Chapter 11
I/O Management and Disk Scheduling

(based on original slides by Pearson)

Categories of I/O Devices

- Human readable
  - Used to communicate with the user
  - Printers and terminals
- Machine readable
  - Used to communicate with electronic equipment
  - Disk drives, USB keys, Sensors, Controllers, Actuators
- Communication
  - Used to communicate with remote devices
  - Digital line drivers, modems

Differences in I/O Devices

- Data rate
  - May be differences of several orders of magnitude between the data transfer rates

I/O Device Data Rates

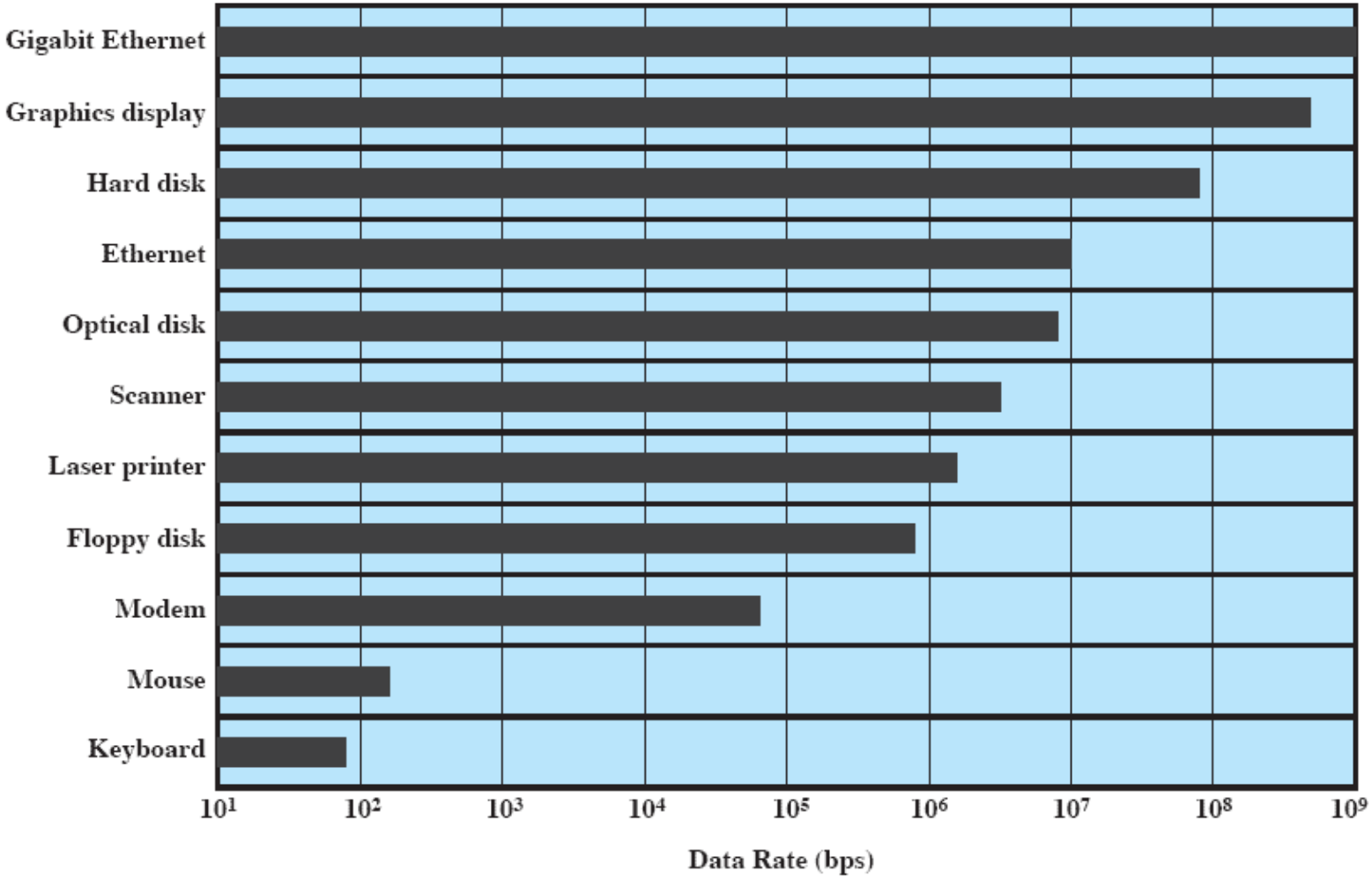


Figure 11.1 Typical I/O Device Data Rates

Differences in I/O Devices

- Application
  - ***The use of the device affects the management software***
  - Example:
    - Disk used to store files requires file management software
    - Disk used to store virtual memory pages needs special hardware and software to support it
    - Terminal used by system administrator may have a higher priority than the regular user's one

Differences in I/O Devices

- Complexity of control
  - Simple one line on/off
  - Time dependent interaction: duty cycle of stepper motors
  - Complex, protocol-based interaction: most communication devices
- Unit of transfer
  - Data may be transferred as a stream of bytes for a terminal or in larger blocks for a disk

Differences in I/O Devices

- Data representation
  - Encoding schemes, parity
  - Example old SMS encoding on cell phones
- Error conditions
  - Devices respond to errors differently
    - Motor loses it's torque
    - CAN does simply not transmit

Performing I/O

- Programmed I/O
  - Process is busy-waiting for the operation to complete
- Interrupt-driven I/O
  - I/O command is issued
  - Processor continues executing instructions
- Direct Memory Access (DMA)
  - DMA module controls exchange of data between main memory and the I/O device
  - Processor interrupted only after entire block has been transferred

Relationship Among Techniques

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

- Processor directly controls a peripheral device
- Controller or I/O module is added
  - Processor uses programmed I/O without interrupts
  - Processor does not need to handle details of external devices

Evolution of the I/O Function

- Controller or I/O module with interrupts
  - Processor does not spend time waiting for an I/O operation to be performed
- Direct Memory Access
  - Blocks of data are moved into memory without involving the processor
  - Processor involved at beginning and end only

Evolution of the I/O Function

- I/O module is a separate processor
- I/O processor
  - I/O module has its own local memory
  - It is a computer in its own right

Direct Memory Address

- Processor delegates I/O operation to the DMA module
- DMA module transfers data directly to or from memory
- When transfer is complete, DMA module sends an interrupt signal to the processor

