*Operating Systems: Internals and Design Principles*

# Chapter 11
# I/O Management and Disk Scheduling

Ninth Edition
By William Stallings

# Categories of I/O Devices

External devices that engage in I/O with computer systems can be grouped into three categories:

**Human readable**

- Suitable for communicating with the computer user
- Printers, terminals, video display, keyboard, mouse

**Machine readable**

- Suitable for communicating with electronic equipment
- Disk drives, USB keys, sensors, controllers

**Communication**

- Suitable for communicating with remote devices
- Modems, digital line drivers

# Differences in I/O Devices

- Devices differ in a number of areas:

*Data Rate*

- There may be differences of magnitude between the data transfer rates

*Application*

- The use to which a device is put has an influence on the software

*Complexity of Control*

- The effect on the operating system is filtered by the complexity of the I/O module that controls the device

*Unit of Transfer*

- Data may be transferred as a stream of bytes or characters or in larger blocks

*Data Representation*

- Different data encoding schemes are used by different devices

*Error Conditions*

- The nature of errors, the way in which they are reported, their consequences, and the available range of responses differs from one device to another

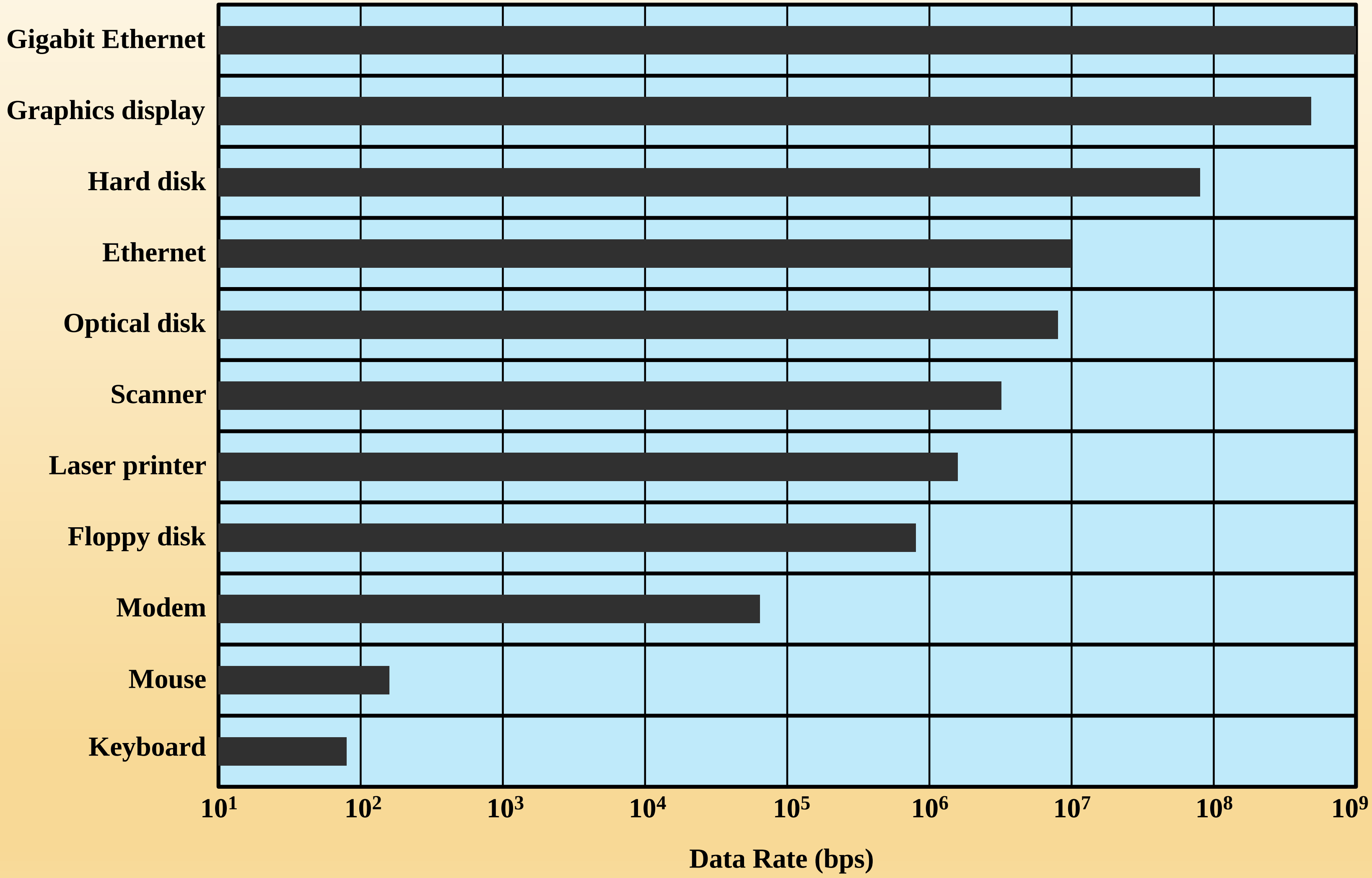


**Figure 11.1 Typical I/O Device Data Rates**

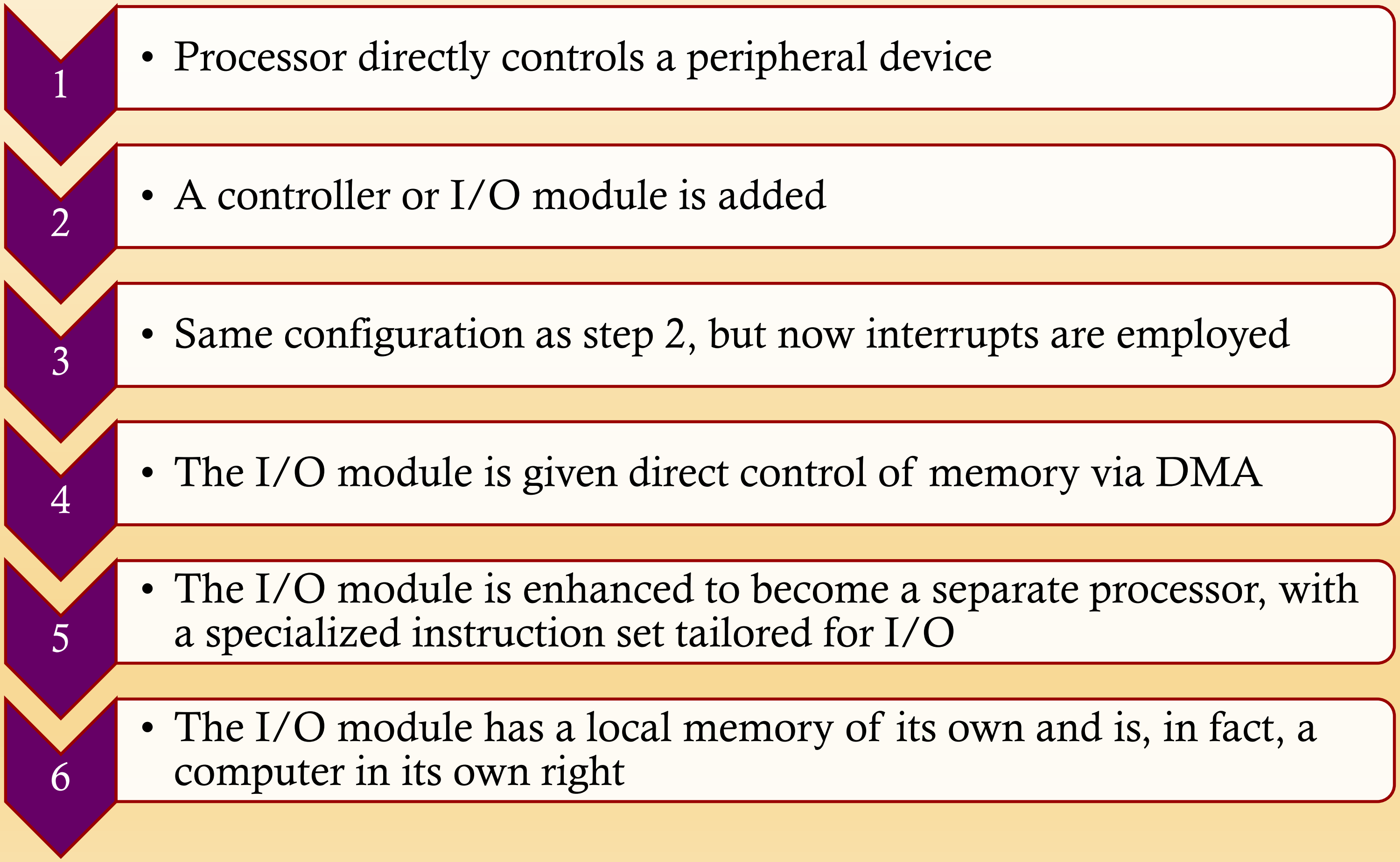
# Organization of the I/O Function

- Three techniques for performing I/O are:
- **Programmed I/O**
  - The processor issues an I/O command on behalf of a process to an I/O module; that process then busy waits for the operation to be completed before proceeding
- **Interrupt-driven I/O**
  - The processor issues an I/O command on behalf of a process
    - If non-blocking – processor continues to execute instructions from the process that issued the I/O command
    - If blocking – the next instruction the processor executes is from the OS, which will put the current process in a blocked state and schedule another process
- **Direct Memory Access (DMA)**
    - A DMA module controls the exchange of data between main memory and an I/O module

# 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



1 - Processor directly controls a peripheral device

2 - A controller or I/O module is added

3 - Same configuration as step 2, but now interrupts are employed

4 - The I/O module is given direct control of memory via DMA

5 - The I/O module is enhanced to become a separate processor, with a specialized instruction set tailored for I/O

