I/O Management and Disk Scheduling

1. I/O Devices

Categories of I/O Devices

Operating systems must manage a wide variety of I/O devices, which can be broadly classified into three main categories based on their function and the nature of their interaction with the system:

Human Readable Devices

These devices facilitate communication between the computer and human users. They are designed for input or output that is understandable by people. Examples include:

- Printers
- Video displays (monitors)
- Keyboards
- Mice

Machine Readable Devices

These devices enable communication between the computer and other electronic equipment or systems. They are typically used for automated data transfer and control. Examples include:

- Disk drives (hard disks, SSDs)
- USB keys
- Sensors
- Controllers
- Actuators

Communication Devices

These are used to connect the computer to remote systems, allowing data exchange over networks or communication lines. Examples include:

- Digital line drivers
- Modems
- Ethernet and WiFi adapters

Differences in I/O Devices

I/O devices vary significantly in several key aspects, making OS design for I/O management complex:

- Data Rate: Devices operate at vastly different speeds. For example, a keyboard has a much lower data transfer rate compared to a hard disk or network interface.
- Application: The same type of device can be used for different purposes. For example, a disk may store files or act as virtual memory, each requiring different management strategies.
- Complexity of Control: Some devices, like printers, have simple control requirements, while others, such as disks, are much more complex and require sophisticated controllers.
- Unit of Transfer: Data can be transferred as single bytes/characters (e.g., terminals) or in larger blocks (e.g., disks).
- Data Representation: Devices may use different encoding schemes, character sets, or parity conventions.
- Error Conditions: The type, reporting, and handling of errors differ widely among devices.

2. Organization of the I/O Function

Techniques for Performing I/O

There are three primary techniques used for I/O operations in modern systems:
Programmed I/O

- The CPU issues an I/O command and then waits (busy-waits) for the operation to complete.
- Simple to implement but inefficient, as the CPU is tied up during the entire operation.

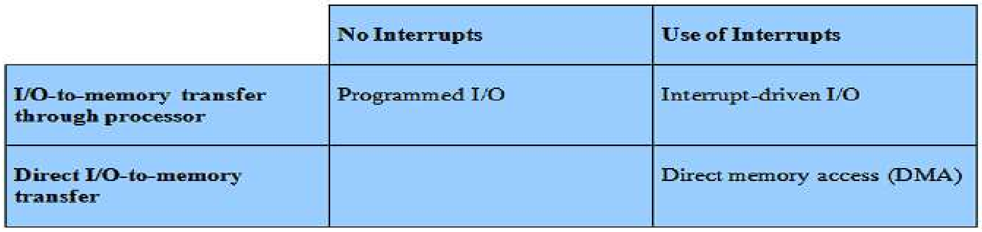
Interrupt-Driven I/O

- The CPU initiates the I/O operation and continues with other tasks.
- The device interrupts the CPU when it is ready, allowing more efficient multitasking.

Direct Memory Access (DMA)

- A DMA controller handles data transfer directly between memory and the I/O device, bypassing the CPU except for initiation and completion.
- This significantly reduces CPU overhead and increases system efficiency, especially for large data transfers.

Table 11.1 I/O Techniques



	No Interrupts	**Use of Interrupts**
I/O-to-memory transfer through processor	Programmed I/O	Interrupt-driven I/O
Direct I/O-to-memory transfer		Direct memory access (DMA)

Evolution of the I/O Function

The management of I/O in computer systems has evolved through several stages:

1. Processor Directly Controls Device: Early systems had the CPU directly manage all aspects of I/O.
2. Controller or I/O Module Added: Offloads some device-specific details from the CPU; CPU still uses programmed I/O.
3. Controller with Interrupts: Devices can signal the CPU when attention is needed, improving efficiency.
4. Direct Memory Access (DMA): Data moves between devices and memory without CPU intervention.
5. I/O Module as a Separate Processor: Specialized I/O processors handle complex I/O tasks, further freeing the CPU

DMA - Direct Memory Address

Direct Memory Access (DMA) is a critical technology that enables peripherals to transfer data directly to/from memory without continuous CPU involvement, significantly improving system efficiency.

