

UNIT 6

Input/Output Device Management

(4 Hrs) — Operating System (CACS251)

6.1 Principles of I/O Hardware

6.1.1 I/O Devices

I/O devices are the means by which a computer communicates with the outside world. They can be divided roughly into two categories:

Type	Description	Examples
Block devices	Store information in fixed-size blocks (512 B – 32 KB), each with its own address; blocks can be read/written independently (<i>seek</i> is possible)	Hard disks, SSDs, CD/DVD, USB drives
Character devices	Deliver or accept a <i>stream of characters</i> ; not addressable, no seek	Keyboard, mouse, printers, network interfaces, terminals
Others	Do not fit either model	Clocks/timers (generate interrupts only)

Device	Approx. data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
Gigabit Ethernet	≈ 125 MB/sec
SATA SSD	≈ 500 MB/sec
USB 3.0	≈ 625 MB/sec
PCIe NVMe SSD	several GB/sec

6.1.2 Device Controllers

An I/O unit typically has two components: a **mechanical component** (the device itself) and an **electronic component** — the **device controller** or adapter (a chip or circuit board). The controller has registers (command, status, data) and often a small buffer. The OS deals with the controller, not the device. Example: the disk controller converts the serial bit stream from the drive into blocks of bytes, performs error correction (ECC), and makes the block available in its buffer for the CPU or DMA to copy to memory.

6.1.3 Memory-Mapped I/O

How does the CPU communicate with the control registers and data buffers of a controller?
Two approaches:

Separate I/O port space	Memory-mapped I/O
Each control register is assigned an I/O port number ; special instructions IN and OUT are used	Control registers are mapped into the memory address space ; each register has a unique memory address (usually at the top of the address space)
Needs special I/O instructions (assembly)	Ordinary instructions (e.g., C pointers) can be used — no assembly needed
Memory and I/O address spaces are separate	No special protection mechanism is needed: keep those addresses out of user address spaces
Used by early x86 (in/out)	Used by most modern processors; caching of device registers must be disabled

6.1.4 Direct Memory Access (DMA)

Without DMA the CPU must copy each byte from the controller (programmed I/O), wasting CPU time. A **DMA controller** can transfer data directly between the device controller and main memory **without CPU intervention**. Steps of a DMA transfer (disk read):

- **1.** The CPU programs the DMA controller with the memory address, byte count and direction; then tells the disk controller to read a block into its buffer.
- **2.** The DMA controller issues a read request to the disk controller over the bus.
- **3.** The disk controller writes the word into memory; the DMA increments the address and decrements the count.
- **4.** The disk controller sends an acknowledgement to the DMA controller; steps 2–4 repeat until the count reaches zero.
- **5.** When the transfer is complete, the DMA controller **interrupts the CPU**.

Note: DMA may work in cycle stealing mode (one word at a time, competing with the CPU for the bus) or burst mode (a block at a time).

6.2 Principles of I/O Software

6.2.1 Goals of I/O Software

Goal	Explanation
Device independence	Programs should be able to access any I/O device without specifying it in advance (a program reading a file works on a floppy, hard disk or USB).
Uniform naming	Files and devices named by a string or integer that does not depend on the device (UNIX: /dev/sda mounted anywhere in the tree).
Error handling	Errors should be handled as close to the hardware as possible (controller first, then driver); upper layers are told only if the lower layer cannot fix it.
Synchronous vs asynchronous	Most physical I/O is asynchronous (interrupt-driven), but user programs are easier to write with blocking (synchronous) calls; the OS makes I/O appear blocking.
Buffering	Data from a device cannot always be stored directly at the destination; buffers decouple device speed from consumer speed.
Sharable vs dedicated devices	Disks are sharable; printers, tape drives are dedicated — the OS must handle both (spooling).

6.2.2 Programmed I/O

The CPU does all the work: it issues the I/O command, then **polls (busy-waits)** the device status until it is ready, copying data one byte or word at a time. Simple, but ties up the CPU 100% during the whole I/O.

```
/* Printing a string with programmed I/O */
copy_from_user(buffer, p, count);          /* p = kernel buffer */
for (i = 0; i < count; i++) {
    while (*printer_status_reg != READY) ; /* busy wait */
    *printer_data_reg = p[i];              /* output one char */
}
return_to_user();
```

6.2.3 Interrupt-Driven I/O

The CPU starts the I/O and then **schedules another process** instead of waiting. When the device is ready it raises an **interrupt**; the interrupt service routine outputs the next character or wakes the blocked process. Efficient CPU use, but an interrupt occurs for every character (byte), and each interrupt costs time.