DMA

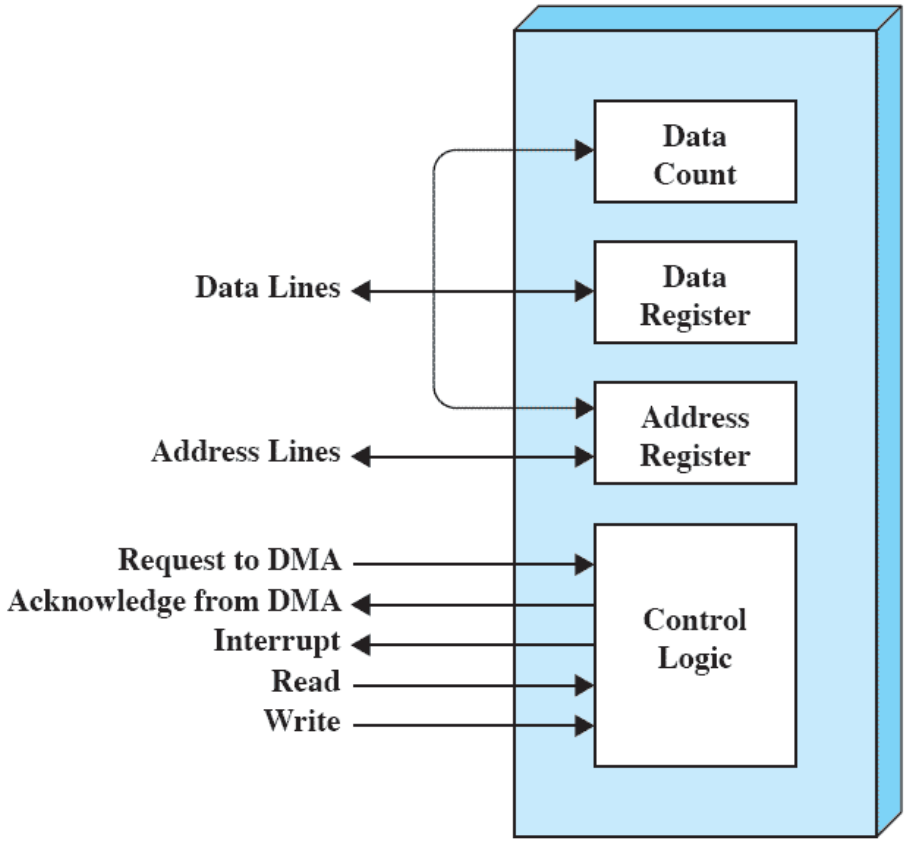
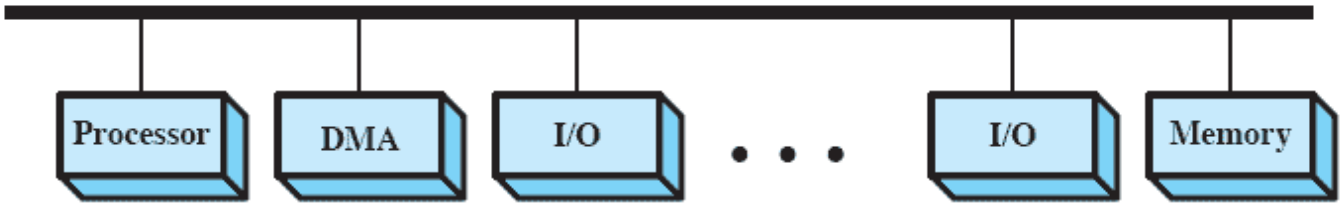