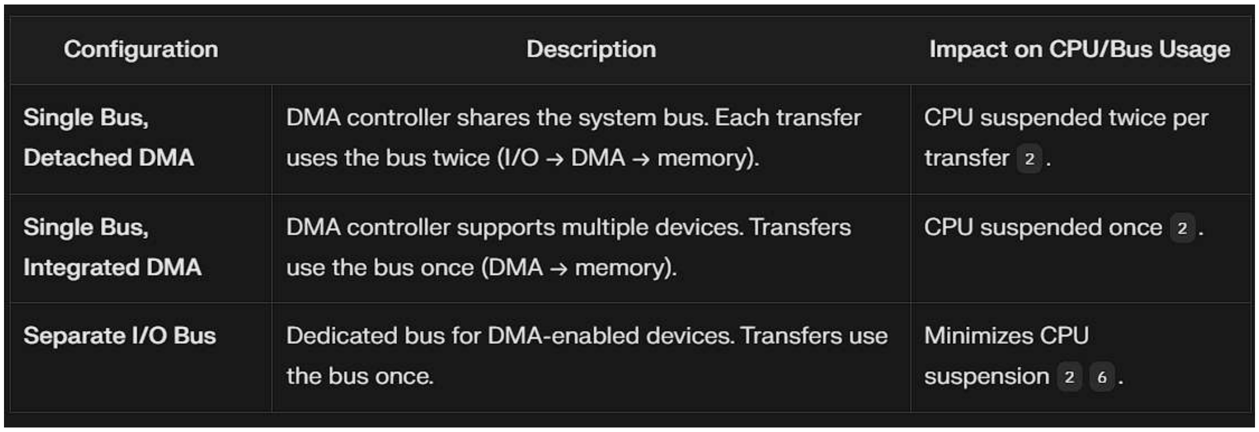
How DMA Works

- The CPU delegates I/O operations to the DMA controller, which manages data transfers independently.
- Steps include:
    1. Initiation: The peripheral sends a DMA request (DREQ) to the controller.
    2. Bus Control: The DMA controller gains bus access via signals (e.g., HOLD/HLDA in Intel 8237A).
    3. Data Transfer: Data moves directly between the device and memory.
    4. Completion: The controller interrupts the CPU once done.

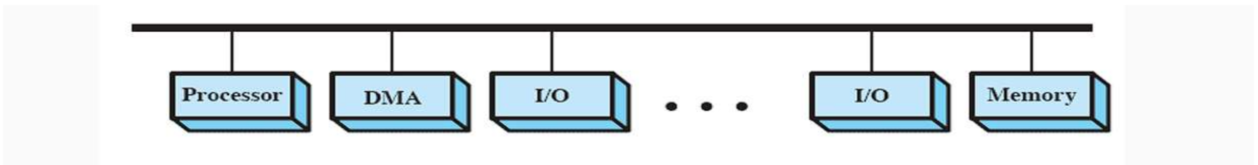
Advantages Over Programmed/Interrupt-Driven I/O

- Eliminates CPU busy-waiting, freeing it for other tasks.
- Ideal for high-speed devices (e.g., disks, network cards)