6.2.4 I/O Using DMA

The DMA controller feeds the characters to the device without bothering the CPU. The CPU is interrupted **once per buffer** (block) rather than once per character, so the number of interrupts is greatly reduced. It is the method of choice for disks and network cards; its drawback is that the DMA controller is slower than the CPU for a fast CPU-bound system.

Method	CPU involvement	Interrupts	Best for
Programmed I/O	Full — CPU polls and copies data	None	Simple/slow devices, embedded systems
Interrupt-driven I/O	Partial — CPU free between interrupts	One per byte/word	Keyboards, mice, low-volume devices
DMA	Minimal — only setup and completion	One per block	Disks, network, sound cards

6.3 I/O Software Layers

I/O software is typically organised in four layers. Each layer has a well-defined function and interface to its neighbours.

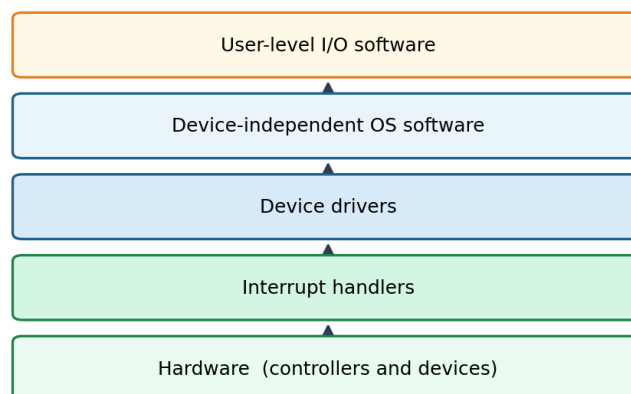


Figure 6.1 — Layers of the I/O software system

6.3.1 Interrupt Handlers

Interrupts are best hidden deep inside the OS. The driver that started an I/O operation blocks until the operation completes and the interrupt occurs. The interrupt handler does whatever is necessary: it saves registers, sets up a context, acknowledges the interrupt controller, runs the service procedure, unblocks the waiting driver (e.g., with up on a semaphore or signal on a condition variable) and returns to the scheduler.

6.3.2 Device Drivers

Each I/O device needs **device-specific code** for controlling it — the **device driver**, usually written by the device manufacturer and supplied with the device. A driver handles one device type (or class). It accepts abstract requests (read block n) from the device-independent layer and translates them into concrete commands for the controller registers; it checks device status, handles errors and then blocks until an interrupt. Drivers normally run in kernel mode and must be **reentrant** and handle hot-plugging (USB).

6.3.3 Device-Independent I/O Software

Function	Explanation
Uniform interfacing for drivers	Standard interface between the OS and all drivers
Buffering	Kernel buffers smooth out speed mismatches (single, double, circular buffering)
Error reporting	Programming errors (invalid buffer) vs real I/O errors (bad disk block)
Allocating and releasing dedicated devices	Open/close or queueing for dedicated devices like printers
Providing a device-independent block size	Hides sector-size differences from higher layers
Naming and protection	Mapping symbolic device names to drivers; access permissions

6.3.4 User-Space I/O Software

A small part of the I/O software consists of **library procedures** linked with user programs (e.g., `write()` in `libc`, `printf`, `fopen`) which run in user space and make system calls. **Spooling** is another user-space mechanism: a special process (**daemon**) and a **spooling directory** allow a dedicated device such as a printer to be shared — users never open the printer directly; they place files in the spool directory and the daemon prints them one by one.

6.4 Disk

6.4.1 Disk Hardware

A magnetic disk consists of one or more **platters** coated with magnetic material, rotating (5400, 7200 or 15000 rpm) on a spindle. Each surface has a read/write **head** mounted on an arm. The surface is divided into concentric **tracks**; each track is divided into **sectors** (usually 512 bytes or 4 KB). All tracks at the same arm position on all platters form a **cylinder**. Modern disks use **zone bit recording** (more sectors on outer tracks) and logical block addressing (LBA). Solid-state drives (SSD) have no moving parts and no seek time.