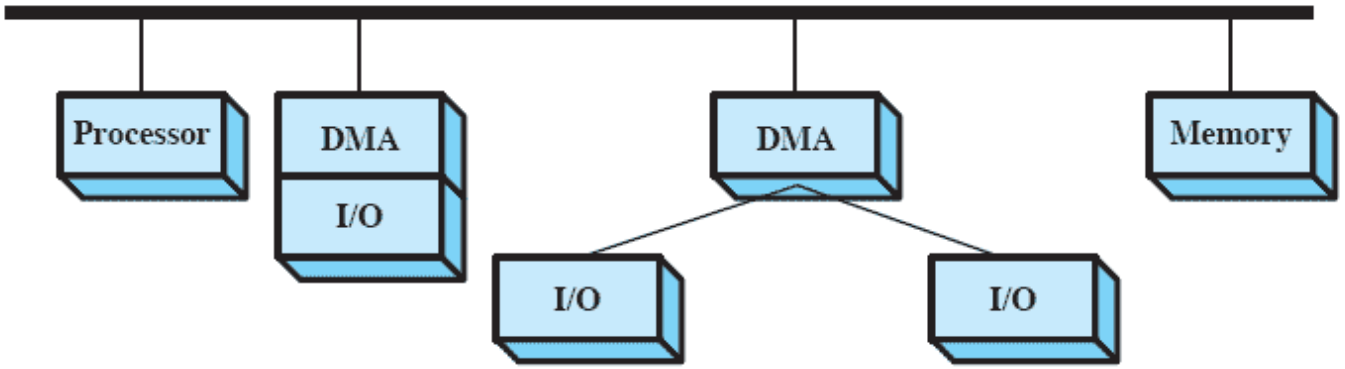
Figure 11.2 Typical DMA Block Diagram

DMA Configurations



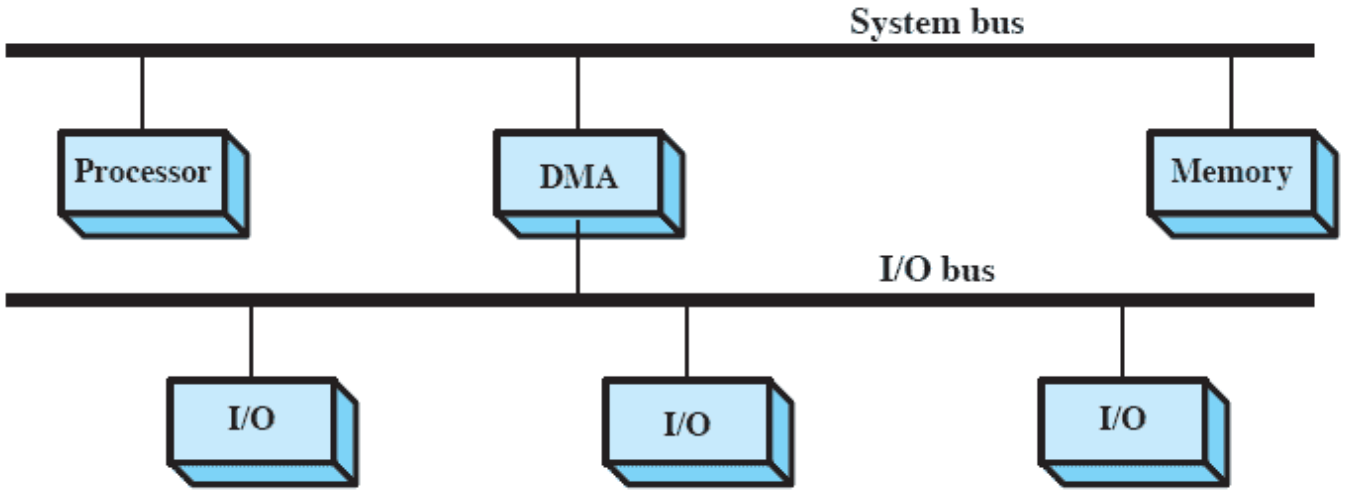
(a) Single-bus, detached DMA

DMA Configurations



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

DMA Configurations



(c) I/O bus

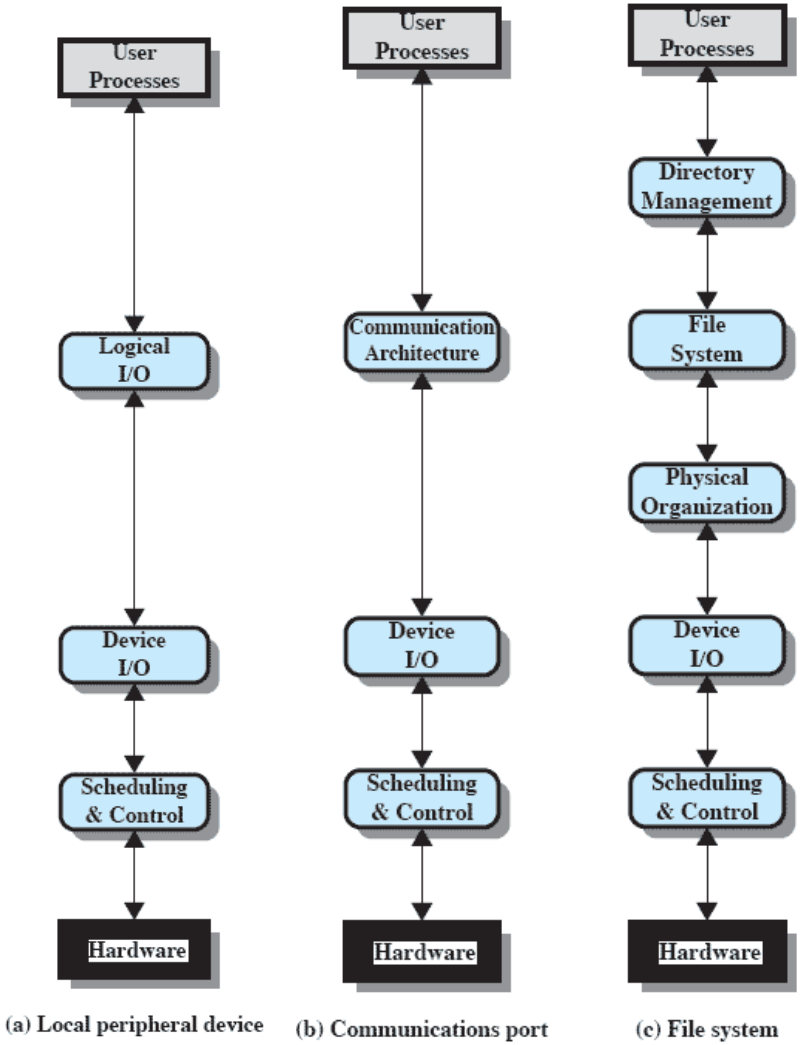
OS Design Issues wrt I/O

- Efficiency
  - Most I/O devices are extremely slow compared to main memory
  - Use of multiprogramming allows interleaving of I/O and processing
  - Even in the future: I/O will not keep up with processor throughputs
  - Swapping is used to bring in additional Ready processes; swapping is an I/O operation

OS Design Issues wrt I/O

- Generality
  - Desirable to handle all I/O devices in a uniform manner
  - Hide most of the details of device I/O in lower-level routines

Model of I/O Operation



(a) Local peripheral device (b) Communications port (c) File system

- Logical I/O: provides open, close, read, write
- Device I/O: converts operations&data into I/O instructions
- Sch&Control: queuing and scheduling of operations
- Directory management: organizes files, symbolic file names are converted
- File system: open, close, read, write
- Physical org: convert file addresses into locations in the disc

Logical Structure of I/O

- Logical I/O
  - treats device as logical resource with no concerns of details of control
  - manages I/O functions on behalf of user process
- Device I/O
  - convert requested operations into sequences of I/O instructions, channel commands, and controller orders
- Scheduling and Control
  - queuing and scheduling of I/O operations
  - handle interrupts, check I/O status,…

Logical Structure of I/O

For secondary storage devices with a file system:

- Directory Management
  - convert symbolic file names to identifiers that reference directly/indirectly the file
  - manage user add, delete, reorganize operations
- File System
  - logical structure of files and operations that can be specified by user
- Physical Organization
  - convert logical references to files/records to physical storage addresses

I/O Buffering

- Reasons for buffering
  - Processes must wait for I/O to complete before proceeding
  - Certain pages must remain in main memory during I/O
- Risk of deadlock:
  - Process calls an I/O routine and blocks
  - Process is swapped out
  - Process waits for I/O completion, but I/O waits for the processor to be swapped in to access the memory

I/O Buffering

- Block-oriented
  - Information is stored in fixed sized blocks
  - Transfers are made a block at a time
  - Used for disks and USB keys
- Stream-oriented
  - Transfer information as a stream of bytes
  - Used for terminals, printers, communication ports, mouse and other pointing devices, and most other devices that are not secondary storage
- The kernel (I/O routines) handle buffering

Single Buffer

- Operating system assigns a buffer in main memory for an I/O request
- Block-oriented
  - Input transfers made to buffer
  - Block moved to user space when needed
  - Another block is moved into the buffer
    - Read ahead

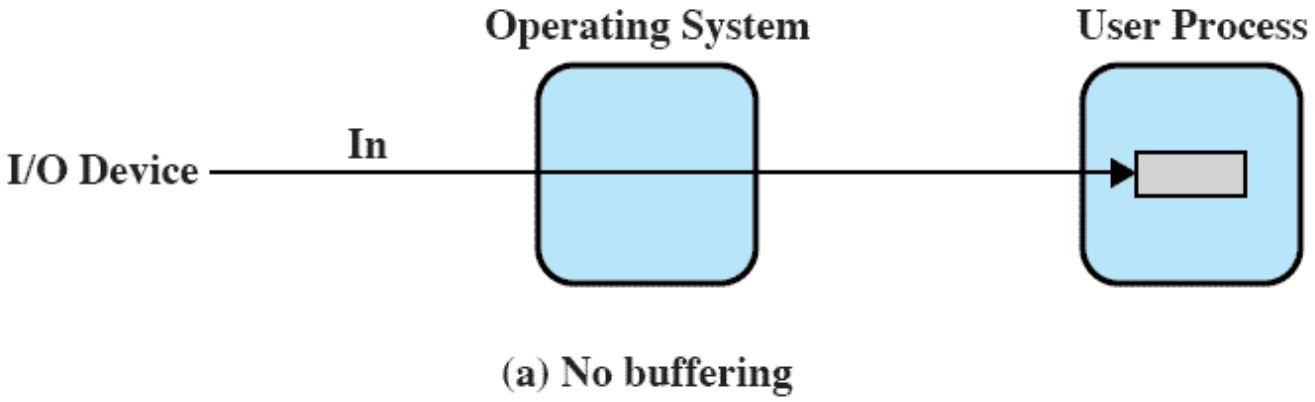
Single Buffer

- Block-oriented
  - User process can process one block of data while next block is read in
  - Swapping can occur since input is taking place in system memory, not user memory
  - Operating system keeps track of assignment of system buffers to user processes

Single Buffer

- Stream-oriented
  - Use a line at a time
  - User input from a terminal is one line at a time with carriage return signaling the end of the line
  - Output to the terminal is one line at a time
  - Flush() call in C

Single Buffer

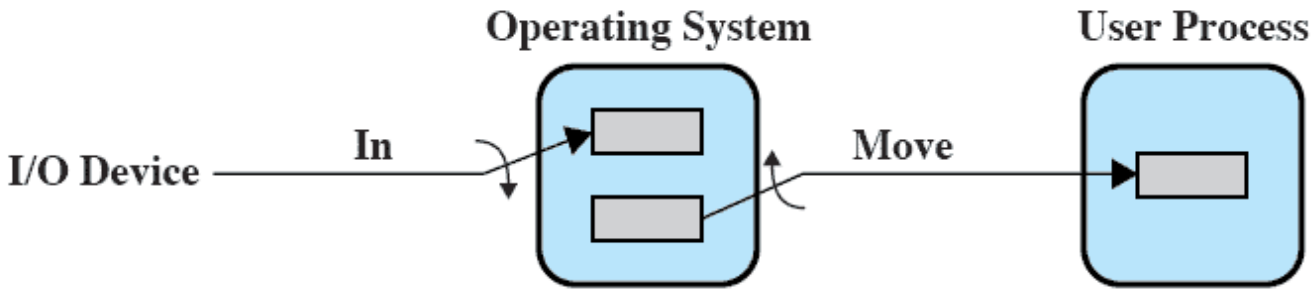


(a) No buffering

(b) Single buffering

Double Buffer

- Use two system buffers instead of one
- A process can transfer data to or from one buffer while the operating system empties or fills the other buffer