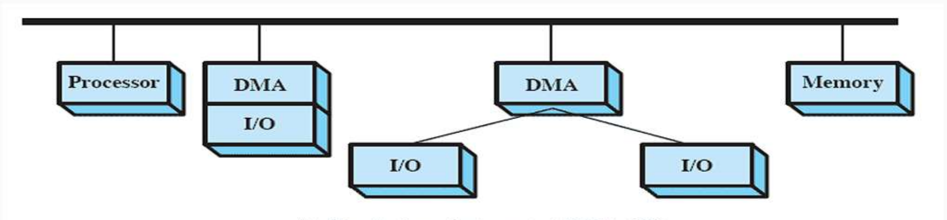
DMA Configurations



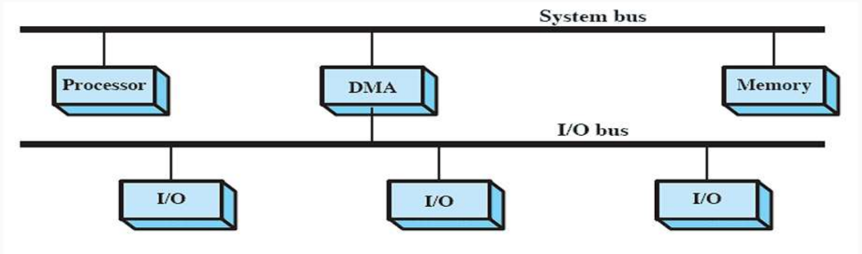
Configuration	Description	Impact on CPU/Bus Usage
Single Bus, Detached DMA	DMA controller shares the system bus. Each transfer uses the bus twice (I/O → DMA → memory).	CPU suspended twice per transfer [2].
Single Bus, Integrated DMA	DMA controller supports multiple devices. Transfers use the bus once (DMA → memory).	CPU suspended once [2].
Separate I/O Bus	Dedicated bus for DMA-enabled devices. Transfers use the bus once.	Minimizes CPU suspension [2] [6].



(a) Single-bus, detached DMA



(b) Single-bus, Integrated DMA-I/O



(c) I/O bus

3. Operating System Design Issues

Efficiency

- **Multiprogramming:** Allows processes to wait for I/O while others execute, improving CPU utilization214.
- **Buffering:** Mitigates speed mismatches between I/O devices (e.g., HDDs at ~200 IOPS) and CPU/memory (billions of operations/sec)39.
- **Swapping Trade-off:** While swapping brings ready processes into memory, it itself is an I/O operation, requiring careful scheduling14.

Efficiency is achieved through overlapping I/O with computation (multiprogramming, DMA) and smoothing data flow (buffering).

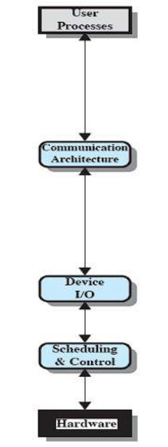
Generality

- Uniform Handling: OS abstracts device-specific details into hierarchical layers (logical I/O → device I/O → scheduling/control)212.
- Device Independence: Standard interfaces (e.g., `read`, `write`) simplify application development

Generality relies on abstraction layers (device drivers, uniform APIs) and hierarchical design to isolate hardware specifics.

Hierarchical design

- A hierarchical philosophy leads to organizing an OS into layers
- Each layer relies on the next lower layer to perform more primitive functions
- It provides services to the next higher layer
- Changes in one layer should not require changes in other layers



(b) Communications port

Local peripheral device

- Logical I/O: – Deals with the device as a logical resource (device identifier and simple commands)
- Device I/O: – Converts requested operations into sequence of I/O instructions, channel commands & controller orders
- Scheduling and Control – Performs actual queuing and control operations

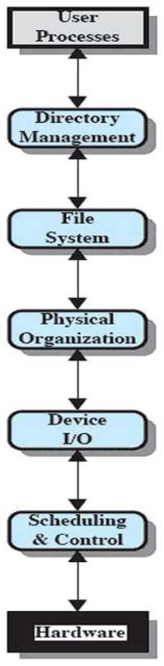
Communications Port

- Similar to previous but the logical I/O module is replaced by a communications architecture
- This consists of a number of layers.

An example is TCP/IP

File System

- **Directory management** Concerned with user operations affecting files
- **File System** Logical structure and operations
- **Physical organisation** Converts logical names to physical addresses

