Operating Systems
Tutorial 8: I/O Systems

Yonghao Lee

May 24, 2025

1 Introduction to I/O Systems

We can categorize I/O devices into 3 groups, depending on what they supply:

- **Input only:** keyboard, mice, remote controllers. These devices only send data to the system without receiving any output back.
- **Output only:** most displays, headphones. These devices only receive data from the system to present information to users.
- **Both:** disks, network cards. These devices can both send data to the system and receive data from it, enabling two-way communication.

The OS has control of those devices. It handles commands by processing requests from applications and translating them into device-specific operations. It manages interrupts that signal when devices complete operations or encounter problems. It also deals with errors by detecting and managing hardware malfunctions.

The OS supplies abstractions for the devices. This means it provides a simplified, uniform interface that hides the complex implementation details of how each device actually works from users and applications. It also handles error management so users don't need to deal with low-level hardware problems directly.

The OS should allow using I/O devices and utilizing CPU at the same time. This means the system can perform computations while simultaneously handling input and output operations, rather than having to wait for one to finish before starting the other.

It should allow parallel or concurrent access to devices, meaning multiple processes can use I/O devices simultaneously without interfering with each other. It should also protect access when necessary to prevent conflicts and maintain data integrity when multiple processes want to use the same device.

1.1 Kernel I/O Subsystem

The kernel I/O subsystem is the part of the kernel that is specifically used for I/O operations.

The OS must manage I/O requests through several key mechanisms:

- **Scheduling:** The OS schedules operations, for example disk accesses, based on device structure and performance considerations. This involves ordering requests to optimize system efficiency.
- **Synchronization:** The OS forbids concurrent accesses to a device when such access can cause conflicts or data corruption.
- **Error Handling:** An I/O request might fail due to either transient reasons (temporary issues that may resolve) or permanent reasons (hardware failures that require different handling).
- **Request Optimization:** It might be useful to postpone or delay requests to minimize the number of accesses in expensive cases, improving overall system performance.

2 Hardware Architecture for I/O

2.1 Bus Architecture for I/O Devices

One way to connect I/O devices is using a bus, which is a shared communication system that connects multiple hardware devices. It connects to the CPU as well and allows data transfer between all connected components.

In a computer, we have different kinds of buses. For example, USB (Universal Serial Bus) connects external devices, and there is always a system bus that connects the CPU to the memory. I/O devices connected to a bus need to listen to the bus to see if any command has arrived for them.

This bus architecture has both advantages and disadvantages:

- **Advantages:**
  1. **Easy to connect new devices:** Adding a new device simply requires connecting it to the existing bus without rewiring the entire system.
  2. **Standardization benefits:** If two computers use the same bus standard, devices can easily work on both systems without hardware modifications.
  3. **Cost-effective:** Sharing a single communication pathway among multiple devices is cheaper than having dedicated connections for each device.
- **Disadvantages:**
  1. **Limited scalability:** As more devices are added to the bus, the available bandwidth must be shared among all devices, reducing performance for each device.
  2. **Bottleneck:** Since all devices share the same communication pathway, only one device can transmit data at a time, creating potential delays.
  3. **Not general:** Since buses are specialized and not universal, devices designed for one bus cannot easily work with another.

2.2 Hardware Ports and Connections

Devices are connected to computers using ports. Serial ports allow transfer of one bit at a time, whereas parallel ports allow transfer of multiple bits at a time by using multiple data lines.

There are many cables and connection types, like USB that we discussed above.

2.3 Device Controllers

Most I/O devices have controllers. These controllers are often designed as chips or cards that allow the device to be plugged into them.

A device controller acts as an intermediary between the computer system and the I/O device, for example, a graphics card controller that converts drawing commands into pixel operations.

The interface between the controller and the device is usually a standard interface.

For example, in hard disks, the controller should know how to insert and retrieve data at a given address. This requires conversion to a specific location on disk (cylinder, sector, etc.). It should also know how to detect and fix errors.