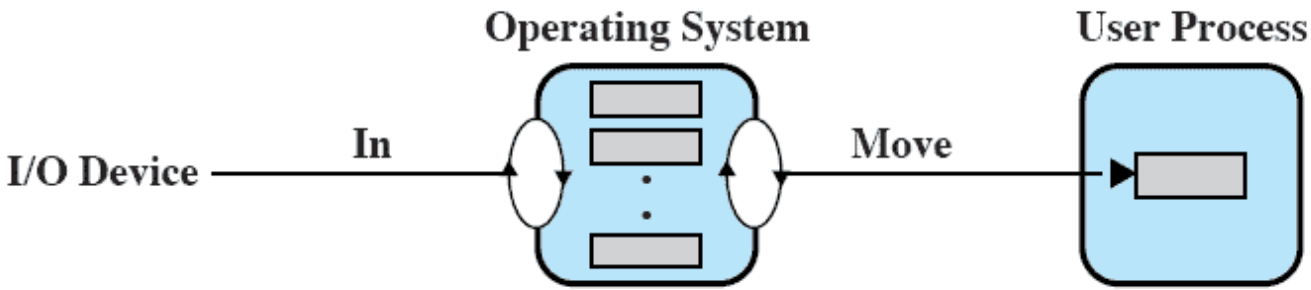


(c) Double buffering

- Process does not have to wait for I/O

Circular Buffer

- More than two buffers are used
- Each individual buffer is one unit in a circular buffer



- (d) Circular buffering

Disk Performance Parameters

- To read or write, the disk head must be positioned at the desired track and at the beginning of the desired sector
- Seek time
  - Time it takes to position the head at the desired track
- Rotational delay or rotational latency
  - Time its takes for the beginning of the sector to reach the head

Timing of Disk I/O Transfer

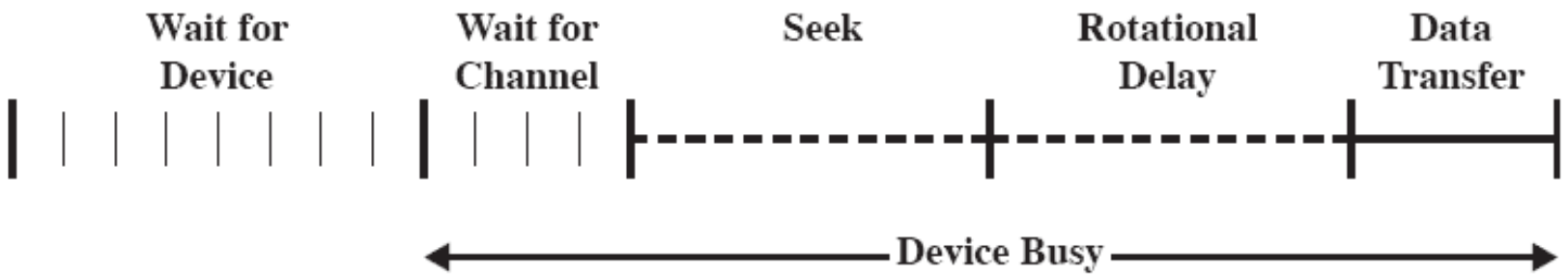


Figure 11.6 Timing of a Disk I/O Transfer

Disk Performance Parameters

- Access time
  - Sum of seek time and rotational delay
  - The time it takes to get in position to read or write

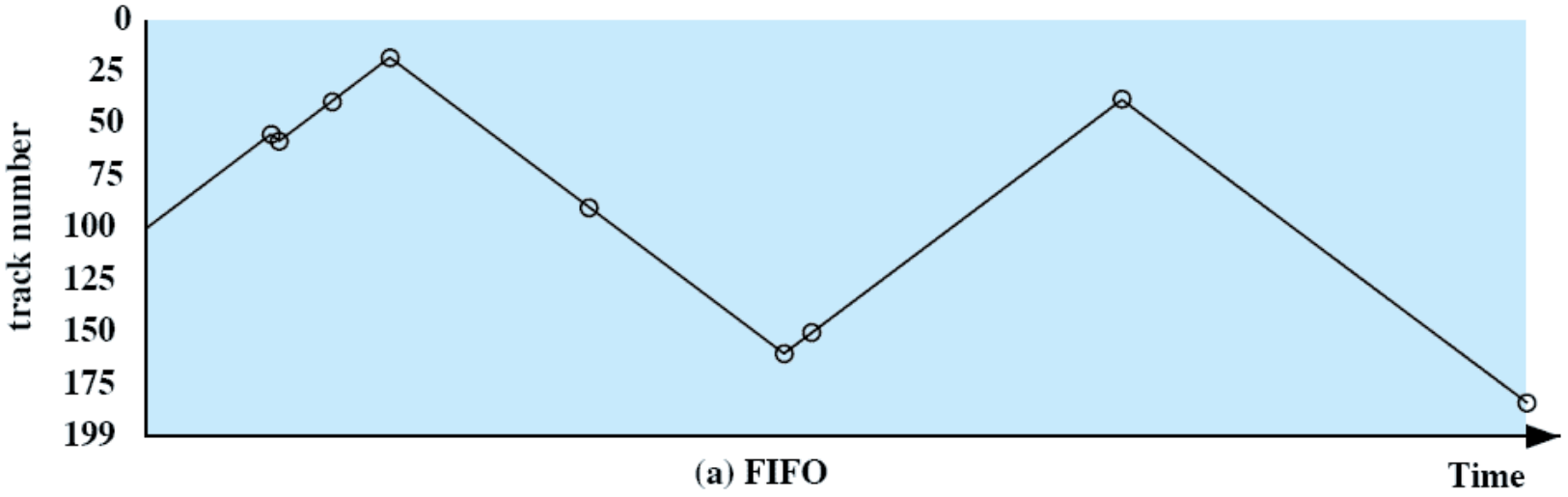
- There can be a huge difference between sequential read access and random read access.