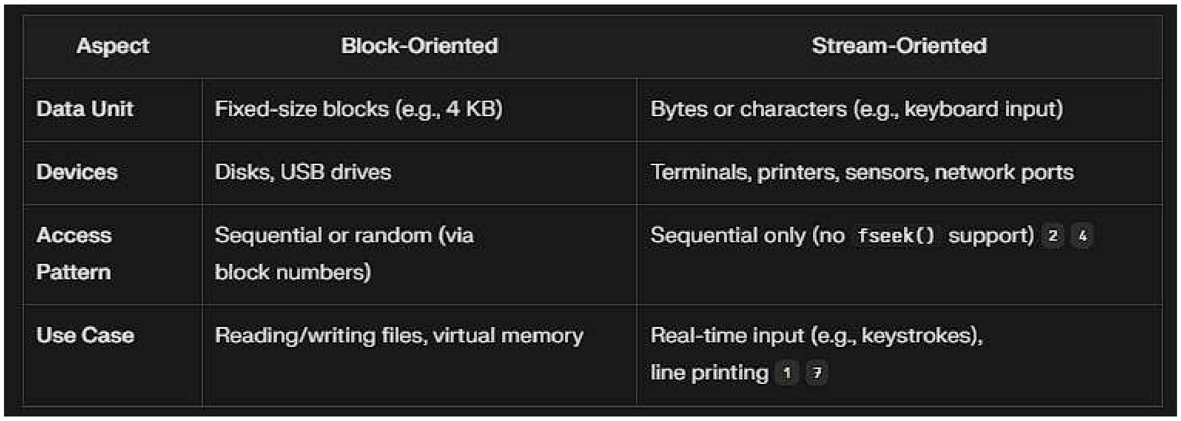


(c) File system

4. I/O buffering

I/O buffering is a critical OS mechanism that optimizes data transfer between devices and processes by mitigating speed mismatches and reducing wait times.

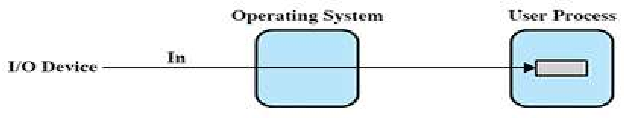
Block-Oriented vs. Stream-Oriented Buffering



Aspect	Block-Oriented	Stream-Oriented
Data Unit	Fixed-size blocks (e.g., 4 KB)	Bytes or characters (e.g., keyboard input)
Devices	Disks, USB drives	Terminals, printers, sensors, network ports
Access Pattern	Sequential or random (via block numbers)	Sequential only (no `fseek()` support) [2] [4]
Use Case	Reading/writing files, virtual memory	Real-time input (e.g., keystrokes), line printing [1] [7]

1. No Buffering

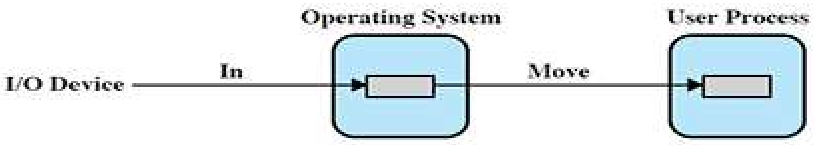
- Direct Access: OS interacts with devices immediately for each I/O request.
- Drawback: High CPU wait time and risk of deadlock if processes are swapped out mid-operation.



(a) No buffering

2. Single Buffer

- Mechanism: One buffer temporarily holds data during transfers.
    - Block Devices: Data is read into the buffer, then moved to user space. The next block is prefetched ("read ahead") while the current block is processed.
    - Stream Devices: Handles data line-by-line (e.g., terminal input) or byte-by-byte (e.g., sensors).
- Limitation: Processes may still wait for buffer refills.



(b) Single buffering

Block-Oriented Single Buffering

Used for devices like disks and USB drives where data is read/written in fixed-size blocks.

How It Works

1. **Input Operation**:
    - Data is read from the device into the buffer.
    - Once the buffer is filled, the block is transferred to **user space**.
    - Meanwhile, the next block is read into the buffer (**read-ahead**).
2. **Output Operation**:
    - The process fills the buffer with data.
    - The OS writes the buffer contents to the device.

    - The process can continue working while the write completes.

Advantages

- **Reduces waiting time** by overlapping I/O with computation.
- **Read-ahead (Anticipated Input)**:
    - Assumes sequential access (common in file reads).
    - Pre-fetches the next block while the current one is being processed.

Disadvantages

- **No overlap if process needs data immediately** (must wait for buffer fill).
- **Wasted cycles** if access pattern is random (read-ahead may be useless).

Stream-Oriented Single Buffering

Used for character-based devices like terminals, printers, and sensors.