2.4 Non-Uniform Memory Access (NUMA) and I/O

In multi-processor systems, the architecture affects I/O performance. Consider a system with 2 CPUs and 2 RAM modules, organized into two nodes:

- **Node 1**: CPU_1 and RAM_A
- **Node 2**: CPU_2 and RAM_B

2.4.1 Memory Access Types:

1. **Local Access (Fast):**
   - CPU_1 accessing RAM_A (within Node 1).
   - CPU_2 accessing RAM_B (within Node 2).
   - These accesses have lower latency.
2. **Remote Access (Slower):**
   - CPU_1 accessing RAM_B (across nodes, via interconnect).
   - CPU_2 accessing RAM_A (across nodes, via interconnect).
   - These accesses have higher latency due to the interconnect traversal.

This architecture affects I/O performance because I/O devices and their data transfers must work within this memory hierarchy.

3 Device Drivers

We talked about controllers, which are hardware components performing as interfaces for devices.

As the OS, how do we know how to talk with the controllers? The OS uses an abstract interface that each device manufacturer should implement. They implement this interface in a program called a driver.

The OS activates the driver when the system boots or when a device is plugged in. Then the following process occurs:

1. The OS makes a request, such as "write this data to the disk"
2. The OS calls the appropriate driver function, like `write()`
3. The driver translates the high-level request into specific commands the controller understands

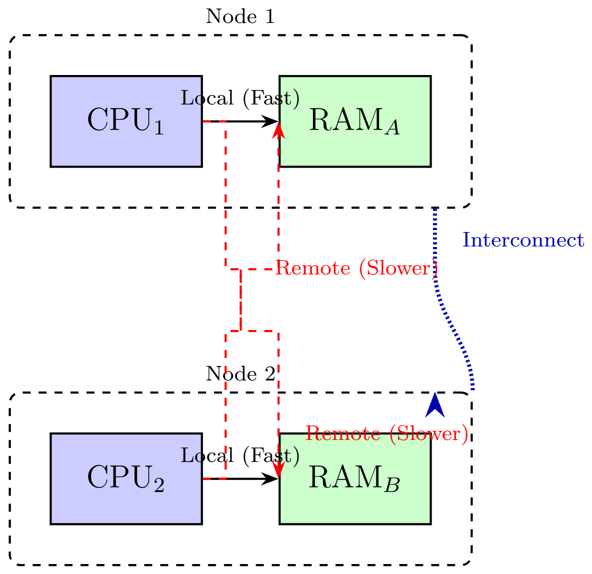

Figure 1: NUMA with 2 CPUs and 2 RAM modules.

4. The driver writes these commands and data to the controller's registers
5. The controller performs the actual physical I/O operation
6. The controller reports back to the driver when done or if there was an error
7. The driver reports the result back to the OS

For example, when writing to a disk, the OS requests "write file.txt to disk," the driver translates this to "write 1024 bytes to cylinder 5, sector 10," puts these commands in controller registers, and the controller physically moves the disk head to write the data.

Drivers are programs that allow the OS to communicate with the controllers. They are considered trusted and run in kernel mode, but writing drivers is challenging.

Drivers act as extensions of the OS and sit in the lowest layer of the operating system, positioned between OS abstractions and hardware as shown in the system layers.