6 - The I/O module has a local memory of its own and is, in fact, a computer in its own right

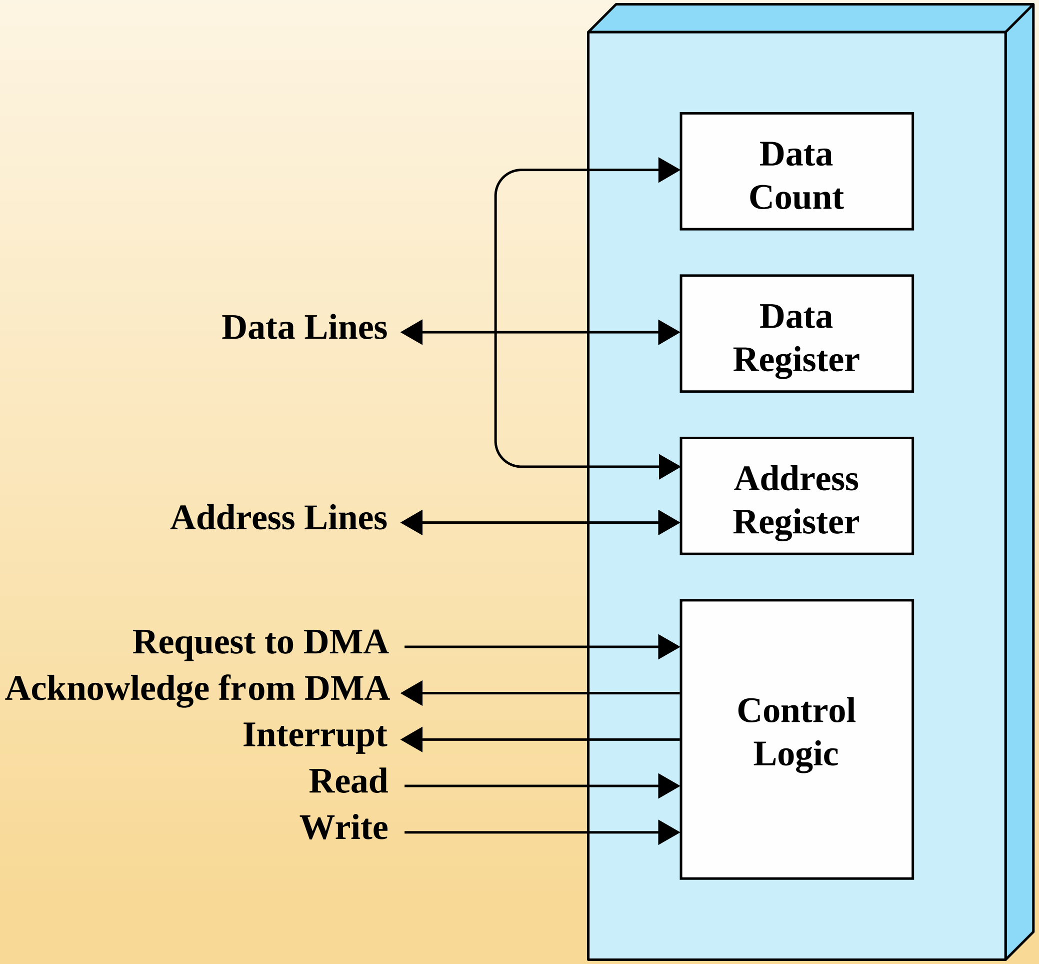


**Figure 11.2 Typical DMA Block Diagram**

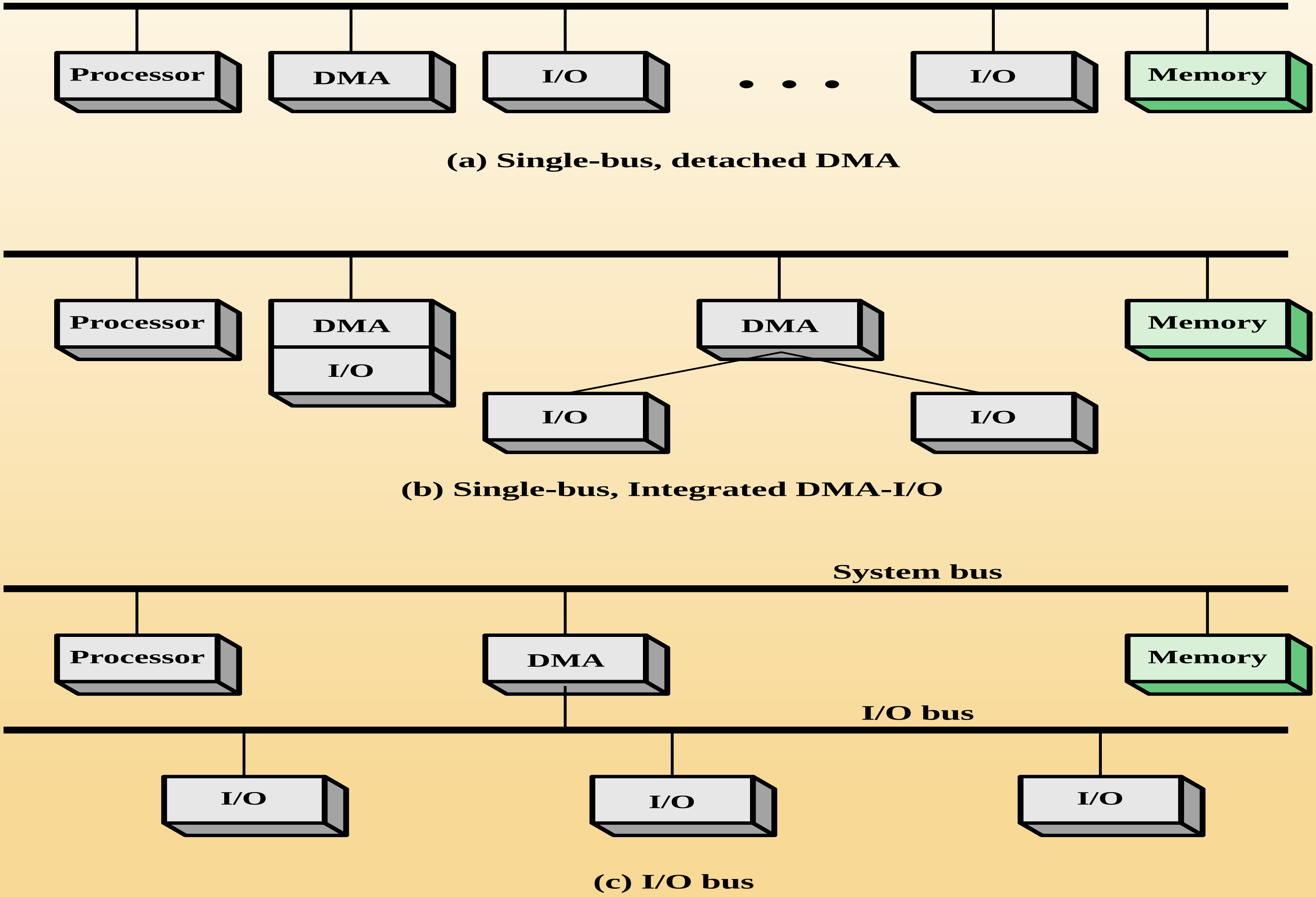


**Figure 11.3 Alternative DMA Configurations**

# Design Objectives

## Efficiency

- Major effort in I/O design
- Important because I/O operations often form a bottleneck
- Most I/O devices are extremely slow compared with main memory and the processor
- The area that has received the most attention is disk I/O

## Generality

- Desirable to handle all devices in a uniform manner
- Applies to the way processes view I/O devices and the way the operating system manages I/O devices and operations
- Diversity of devices makes it difficult to achieve true generality
- Use a hierarchical, modular approach to the design of the I/O function