6.4.2 Disk Access Time

The time to read or write a disk block is determined by three factors:

Component	Meaning	Typical value
Seek time	Time to move the arm to the correct cylinder	4 – 10 ms (average)
Rotational delay (latency)	Time for the desired sector to rotate under the head; average = $\frac{1}{2}$ rotation	7200 rpm $\rightarrow$ 4.17 ms
Transfer time	Time to read/write the data as it passes under the head	$\approx$ 0.04 ms per 4 KB

Disk access time = Seek time + Rotational delay + Transfer time. Seek time dominates, so reducing arm movement is the key to disk scheduling.

Example — Disk access time

A disk rotates at 7200 rpm, has an average seek time of 9 ms and a transfer rate of 100 MB/s. Find the average time to read a 4 KB block.

Solution

One rotation = $60/7200$ s = 8.33 ms $\rightarrow$ average rotational delay = $8.33/2 = 4.17$ ms.

Transfer time = 4 KB / 100 MB/s = 0.04 ms.

Access time = $9 + 4.17 + 0.04 \approx 13.2$ ms.

6.5 Disk Scheduling Algorithms

When several requests are pending the OS decides the **order** in which to service them so as to minimise total seek time (arm movement). Numerical example used for all algorithms:

Example — Request queue

Disk with 200 cylinders (0–199). Request queue (cylinders): **98, 183, 37, 122, 14, 124, 65, 67**. The head is initially at cylinder **53**. Calculate the total head movement for each algorithm.

(1) FCFS Scheduling

Requests are served in arrival order. Fair, no starvation, but poor performance because the arm swings wildly.

Solution — FCFS

$53 \rightarrow 98 \rightarrow 183 \rightarrow 37 \rightarrow 122 \rightarrow 14 \rightarrow 124 \rightarrow 65 \rightarrow 67$

Movement = $45 + 85 + 146 + 85 + 108 + 110 + 59 + 2 = \mathbf{640}$ cylinders.

(2) SSTF (Shortest Seek Time First)

Serves the request closest to the current head position. Similar to SJF; reduces movement but can cause **starvation** of distant requests when new nearby ones keep arriving; not optimal overall.

Solution — SSTF

$53 \rightarrow 65 \rightarrow 67 \rightarrow 37 \rightarrow 14 \rightarrow 98 \rightarrow 122 \rightarrow 124 \rightarrow 183$

Movement = $12 + 2 + 30 + 23 + 84 + 24 + 2 + 59 = \mathbf{236}$ cylinders.

(3) SCAN (Elevator Algorithm)

The arm starts at one end and moves toward the other end, servicing requests on the way; at the other end the direction reverses. Behaves like an elevator. Good throughput, no starvation, but requests just behind the arm wait for nearly two sweeps.

Solution — SCAN (head moving towards 0)

$53 \rightarrow 37 \rightarrow 14 \rightarrow 0 \rightarrow 65 \rightarrow 67 \rightarrow 98 \rightarrow 122 \rightarrow 124 \rightarrow 183$

Movement = $53 + 183 = \mathbf{236}$ cylinders (53 down to 0, then 0 up to 183).