Disk Scheduling Policies

- Seek time is the reason for differences in performance
- For a single disk there will be a number of I/O requests
- If requests are selected randomly, get poor performance

Disk Scheduling Policies

- First-in, first-out (FIFO)
  - Process requests sequentially
  - Fair to all processes
  - Approaches random scheduling in performance if there are many processes



Disk Scheduling Policies

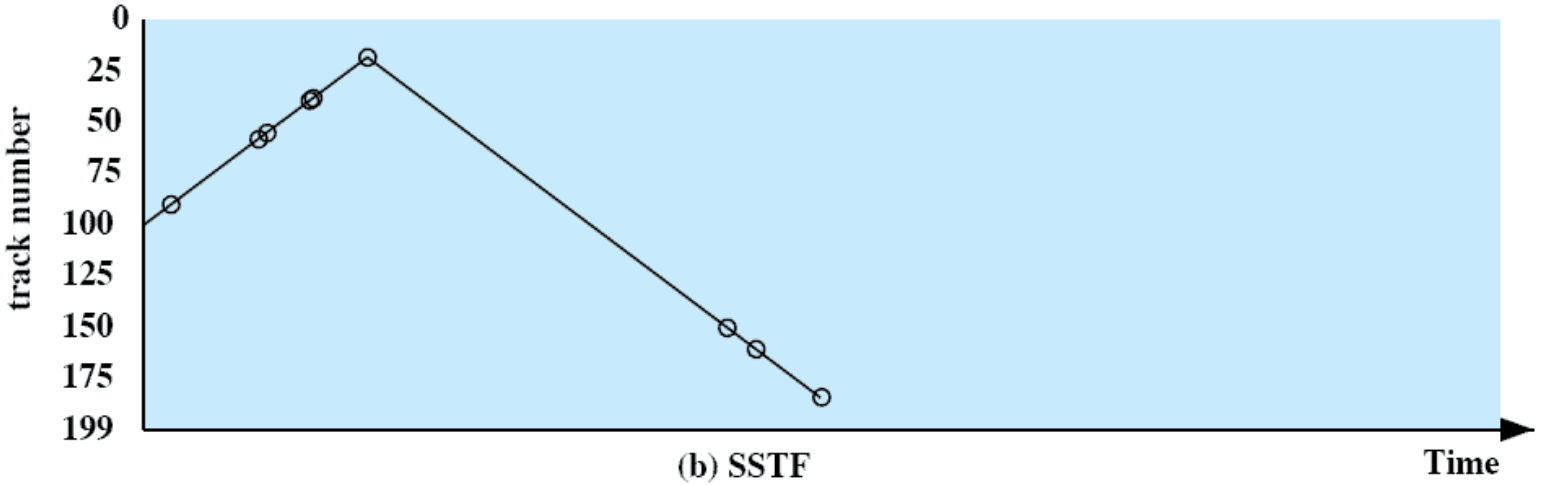
- Priority-based scheduling
  - Goal is not to optimize disk use but to meet other objectives
  - Example 1:
    - Short batch jobs may have higher priority
    - Provide good interactive response time
  - Example 2:
    - Earlier deadlines have higher priority
  - Can lead to counter measures by users
  - Can lead to starvation (principle of locality)

Disk Scheduling Policies

- Last-in, first-out
  - Good for transaction processing systems
    - The device is given to the most recent user so there should be little arm movement
  - Possibility of starvation since a job may never regain the head of the line

Disk Scheduling Policies

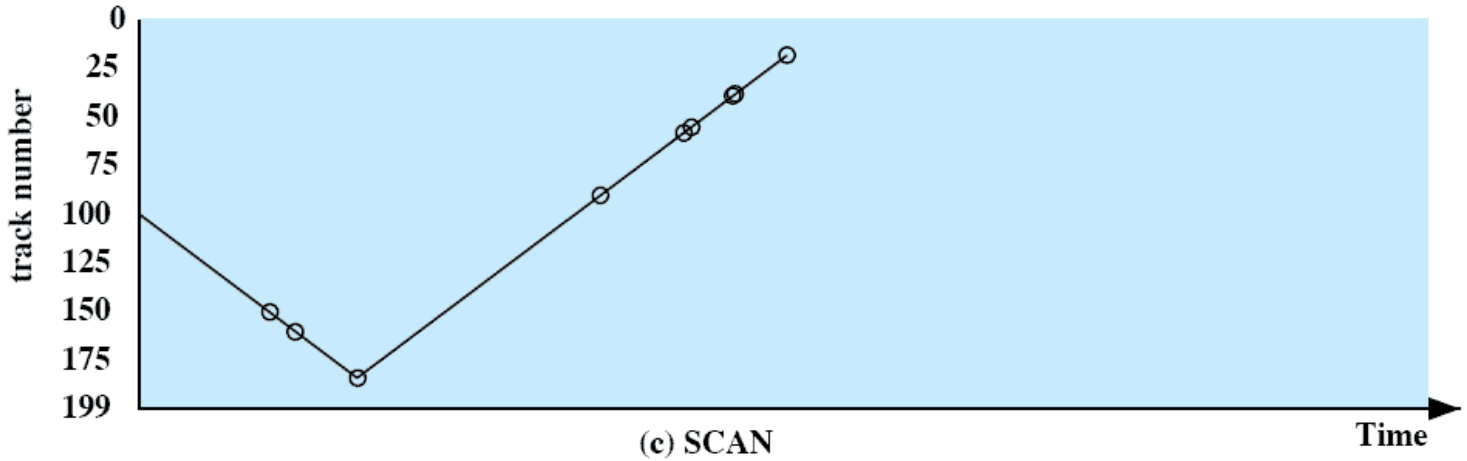
- Shortest Service Time First
  - Select the disk I/O request that requires the least movement of the disk arm from its current position
  - Always choose the minimum seek time
  - Requires knowledge of track position
  - Starvation still possible



(b) SSTF

Disk Scheduling Policies

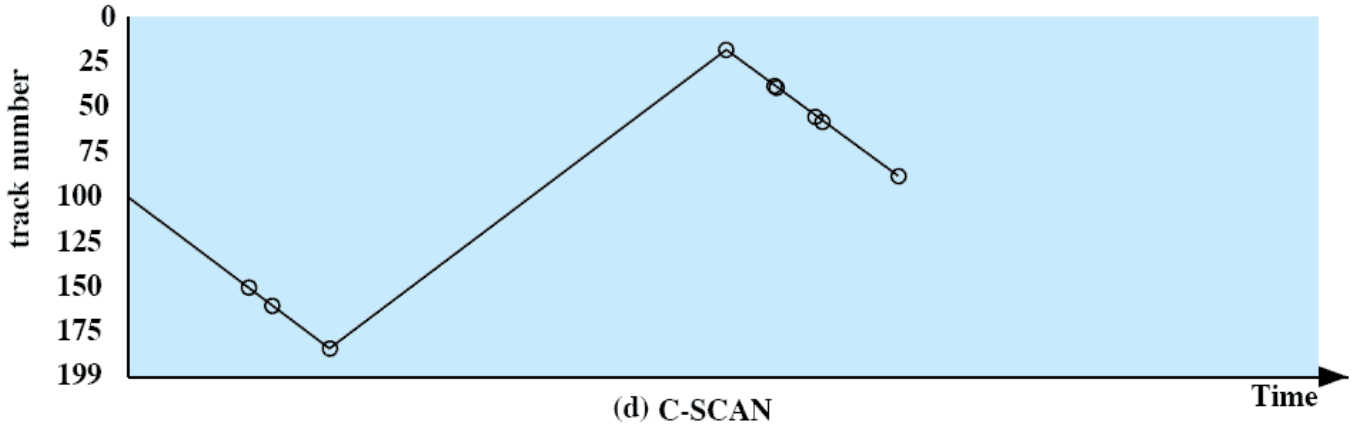
- SCAN
  - Arm moves in one direction only (increasing or decreasing track#), satisfying all outstanding requests until it reaches the last track in that direction, reverse the direction
  - Similar performance as SSTF
  - Biased against area most recently traversed
  -



(c) SCAN

Disk Scheduling Policies

- C-SCAN
  - Restricts scanning to one direction only
  - When the last track has been visited in one direction, the arm is returned to the opposite end of the disk and the scan begins again.
  - Deterministic, reduces delay for new requests