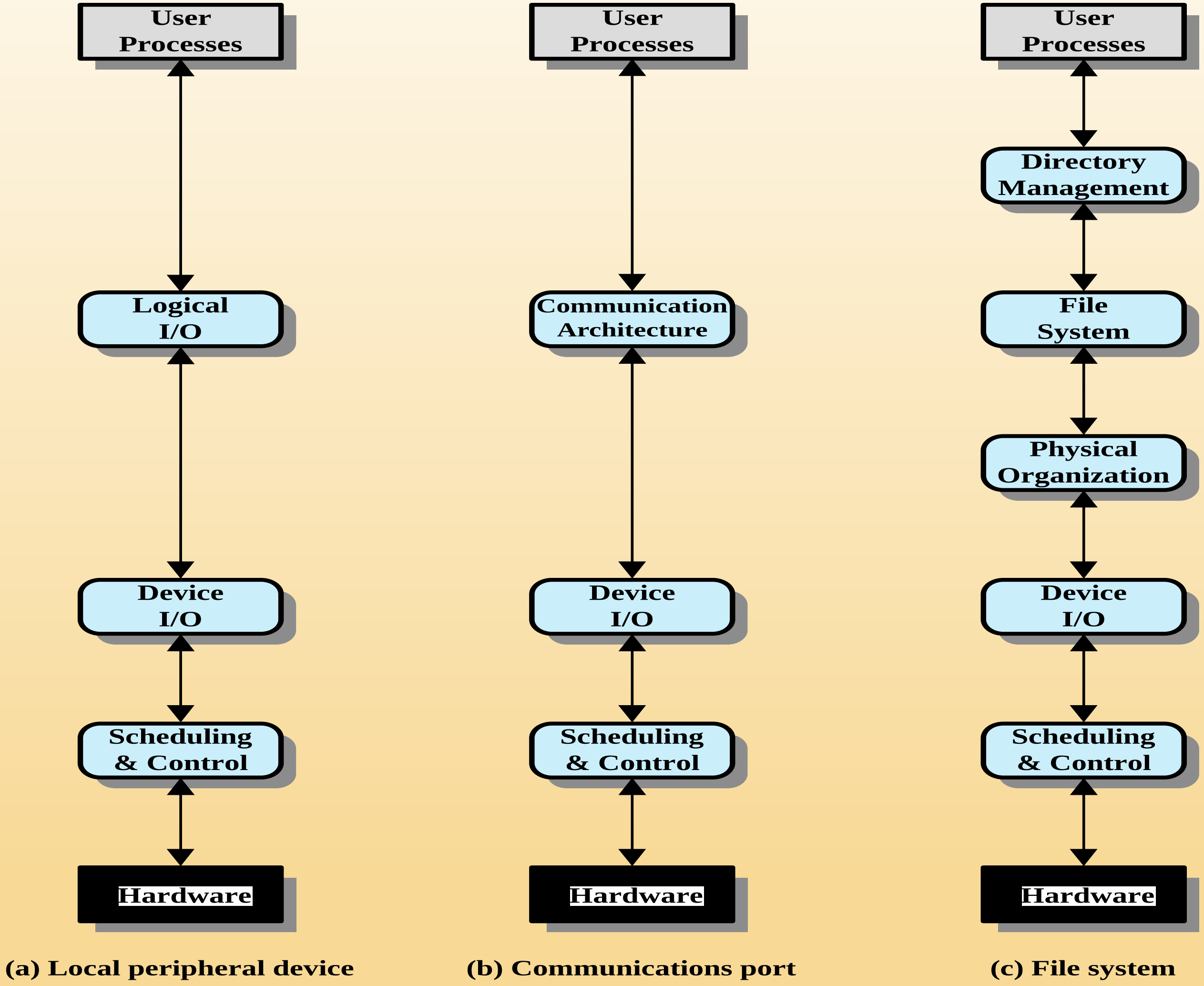


**Figure 11.4 A Model of I/O Organization**

# Buffering

- To avoid overheads and inefficiencies, it is sometimes convenient to perform input transfers in advance of requests being made, and to perform output transfers some time after the request is made

### Block-oriented device

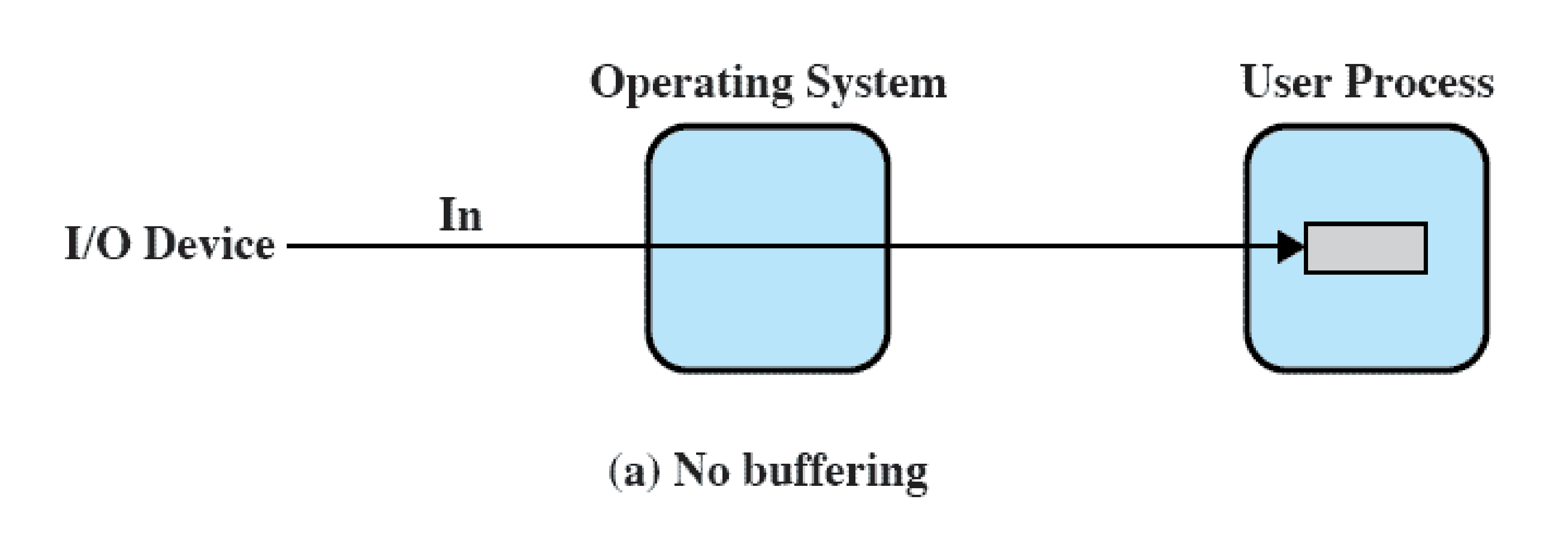
- Stores information in blocks that are usually of fixed size
- Transfers are made one block at a time
- Possible to reference data by its block number
- Disks and USB keys are examples