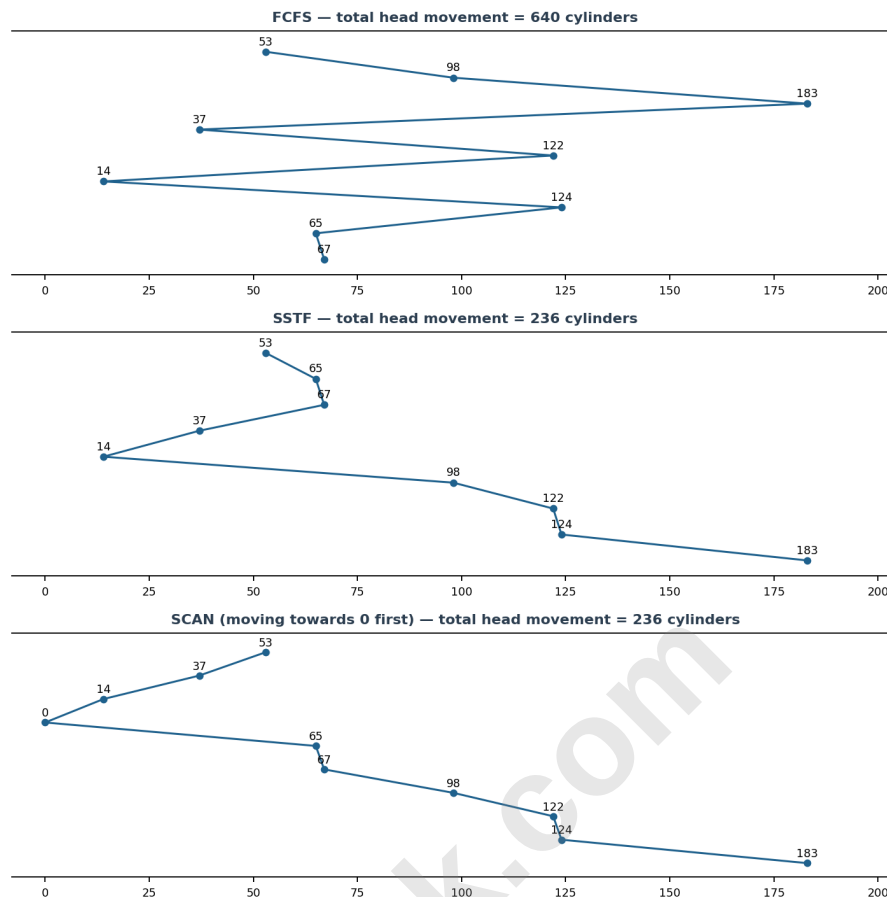


Figure 6.2 — Head movement for FCFS, SSTF and SCAN

(4) C-SCAN (Circular SCAN)

Like SCAN but the head services requests in **one direction only**; on reaching the end it returns immediately to the other end **without servicing**, then starts again. It gives a more uniform wait time.

Solution — C-SCAN (moving towards 199)

53 → 65 → 67 → 98 → 122 → 124 → 183 → 199 → (return) 0 → 14 → 37

Movement = $(199 - 53) + 199 + 37 = 146 + 199 + 37 = 382$ cylinders (including the return sweep).

(5) LOOK and C-LOOK Scheduling (“Lock” scheduling)

Practical versions of SCAN and C-SCAN: the arm goes only as far as the **last request** in each direction, then reverses (LOOK) or jumps back (C-LOOK) — it does not travel to the physical end of the disk.

Solution — LOOK (towards 0 first)

53 → 37 → 14 → 65 → 67 → 98 → 122 → 124 → 183

Movement = (53 - 14) + (183 - 14) = 39 + 169 = **208 cylinders**.

C-LOOK (towards 199): 53 → ... → 183, jump to 14, → 37 : 130 + 169 + 23 = **322 cylinders**.

Algorithm	Total movement	Starvation?	Remark
FCFS	640	No	Fair but slow
SSTF	236	Possible	Good for light load
SCAN	236	No	Elevator; good for heavy load
C-SCAN	382 (incl. return)	No	Uniform waiting time
LOOK	208	No	Best here — no travel to the disk end
C-LOOK	322	No	Uniform + efficient

Note: Rotational latency can also be scheduled (e.g., serve sectors in rotational order), and SSDs need no seek-based scheduling — a simple FIFO/NOOP scheduler is often used.

Key Takeaways

- ✓ I/O devices are **block** or **character** devices, controlled through **device controllers**; registers are accessed through I/O ports or **memory-mapped I/O**.
- ✓ **DMA** moves data between device and memory without the CPU and interrupts only once per block.
- ✓ I/O software methods: **programmed I/O** (polling), **interrupt-driven I/O** and **DMA**.
- ✓ Layers: interrupt handlers → device drivers → device-independent software → user-space I/O software.
- ✓ Disk access time = seek + rotational delay + transfer time; seek time dominates.
- ✓ Disk scheduling: FCFS (640), SSTF (236), SCAN (236), C-SCAN (382), LOOK (208) for the standard example.

— End of Unit 6 —