(d) C-SCAN

Disk Scheduling Policies

- N-step-SCAN
  - Segments the disk request queue into subqueues of length N
  - Subqueues are processed one at a time, using SCAN
  - New requests added to some other queue when queue is processed

Comparison

- assume track requests are 55, 58, 39, 18, 90, 160, 150, 38, 184 and the current track is 100:

FIFO		SSTF		SCAN		C-SCAN	
Nxt Trk	# Trks	Nxt Trk	# Trks	Nxt Trk	# Trks	Nxt Trk	# Trks
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
Avg:	55.3	Avg:	27.5	Avg:	27.8	Avg:	35.8

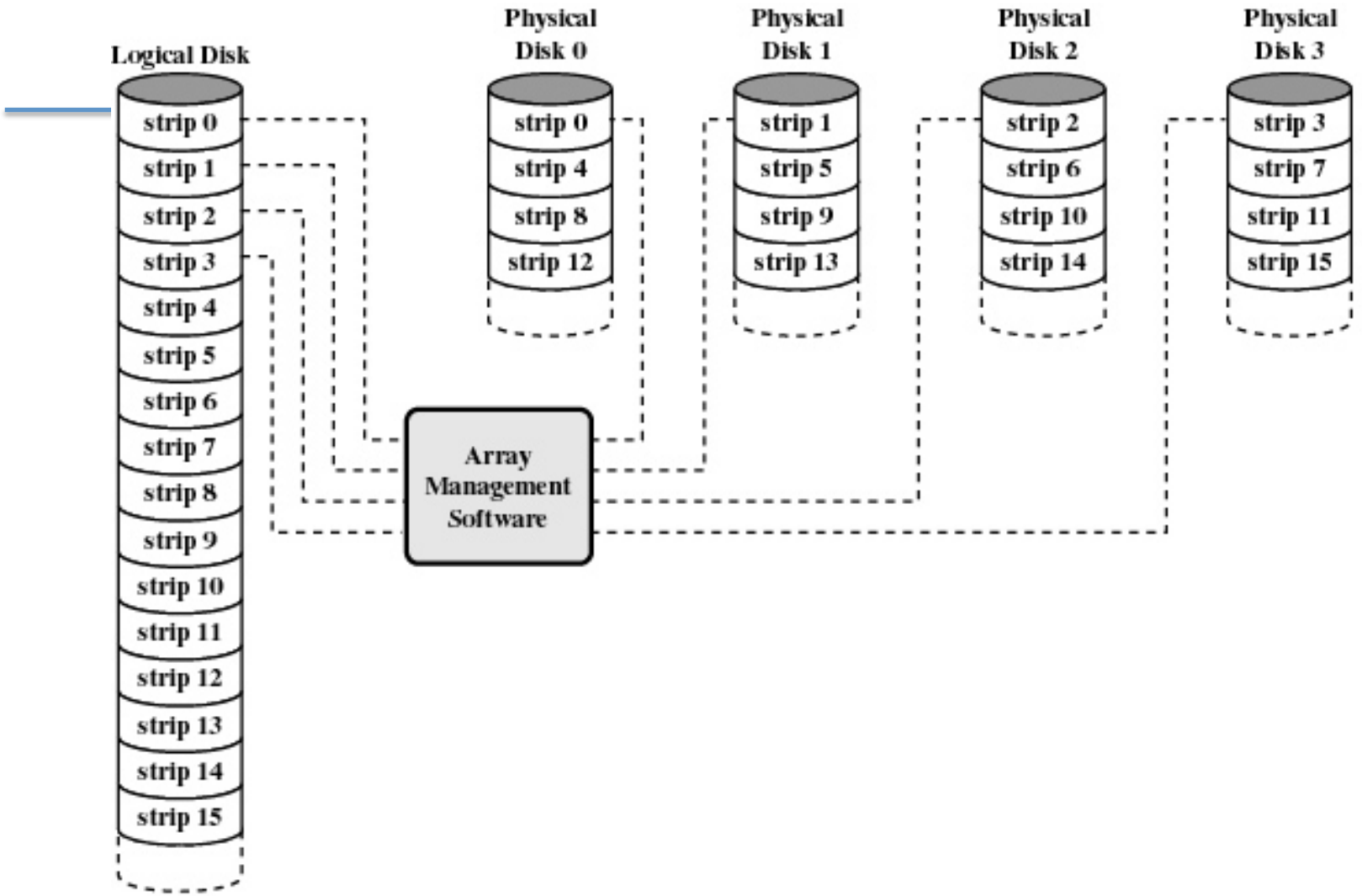
RAID (Redundant Array of Independent Disks)

- Organize data in which redundancy can be added to improve reliability

- Set of physical disk drives viewed by the operating system as a single logical drive
- Data are distributed across the physical drives of an array
- Redundant disk capacity is used to store parity information → recovery possible

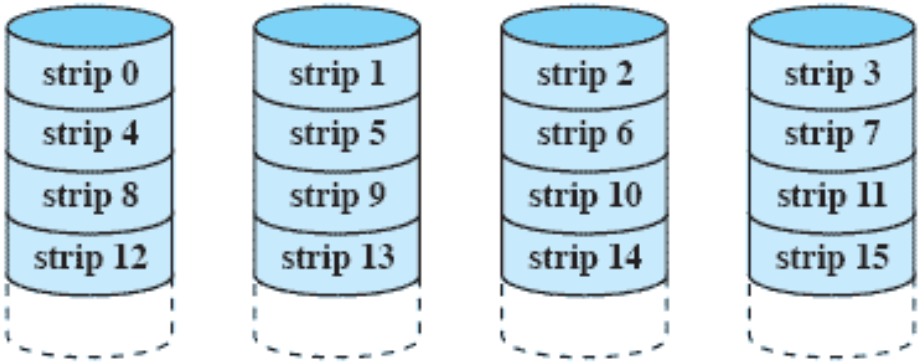
RAID

Category	Level	Description	I/O Req. Rate	Transfer Rate	Typ. App.
Stripping	0	Non-redundant	Large strips: Excellent	Small Strips: Excellent	high performance, non critical data
Mirroring	1	Mirrored	Good/Fair	Fair/fair	System drives; critical files
Parallel Access	2	Redundant via Hamming code	Poor	Excellent	
	3	Bit-interleaved parity	Poor	Excellent	Big I/O req. size
Independent Access	4	Block-interleaved parity	Excellent/fair	Fair/poor	
	5	Block-interleaved distributed parity	Excellent/fair	Fair/poor	High req. rate, read intensive
	6	Block-interleaved dual distributed parity	Excellent/poor	Fair/poor	need very high availability



Data Mapping for a RAID Level 0 Array [MASS97]

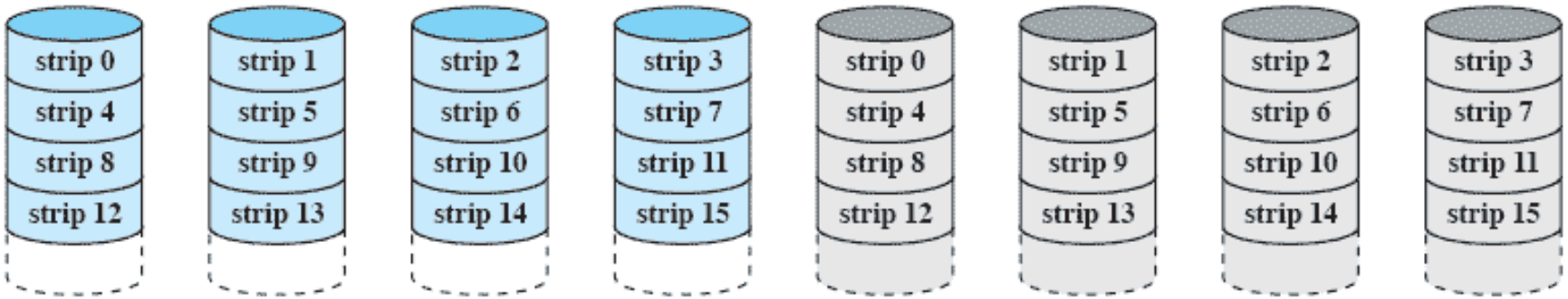
RAID 0 (nonredundant)