Two Modes of Operation

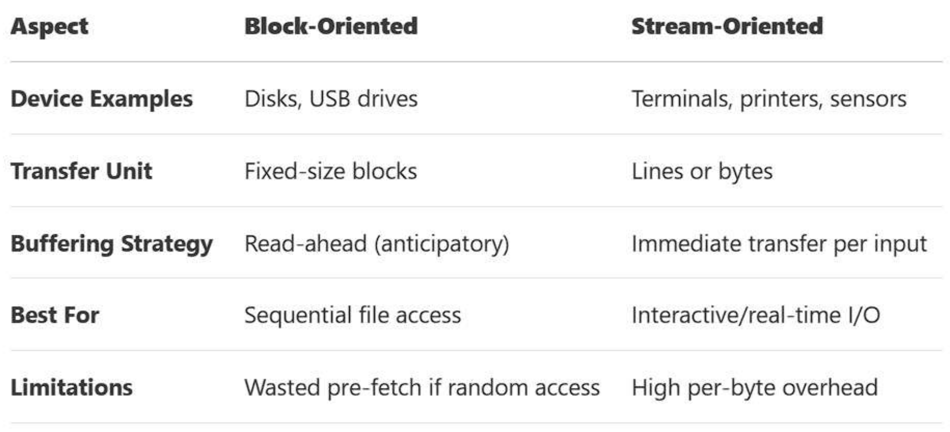
1. **Line-at-a-Time**
    - Used in **terminals and line printers**.
    - Buffer collects data until a **carriage return (Enter key)** is encountered.
    - Example: Typing in a command shell—input is sent only after pressing Enter.
2. **Byte-at-a-Time**
    - Used for **interactive devices** like keyboards, mice, and sensors.
    - Each keystroke or sensor reading triggers an immediate transfer.
    - Example: Real-time controller inputs where every byte matters.

Advantages

- **Efficient for interactive I/O** (avoids waiting for full blocks).
- **Low latency** for byte-stream devices.

Disadvantages

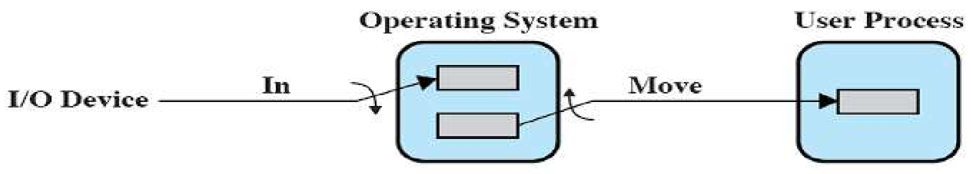
- **High overhead** if used for block devices (inefficient for disk I/O).
- **No parallelism—**process must wait for each byte/line to transfer.



Aspect	Block-Oriented	Stream-Oriented
Device Examples	Disks, USB drives	Terminals, printers, sensors
Transfer Unit	Fixed-size blocks	Lines or bytes
Buffering Strategy	Read-ahead (anticipatory)	Immediate transfer per input
Best For	Sequential file access	Interactive/real-time I/O
Limitations	Wasted pre-fetch if random access	High per-byte overhead

3. Double Buffer

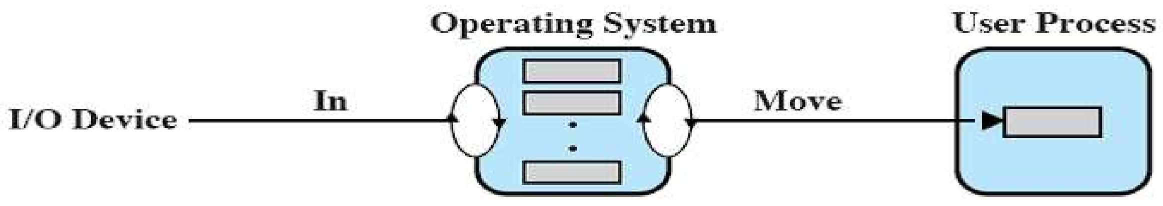
- Mechanism: Two buffers alternate between I/O operations and process access.
  - Block Devices: Enables continuous data flow (e.g., video streaming).
  - Stream Devices: Reduces wait time for line-at-a-time operations but offers no benefit for byte-at-a-time.
- Drawback: Increased complexity; ineffective for rapid I/O bursts.



(c) Double buffering

4. Circular Buffer

- Mechanism: Multiple buffers arranged in a ring to handle bursty I/O (e.g., network traffic).
- Use Case: Solves limitations of double buffering by allowing concurrent read/write across buffers



(d) Circular buffering

Advantages of Buffering

- **Efficiency:** Reduces CPU idle time by overlapping I/O with computation.
- **Deadlock Prevention:** Keeps critical pages in memory during I/O.
- **Smoothing:** Balances speed mismatches between fast CPUs and slow devices.