### Stream-oriented device

- Transfers data in and out as a stream of bytes
- No block structure
- Terminals, printers, communications ports, mouse and other pointing devices, and most other devices that are not secondary storage are examples

# No Buffer

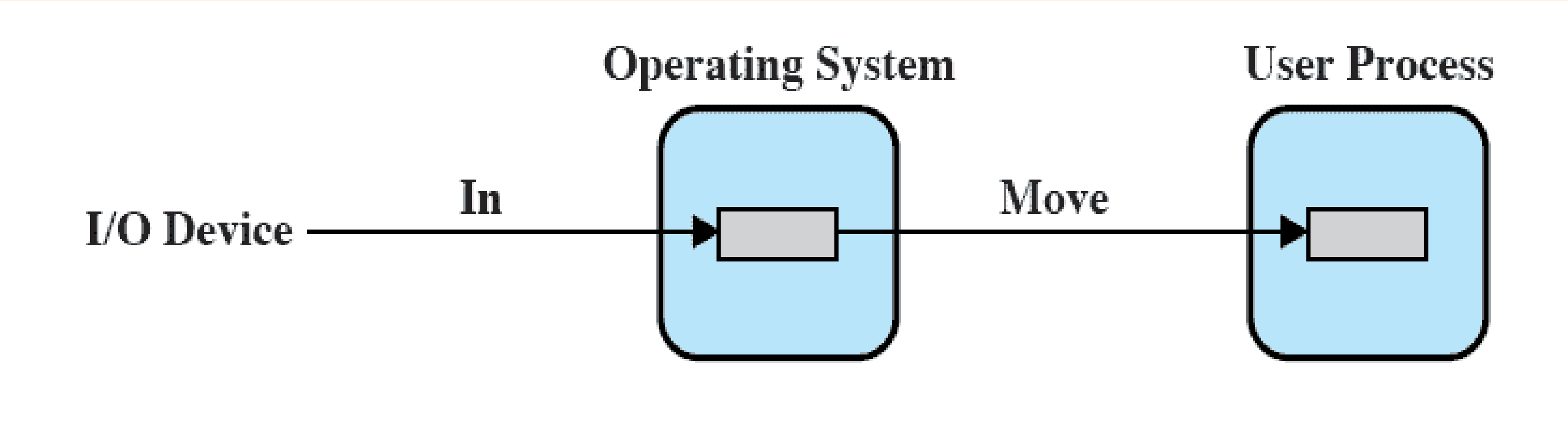
- Without a buffer, the OS directly accesses the device when it needs



(a) No buffering

# Single Buffer

- The simplest type of support that the operating system can provide
- When a user process issues an I/O request, the OS assigns a buffer in the system portion of main memory to the operation



(b) Single buffering

# Single Buffer

- Input transfers are made to the system buffer
- Reading ahead/anticipated input
  - Is done in the expectation that the block will eventually be needed
  - When the transfer is complete, the process moves the block into user space and immediately requests another block
- Approach generally provides a speedup compared to the lack of system buffering
  - The user process can be processing one block of data while the next block is being read in
  - The OS is able to swap the process out because the input operation is taking place in system memory rather than user process memory
- Disadvantages:
  - Complicates the logic in the operating system
  - Swapping logic is also affected

# Single Buffering for Stream-Oriented I/O

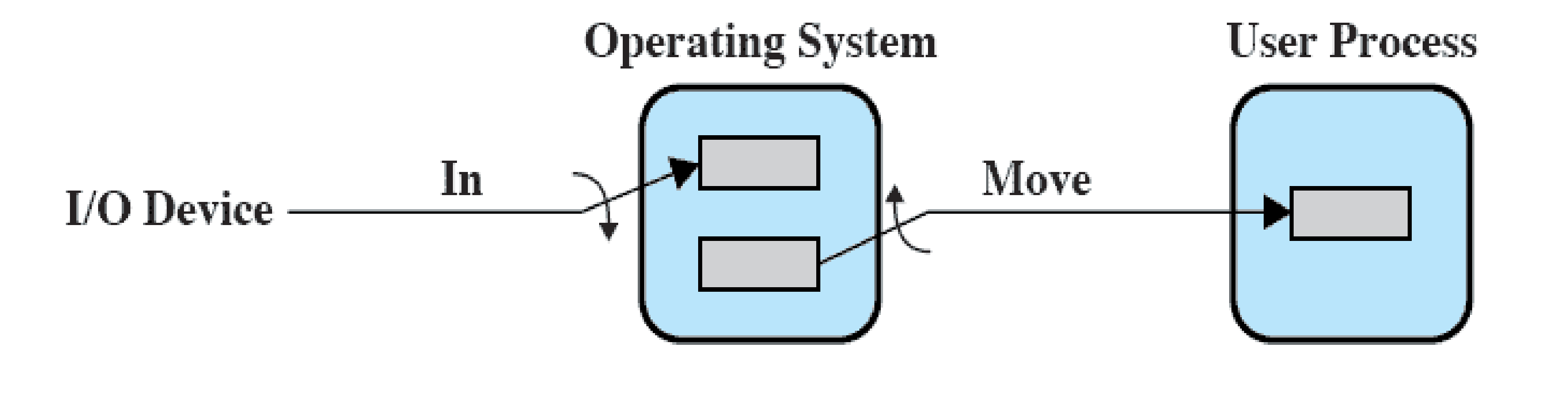
- Can be used in a line-at-a-time fashion or a byte-at-a-time fashion