(a) RAID 0 (non-redundant)

- High data transfer capacity
- Lower reliability (Mean time to failure increased due to several drives)
- Not “true RAID”

RAID 1 (mirrored)

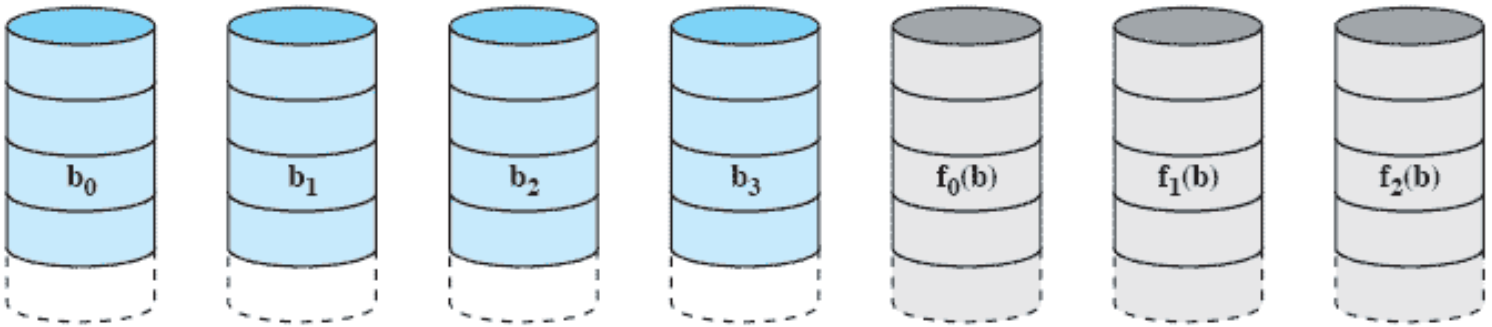


(b) RAID 1 (mirrored)

RAID 1 (mirrored)

- redundancy achieved by simple duplication of all data
- advantages:
  - read requests can be serviced by either disk
  - write requires both strips be updated but it can be done in parallel
  - recovery from single drive failure simple: data may still be accessed from other drive.
- primary disadvantage:
  - cost of twice the disk space required

RAID 2 (redundancy through Hamming code)

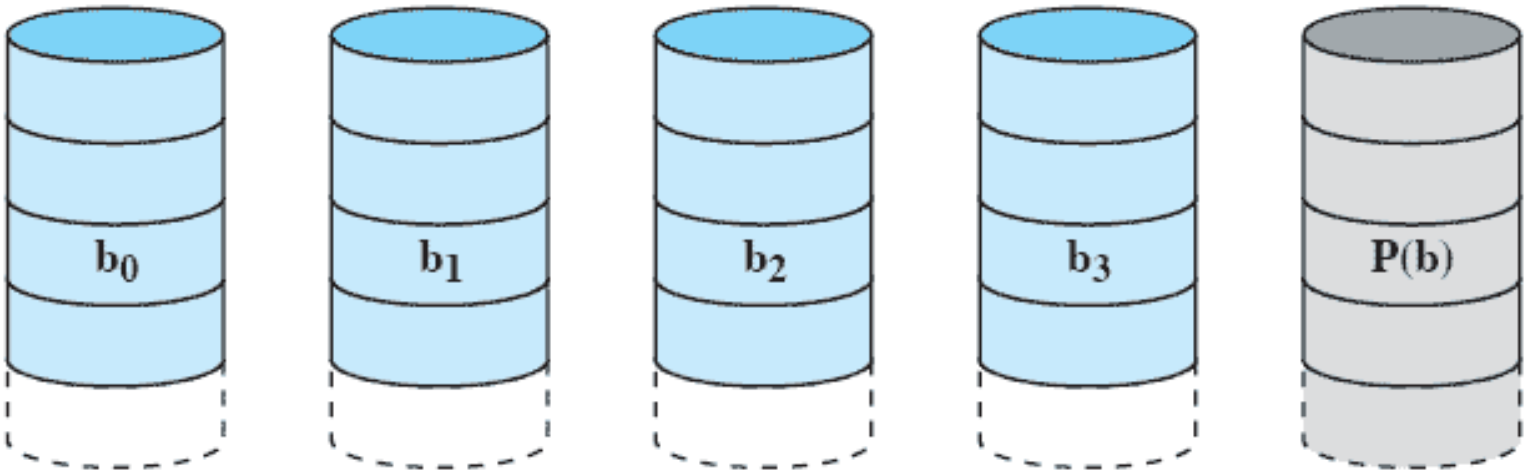


(c) RAID 2 (redundancy through Hamming code)

RAID 2

- all disks participate in disk transfers, spindles synchronized so each disk head is in same position on each disk
- error correcting code calculated across corresponding bits on each data disk; bits of code stored in corresponding bit positions on multiple parity disks
- single-bit errors recognized and corrected immediately with no impact on access time
- on single write, all data and parity disks must be accessed
- generally an overkill solution and not used given the high reliability of modern disks

RAID 3 (bit-interleaved parity)



(d) RAID 3 (bit-interleaved parity)

RAID 3

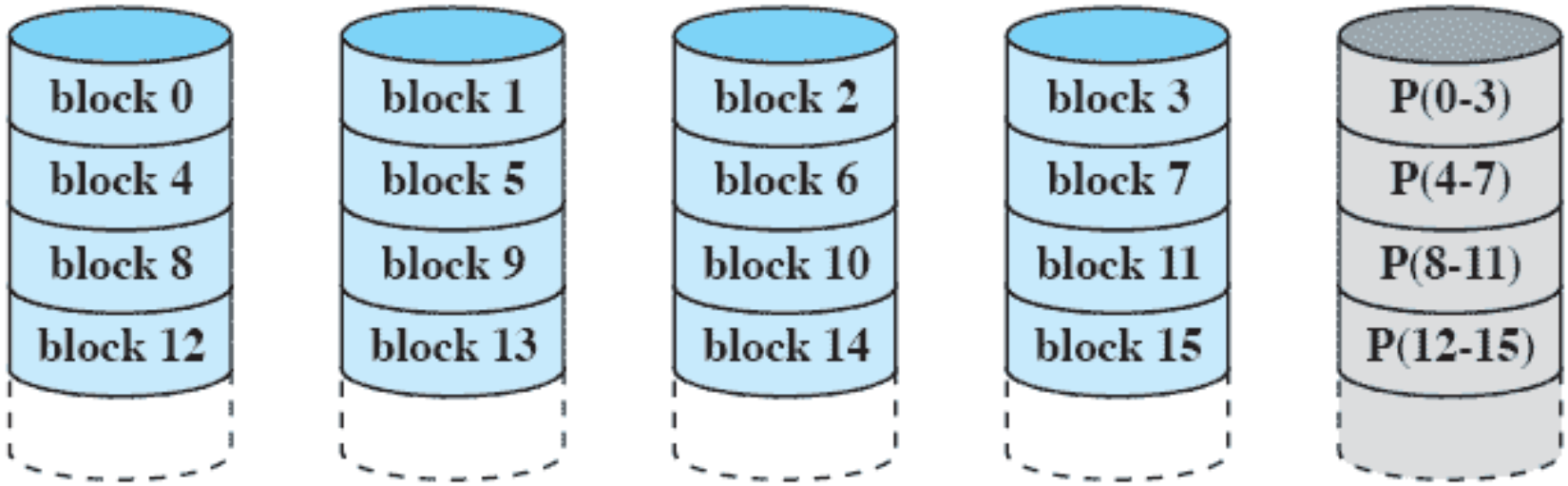
- uses only one redundant disk
- uses parallel access, data distributed in small strips (bit/byte level)
- compute a simple parity bit for the set of individual bits in same position on all of the data disks
- example: disks X0, X1, X2, X3 store data and X4 stores parity
  - parity for i-th bit calculated as:

    $$X4(i) = X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i)$$

  - let X1 fail, then add $x4(i) \oplus X1(i)$ to both sides as:

    $$X1(i) = X4(i) \oplus X3(i) \oplus X2(i) \oplus X0(i)$$

RAID 4 (block-level parity)

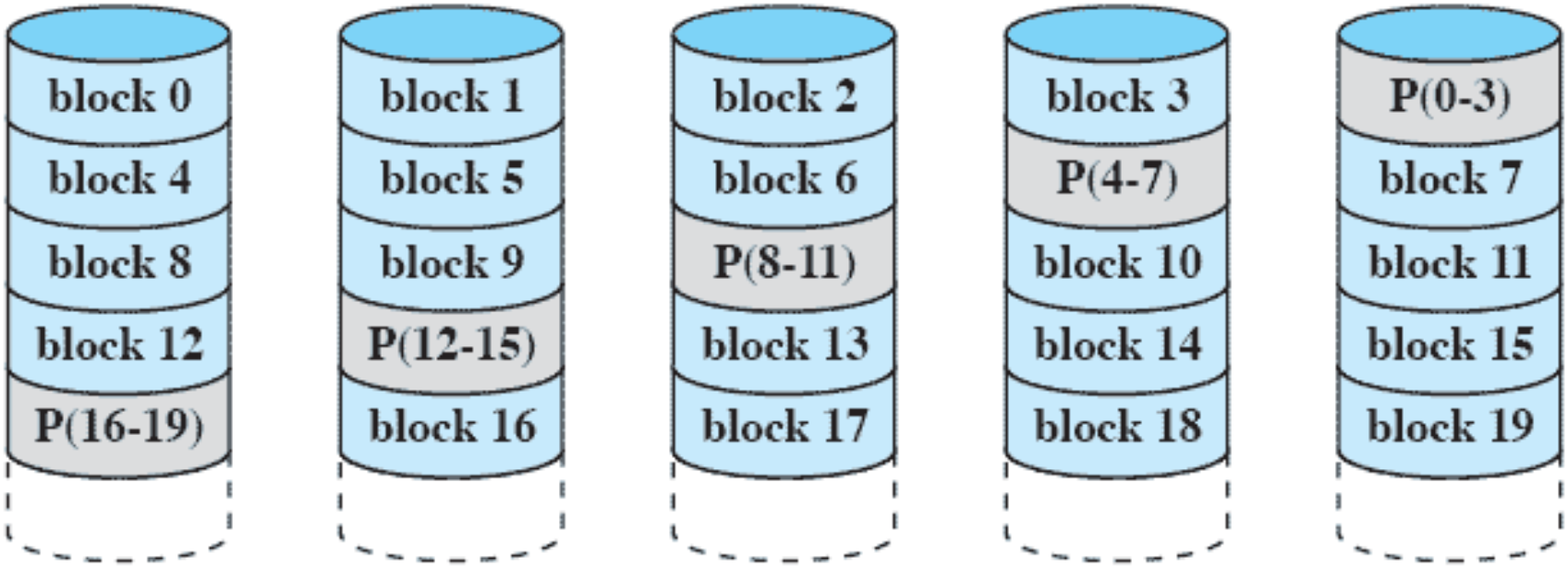


(e) RAID 4 (block-level parity)

RAID 4

- uses independent access array – each disk operates independently --> separate I/O requests can be satisfied in parallel
- a bit-by-bit parity strip calculated across corresponding strips on each data disk
  - parity bits stored in corresponding strip on parity disk
- penalty on small size writes
  - update data + corresponding parity bits

RAID 5 (block-level distributed parity)



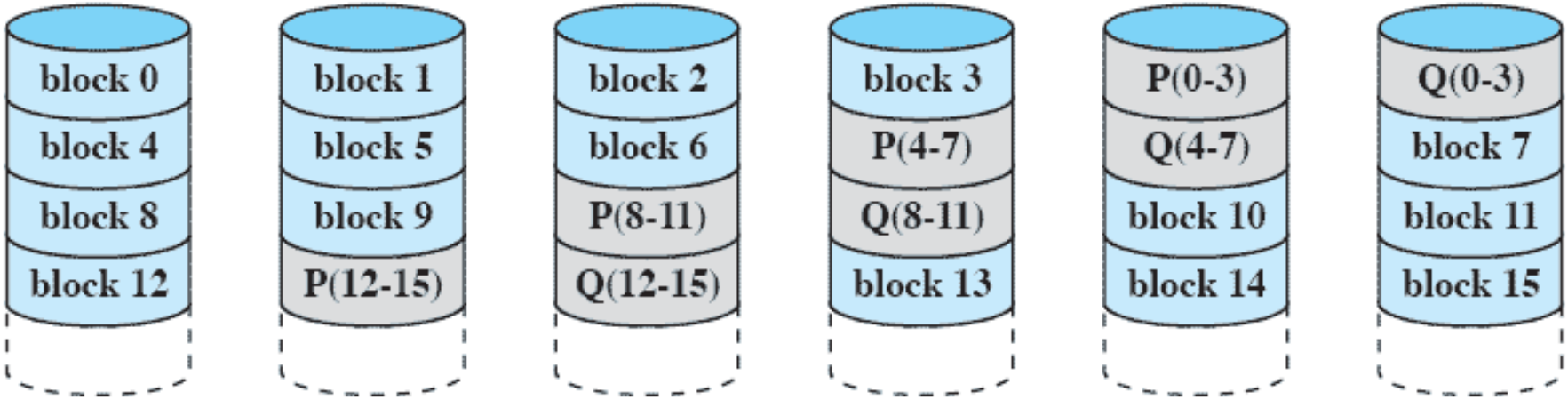
(f) RAID 5 (block-level distributed parity)

RAID 5

- similar to RAID 4 *except* parity strips distributed across all disks

- this distribution of parity strips avoids potential I/O bottleneck of single disk parity of RAID 4
- Bad when many incoming write requests hit the same parity stripe

RAID 6 (dual redundancy)



(g) RAID 6 (dual redundancy)

RAID 6

- two different parities are carried and stored in separate blocks on different disks
- user with N disk requirement needs N+2 disk configuration
- possible to regenerate data even if two disks containing user data fail
- extremely high availability but substantial write penalty since each write affects two parity blocks