Limitations

But with enough I/O demand eventually all buffers become full and advantage is lost

- **Memory Overhead:** Buffers consume RAM, which can be wasteful for unpredictable data sizes.
- **Buffer Overflow:** Excessive demand fills all buffers, negating benefits.
- **Latency:** Delays between data write and processing in some scenarios

When there is a variety of I/O and process activities to service, buffering can increase the efficiency of the OS and the performance of individual processes.

5. Disk Scheduling

Goal: Efficient use of Disk fast access to disk and large disk BW

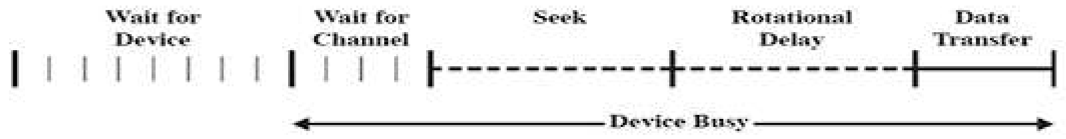


Figure 11.6 Timing of a Disk I/O Transfer

Positioning the Read/Write Heads

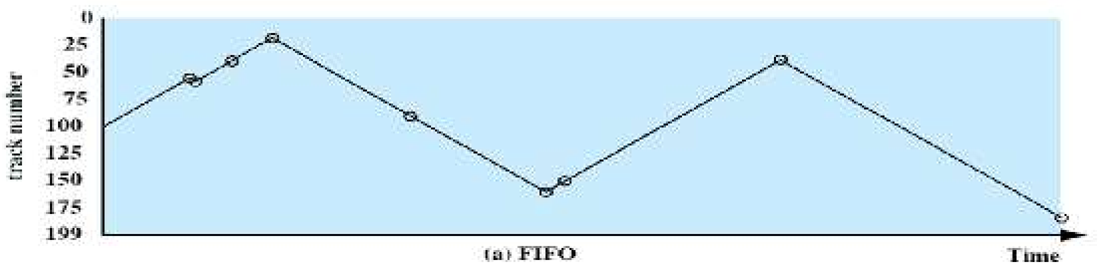
- When the disk drive is operating, the disk is rotating at constant speed
- Track selection involves moving the head in a movable-head system or electronically selecting one head on a fixed-head system

To optimize disk performance, operating systems use scheduling algorithms to minimize access time, which comprises:

- **Seek Time:** Time to move the disk arm to the target track.
- **Rotational Latency:** Time for the disk to rotate the desired sector under the head.
- **Transfer Time:** Time to read/write data.

1. FIFO (FCFS)

- Processes requests sequentially in arrival order.
- Fair to all processes but high seek time.
- Approximates random scheduling with many requests.



2. Priority-Based

- Prioritizes short jobs (e.g., interactive tasks) for faster response.
- Ignores disk efficiency; risks starvation for long jobs.
- Unsuitable for databases due to inconsistent throughput.

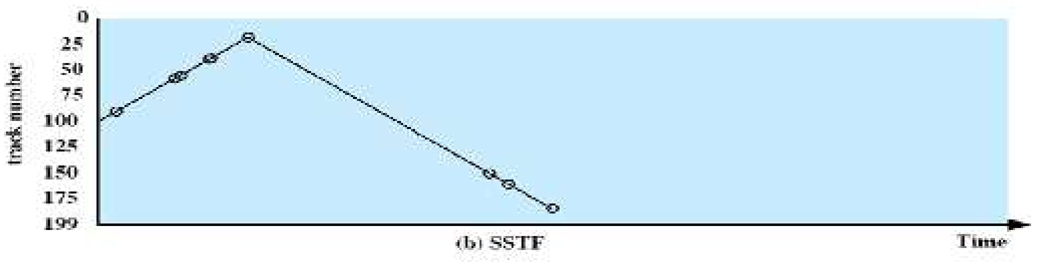
3. LIFO

- Serves most recent requests first, minimizing arm movement.

- Efficient for transaction systems but risks starvation for older requests.