- Line-at-a-time operation is appropriate for scroll-mode terminals (dumb terminals)
  - With this form of terminal, user input is one line at a time, with a carriage return signaling the end of a line
  - Output to the terminal is similarly one line at a time

- Byte-at-a-time operation is used on forms-mode terminals, when each keystroke is significant and for many other peripherals, such as sensors and controllers

# Double Buffer

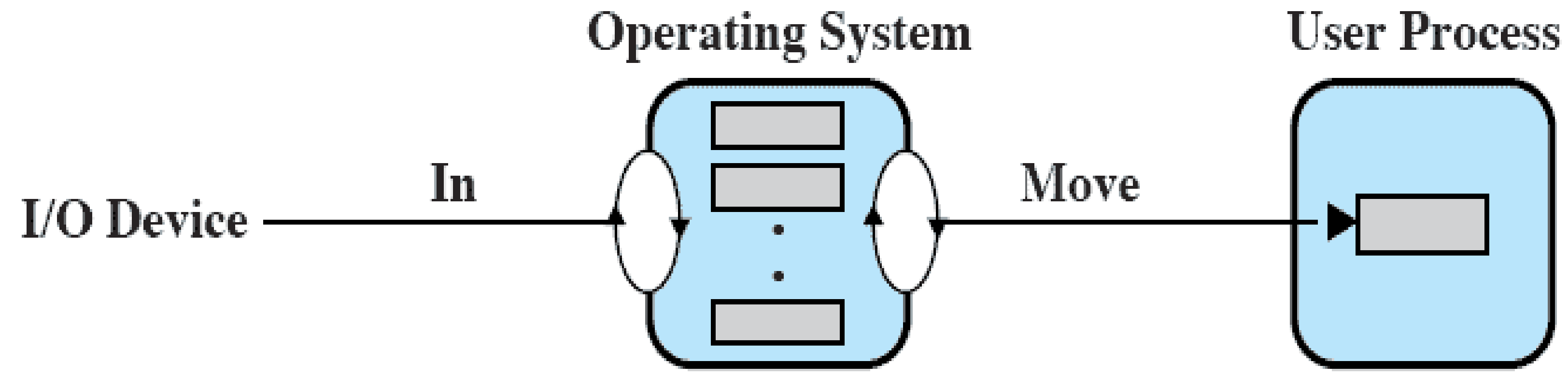
- Assigning two system buffers to the operation
- A process now transfers data to or from one buffer while the operating system empties or fills the other buffer
- Also known as buffer swapping



(c) Double buffering

# Circular Buffer

- When more than two buffers are used, the collection of buffers is itself referred to as a circular buffer
- Each individual buffer is one unit in the circular buffer



