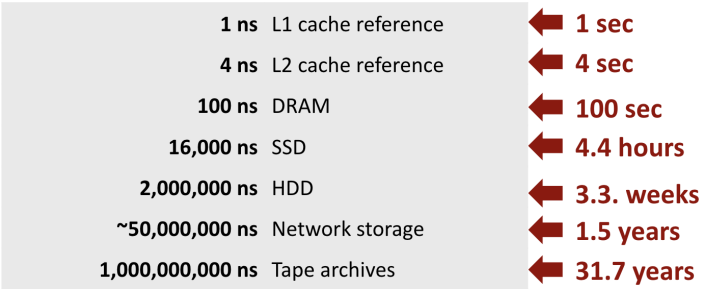
1.7.2 The Storage Hierarchy

Modern computing systems organize storage into a hierarchy based on speed, capacity, and cost: *Main memory for currently used data* and *Disk for storing application data*



1.7.3 Performance Considerations: Latency

Understanding storage performance requires awareness of access latency across different technologies:



These latency figures are critical knowledge for system designers and software engineers when optimizing data access patterns.

1.7.4 Disk Storage Characteristics

Magnetic and solid-state disks serve as the predominant secondary storage devices in modern systems:

Definition (Disk Block). *A disk block (or page) is the fundamental unit of data storage and retrieval for disk-based storage systems.*

Key characteristics of disk storage include:

- Non-uniform access times, unlike RAM
- Access time variations based on disk technology (magnetic vs. flash)
- Performance differences between sequential and random access patterns

The relative placement of data on physical disks significantly impacts application performance, making storage optimization a critical consideration in system design.

1.7.5 Data Integrity in Storage Systems

Modern storage systems employ various techniques to ensure data integrity:

- Error detection and correction codes to handle bit errors
- Capabilities to detect data corruption
- Error handling at both controller and OS levels

Despite these safeguards, storage devices remain a potential single point of failure in computing systems, with limitations in performance, capacity, and reliability.

1.8 Redundant Array of Inexpensive Disks (RAID)

Definition (Redundant Array of Inexpensive Disks (RAID)). *RAID (Redundant Array of Inexpensive Disks) is a storage virtualization technology that combines multiple physical disk drives into a single logical unit for improved performance, capacity, or reliability.*

1.8.1 Storage System Requirements

Effective storage systems must satisfy three key requirements:

- Speed: Data must be accessible with minimal latency
- Reliability: Retrieved data must match what was originally stored
- Affordability: Cost must be reasonable relative to system requirements

RAID technology addresses these requirements by building logical storage volumes from multiple physical disks, providing:

- Higher throughput through data striping
- Improved reliability through redundancy
- Cost-effective scaling of storage capacity

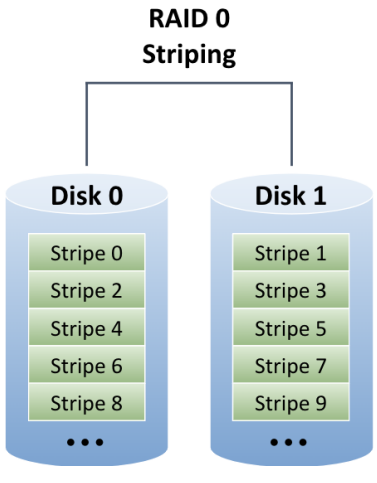
1.8.2 RAID 0: Striping

RAID 0 focuses on performance optimization by distributing data across multiple disks, allowing parallel access to improve throughput and reduce access times.

Definition (RAID 0). *RAID 0 implements block-level striping without parity or mirroring, distributing data evenly across multiple disks without redundancy.*

Characteristics of RAID 0:

- Files are striped across multiple disks
- No data redundancy or fault tolerance
- Provides maximum performance and full storage capacity
- Cumulative bandwidth across all member disks
- Total storage capacity equals the sum of all disk capacities
- Reduced reliability as disk count increases (higher probability of failure)



Example 1.8.2.1. *In a four-disk RAID 0 array, each with a mean time between failures (MTBF) of 100,000 hours, the expected MTBF for the entire array would be approximately 25,000 hours (one-fourth of a single disk's MTBF), since failure of any single disk results in data loss for the entire array.*

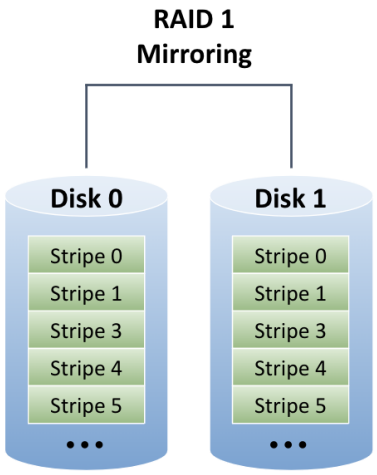
1.8.3 RAID 1: Mirroring

RAID 1 focuses on data reliability through complete redundancy, creating exact copies of data across multiple disks to protect against hardware failures.

Definition (RAID 1). *RAID 1 implements disk mirroring, where identical data is written to two or more drives, providing complete data redundancy.*

Characteristics of RAID 1:

- Data blocks are duplicated across multiple drives
- Excellent protection against disk failure
- Does not protect against data corruption
- Usable storage capacity equals that of a single disk
- Read operations can be parallelized
- Write performance equivalent to a single disk
- Higher cost per usable gigabyte
- Commonly used for critical systems and sensitive information

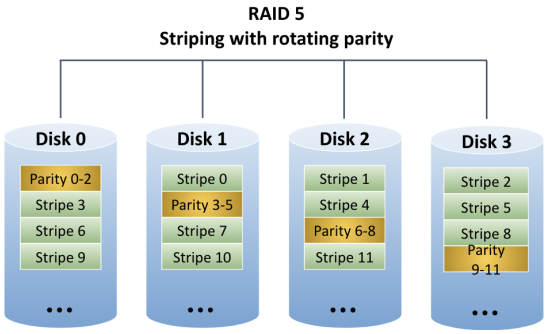


1.8.4 RAID 5: Distributed Parity

RAID 5 balances performance and reliability by distributing both data and parity information across all disks in the array, providing fault tolerance with better storage efficiency than mirroring.

RAID 5 has the following features:

- Distributed parity blocks across all disks
- Can survive failure of any single disk
- Data can be reconstructed using XOR operations on remaining drives
- Good read performance: approximately (N-1) times that of a single disk
- Write operations are more complex due to parity calculations
- Storage efficiency: usable capacity is (N-1) disks in an N-disk array
- Commonly used in data center environments



Definition (RAID 5). *RAID 5 implements block-level striping with distributed parity, providing fault tolerance while using less storage for redundancy than mirroring.*

Definition (Parity). *Parity is a fault tolerance mechanism where redundant data is calculated and stored to enable recovery from single-disk failures. For a set of blocks S_i through S_j, the parity P_{i-j} is calculated as: $P\_i - j = S\_i \oplus S\_i + 1 \oplus ... \oplus S\_j$, where $\oplus$ represents the XOR operation.*

Example 1.8.4.1. *In a 5-disk RAID 5 array, if disk 3 fails, its data can be reconstructed by performing XOR operations on the corresponding blocks from the other 4 disks. This allows the system to continue operation despite the failure, though with degraded performance until the failed disk is replaced and rebuilt.*