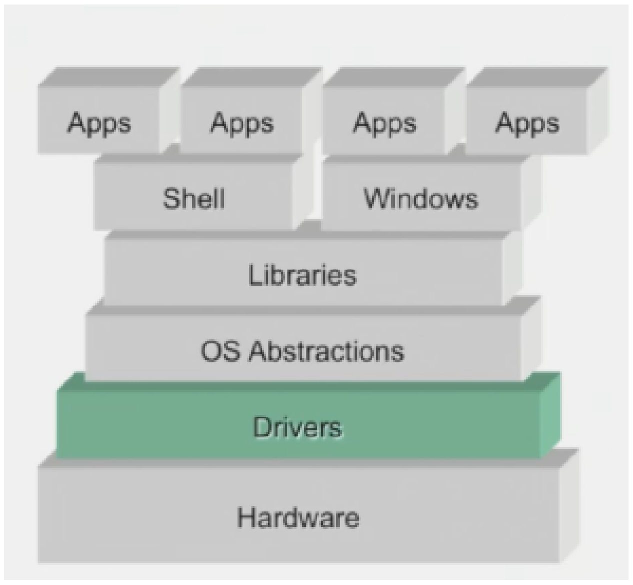

Figure 2: System layers showing driver position

There are many devices but few operating systems. Therefore, each OS defines standard driver interfaces, and device manufacturers write drivers that implement these standard interfaces. This allows the OS to communicate with different hardware using consistent commands.

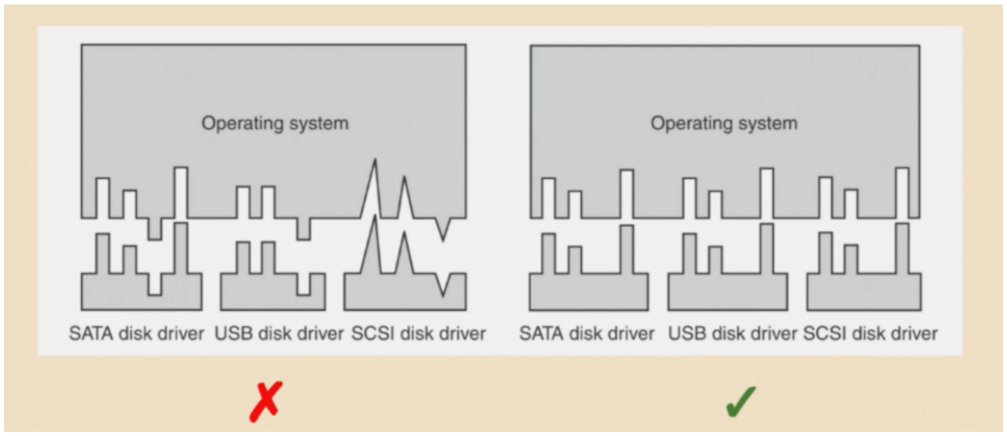

Figure 3: Driver interface standardization. Left (incorrect): Each driver (SATA, USB, SCSI disk drivers) has different interfaces with the operating system, making communication inconsistent. Right (correct): All drivers implement the same standard interface, allowing the OS to use uniform commands regardless of the underlying hardware type.

4 I/O Communication Methods

After a driver sends commands to a controller, it needs to know when the I/O operation is complete. There are two main approaches for this communication:

- **Interrupts:** The device sends a signal to the CPU when it finishes the operation.
- **Polling:** The CPU continuously checks relevant registers of the device controller to see if the operation is finished.

Interrupts are the preferred approach because they prevent busy waiting and allow the CPU to work on other tasks while the I/O operation is in progress.

4.1 Interrupt-Based I/O Process

1. The CPU sets relevant controller registers with the operation parameters (what data, where to write, etc.)
2. The CPU sends the controller a command to start the operation
3. The device performs the requested I/O operation
4. The CPU continues working on other tasks until an interrupt is received
5. When the interrupt arrives, the CPU checks the operation for correctness
6. The CPU saves the result in main memory if necessary

This interrupt-driven approach allows for efficient multitasking, as the CPU is not blocked waiting for slow I/O operations to complete.

4.2 DMA (Direct Memory Access)

Direct Memory Access (DMA) is an optimization technique that eliminates the need for CPU intervention during data transfer operations. It is called ”direct” because data moves directly from the device to memory, bypassing the CPU entirely during the actual transfer. In the interrupt-based I/O process described earlier, the CPU must handle the data transfer from the device to main memory. DMA removes this burden from the CPU.

4.2.1 How DMA Works

1. **CPU programs the DMA controller:** The CPU configures the DMA controller's registers with transfer parameters:
    - Address register: specifies the memory location where data should be stored
    - Count register: indicates how many bytes to transfer
    - Control register: defines the direction and type of transfer operation
2. **DMA controller requests transfer:** The DMA controller initiates the data transfer operation directly with the device.
3. **Direct data transfer:** Data moves directly from the device (such as a disk drive) to main memory without passing through the CPU.