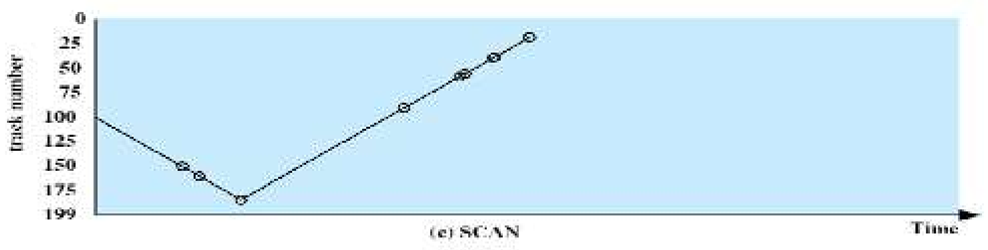
4. SSTF (Shortest Seek Time First)

- Selects nearest request to reduce seek time.
- High throughput but starvation risk for distant tracks.



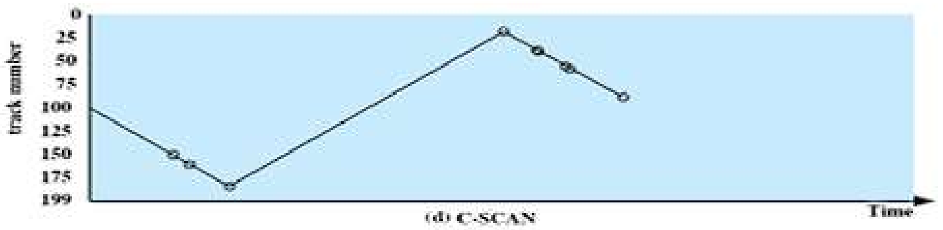
5. SCAN (Elevator Algorithm)

- Arm moves in one direction, servicing requests until end, then reverses.
- Balances fairness and efficiency but delays edge-track requests.



6. C-SCAN

- Services requests in one direction only; jumps to start after reaching end.
- Uniform wait times but wastes time resetting without servicing return path.



(d) C-SCAN

7. N-step-SCAN

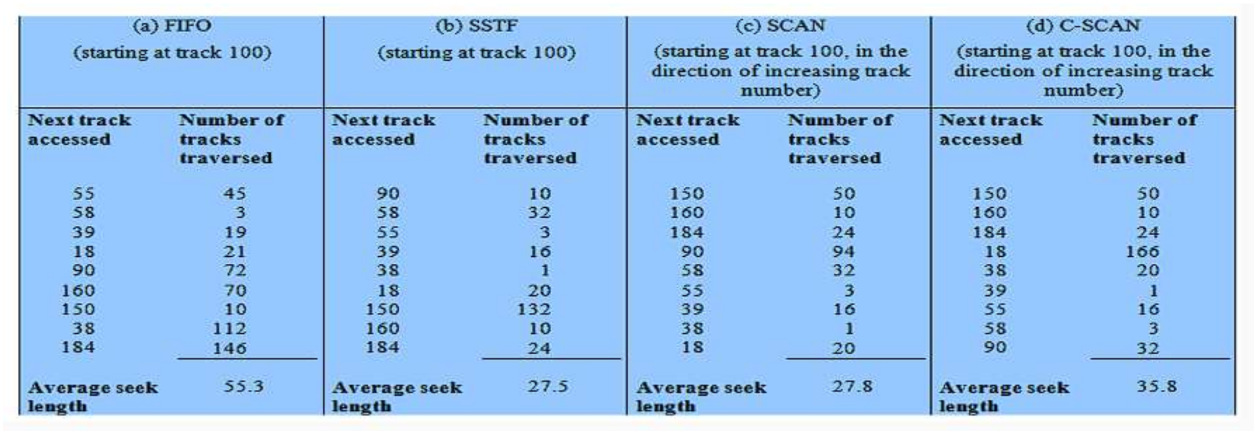
- Divides request queue into subqueues (size *N*).
- Processes each subqueue with SCAN, preventing indefinite postponement.

8. FSCAN

- Uses two queues: one for current scan, another for new requests.
- Eliminates starvation by deferring new requests until the current batch completes.

  All new requests are put into the other queue. Service of new requests is deferred until all of the old requests have been processed.

Comparison of Disk Scheduling Algorithms



(a) FIFO (starting at track 100)		(b) SSTF (starting at track 100)		(c) SCAN (starting at track 100, in the direction of increasing track number)		(d) C-SCAN (starting at track 100, in the direction of increasing track number)	
Next track accessed	**Number of tracks traversed**	**Next track accessed**	**Number of tracks traversed**	**Next track accessed**	**Number of tracks traversed**	**Next track accessed**	**Number of tracks traversed**
55	45	90	10	150	50	150	50
58	3	58	32	160	10	160	10
39	19	55	3	184	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	146	184	24	18	20	90	32
Average seek length	55.3	**Average seek length**	27.5	**Average seek length**	27.8	**Average seek length**	35.8

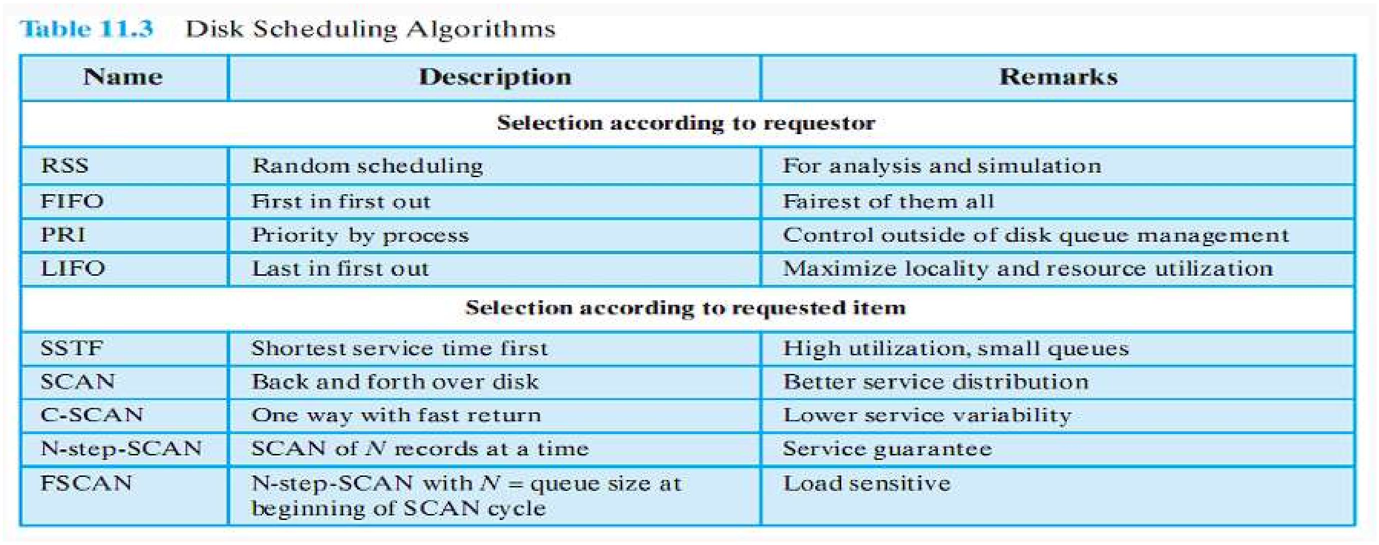


Table 11.3 Disk Scheduling Algorithms

Name	Description	Remarks
Selection according to requestor		
RSS	Random scheduling	For analysis and simulation
FIFO	First in first out	Fairest of them all
PRI	Priority by process	Control outside of disk queue management
LIFO	Last in first out	Maximize locality and resource utilization
Selection according to requested item		
SSTF	Shortest service time first	High utilization, small queues
SCAN	Back and forth over disk	Better service distribution
C-SCAN	One way with fast return	Lower service variability
N-step-SCAN	SCAN of N records at a time	Service guarantee
FSCAN	N-step-SCAN with N = queue size at beginning of SCAN cycle	Load sensitive

Key Trade-offs:

- **Efficiency vs. Fairness:** SSTF optimizes seek time but sacrifices fairness; SCAN/FSCAN balance both.
- **Starvation Mitigation:** N-step-SCAN and FSCAN address starvation in dynamic workloads.
- **Use Case**: LIFO suits transaction systems; C-SCAN ideal for uniform service.
- SSTF is optimal for minimizing seek time but unsuitable for real-time systems due to starvation.
- SCAN balances efficiency and fairness, ideal for general-purpose systems.
- C-SCAN ensures uniform wait times but incurs higher overhead from backtracking.
- FCFS is rarely used in practice due to poor performance.