4. **Transfer acknowledgment:** The DMA controller manages the transfer completion and confirms when the operation is finished.
5. **Interrupt notification:** Once the transfer is complete, the DMA controller sends an interrupt to notify the CPU that the data is ready in memory.

4.2.2 Advantages of DMA

DMA provides several significant benefits over CPU-controlled data transfers:

- **CPU efficiency:** The CPU is freed from performing repetitive data copying tasks and can execute other processes during I/O operations.
- **Improved performance:** Direct hardware-to-memory transfers are typically faster than CPU-mediated transfers.
- **Concurrent processing:** While the DMA controller handles data movement, the CPU can simultaneously work on other computational tasks.

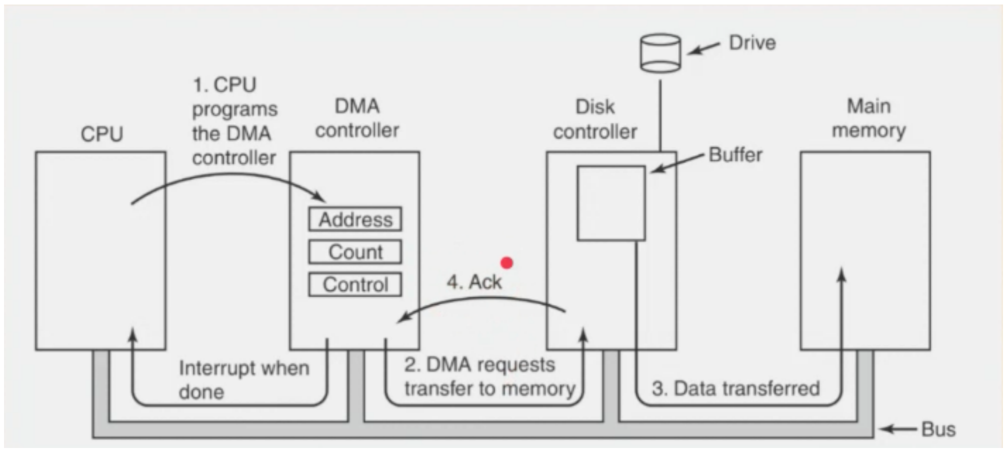

Figure 4: DMA operation showing direct data transfer from device to memory without CPU intervention in the data path.

5 Storage Devices

5.1 Hard Disk Drive (HDD)

These are non-volatile storage devices where data is organized into basic transfer units called sectors. HDDs can store much more data than RAM at lower cost, but they are mechanically fragile—scratches and dust can cause damage. They are significantly slower than solid-state storage due to their physical-magnetic mechanism that requires mechanical movement for reading and writing operations.

5.2 Solid State Drive (SSD)

SSDs use flash memory technology where data is stored in memory cells. Each cell stores electrical charge, and changing the charge level changes the stored bits. They are built in a 3D structure with these cells arranged in thousands of rows and columns, stacked in multiple layers. Unlike HDDs, SSDs have no moving parts, making them more durable and faster. However, each write operation causes slight physical wear to the cells, limiting their lifespan through write/erase cycles.

SSDs are the common storage technique today. They are faster than HDDs, though more expensive. We use SSDs in our cellphones and laptops, which is why these devices usually contain less storage capacity.

5.3 Improving Disk Performance

Compared to the CPU, disk latency is very high—seek time is on the order of milliseconds. We can increase disk performance by allowing parallel reading, but the data should be spread across several disks. Fortunately, this is affordable due to the low cost of disks.

In 1988, several methods were introduced to improve performance and reliability. These were called RAID (Redundant Array of Inexpensive Disks) methods.

5.3.1 RAID 0 - Striping

We have several disks and spread our data across them. Each portion is called a stripe, which spans across multiple disks. However, if one disk fails, all data is lost since it's distributed without redundancy.

5.3.2 RAID 1 - Mirroring

We save a duplicate copy on separate disks. This is called mirroring, so if one disk fails, we still have a copy. Additionally, both disks can be read simultaneously for better read performance.

5.3.3 RAID 4 - Parity

- All disks except for one contain the data.
- The remaining disk is called "the parity disk".
  - Contains the XOR of data from all other disks.
  - $parity = (disk\ 1) \oplus (disk\ 2) \oplus \cdots \oplus (disk\ n)$
- **Why?**
  - If one disk fails, can be used to retrieve data.
  - Remember that $x \oplus (x \oplus y) = y$
- **Example with 3 disks:**

  - Disk 1: 1101, Disk 2: 0110
  - Parity disk: $1101 \oplus 0110 = 1011$
  - If Disk 1 fails: $missing = 0110 \oplus 1011 = 1101$ (recovered!)
  - If Disk 2 fails: $missing = 1101 \oplus 1011 = 0110$ (recovered!)

5.4 Disk Partitions

We can divide the disk into separate, independent portions called partitions. Each partition can host a different file system. This is an example of virtualization—even though each partition is treated as an independent disk and considered separate storage, they are all part of the same physical disk.

The partitions are listed in the partition table, which is also stored on the disk. Each partition entry contains the sector where it starts and its length.

6 User Interface Devices

6.1 Keyboard

Each key is associated with a number called the scan code. This is not the ASCII code—it's an internal hardware identifier.

Whenever we press a key, the keyboard controller sends an interrupt to the CPU. When we release it, another interrupt is sent. The keyboard driver maintains a list of which keys are currently pressed and communicates this to the operating system.

The desktop environment (such as GNOME in Linux) is a graphical interface that interprets user input and manages interaction with various devices.

Although the keyboard and monitor are separate, independent devices, we expect the characters we type to appear on the screen. This behavior is called echoing.

The keyboard driver extracts key information using the I/O port associated with the keyboard. Everything else—including case conversion, character mapping, and language support—is handled by software layers above the driver.

6.2 Mouse

- A mouse message is generated when:
  - A mouse button was clicked.
  - The mouse was moved "significantly".
    - For example, at least 0.1 millimeters.
    - Sometimes this unit is called a mickey.
- This message includes Δx, Δy and buttons status.
  - These are "how much the x and y coordinate changed".

- The operating system interacts with the GUI to notify it.
  - The GUI manager keeps tracking on what window is currently ”active”, and processes the click/cursor move events.

A System Startup

We have been talking about many topics on the premise that the OS is running. But how does the OS start running on the computer?

The OS files are stored on the disk and must be transferred to main memory (RAM) where the CPU can execute them.

On the disk, there is a special sector called sector 0, known as the Master Boot Record (MBR). This is not where the OS itself is stored.

This sector contains a short boot code and the partition table at the end. Note that GPT (GUID Partition Table) is actually a newer alternative to MBR, not the same thing.

At least one partition should be marked as ”active” or ”bootable”. The BIOS (Basic Input/Output System) reads this partition to continue the boot process. BIOS is firmware that comes with the PC and operates independently of any operating system.

A.1 Boot Loaders

However, the MBR is very small (only 512 bytes), so the boot code it contains must be extremely short. For more complex operating systems, this small space is insufficient. Therefore, the MBR typically contains just enough code to locate and load a larger, more sophisticated program called a boot loader.

The boot loader is responsible for loading the kernel code into memory, configuring it, and starting the kernel execution.

The kernel is stored as files on the disk, but neither the CPU nor the BIOS understands the abstraction of files or file systems. Therefore, the boot loader must have built-in knowledge of the file system structure to locate and access the kernel files directly from the disk sectors.

A.2 Multi-Booting

If more than one partition is marked as bootable, you can choose which partition to boot from, allowing you to run different operating systems on the same computer.

In the BIOS, we can change the default boot partition or change the boot priority order of different storage devices.

Different partitions can contain different operating systems. These partitions can use the same file system type (such as both using NTFS) or different file systems entirely. We will discuss file systems in more detail later in the course.