(d) Circular buffering

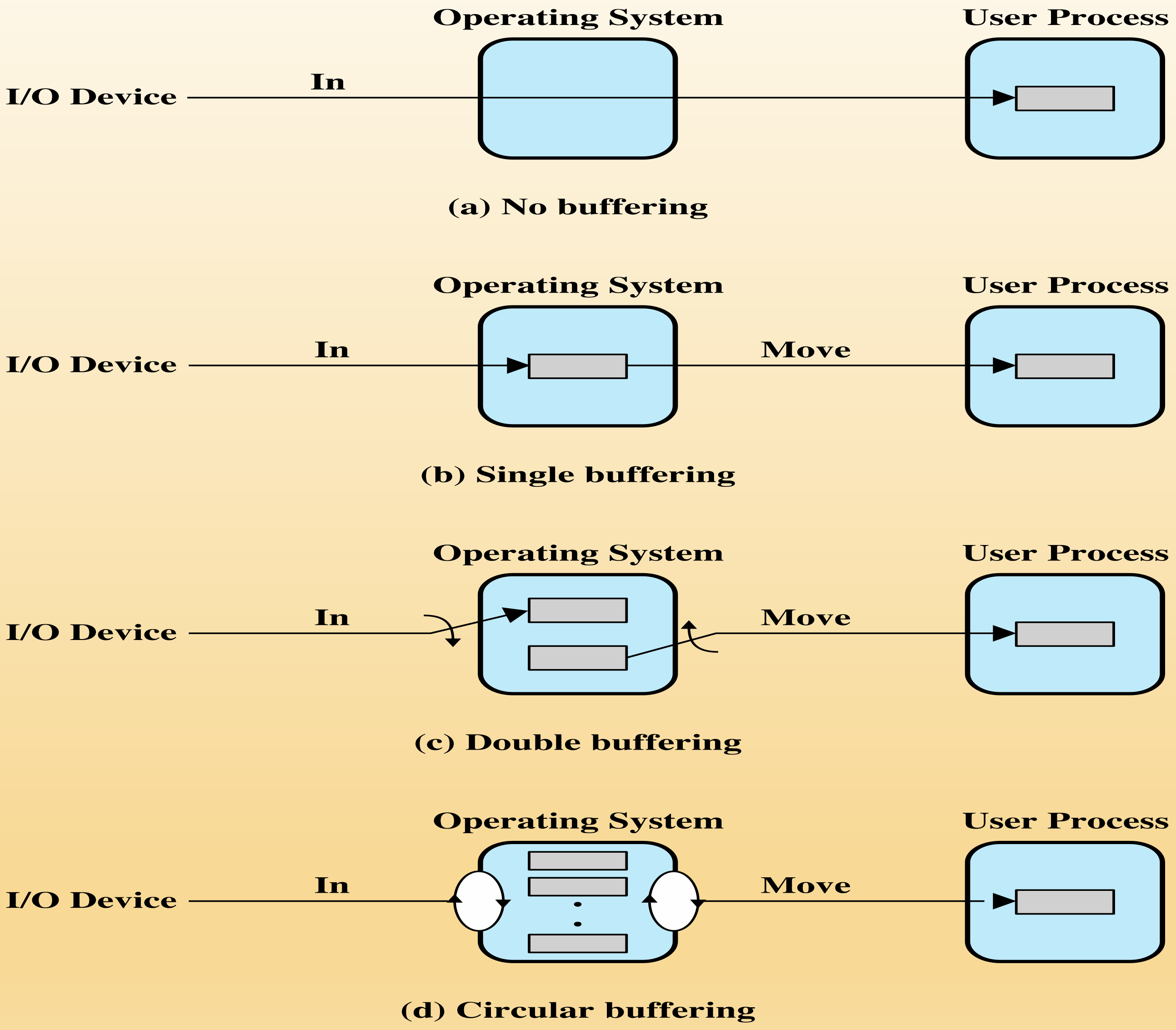


**Figure 11.5 I/O Buffering Schemes (input)**

# Disk Performance Parameters

- The actual details of disk I/O operation depend on the:
  - Computer system
  - Operating system
  - Nature of the I/O channel and disk controller hardware

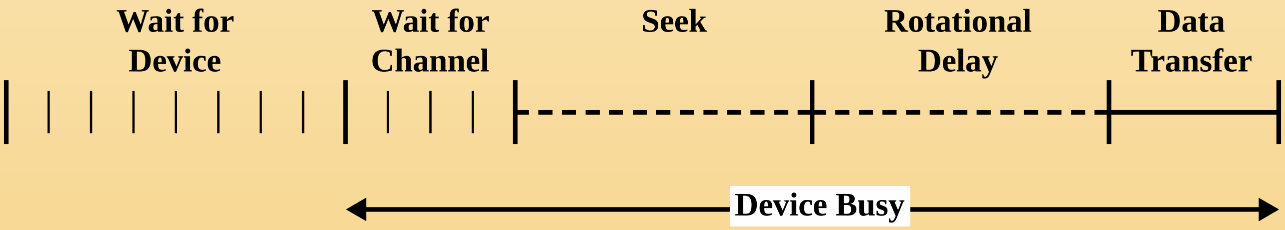


**Figure 11.6 Timing of a Disk I/O Transfer**

# Disk Performance Parameters

- When the disk drive is operating, the disk is rotating at constant speed
- To read or write the head must be positioned at the desired track and at the beginning of the desired sector on that track
- Track selection involves moving the head in a movable-head system or electronically selecting one head on a fixed-head system
- On a movable-head system the time it takes to position the head at the track is known as **seek time**
- The time it takes for the beginning of the sector to reach the head is known as **rotational delay**
- The sum of the seek time and the rotational delay equals the **access time**

# Seek Time

- The time required to move the disk arm to the required track
- Consists of two key components:
    - The initial startup time
    - The time taken to traverse the tracks that have to be crossed once the access arm is up to speed
- Settling time
    - Time after positioning the head over the target track until track identification is confirmed
- Much improvement comes from smaller and lighter disk components
- A typical average seek time on contemporary hard disks is under 10ms

# Disk Performance

- Rotational delay
  - The time required for the addressed area of the disk to rotate into a position where it is accessible by the read/write head
  - Disks rotate at speeds ranging from 3,6000 rpm (for handheld devices such as digital cameras) up to 15,000 rpm

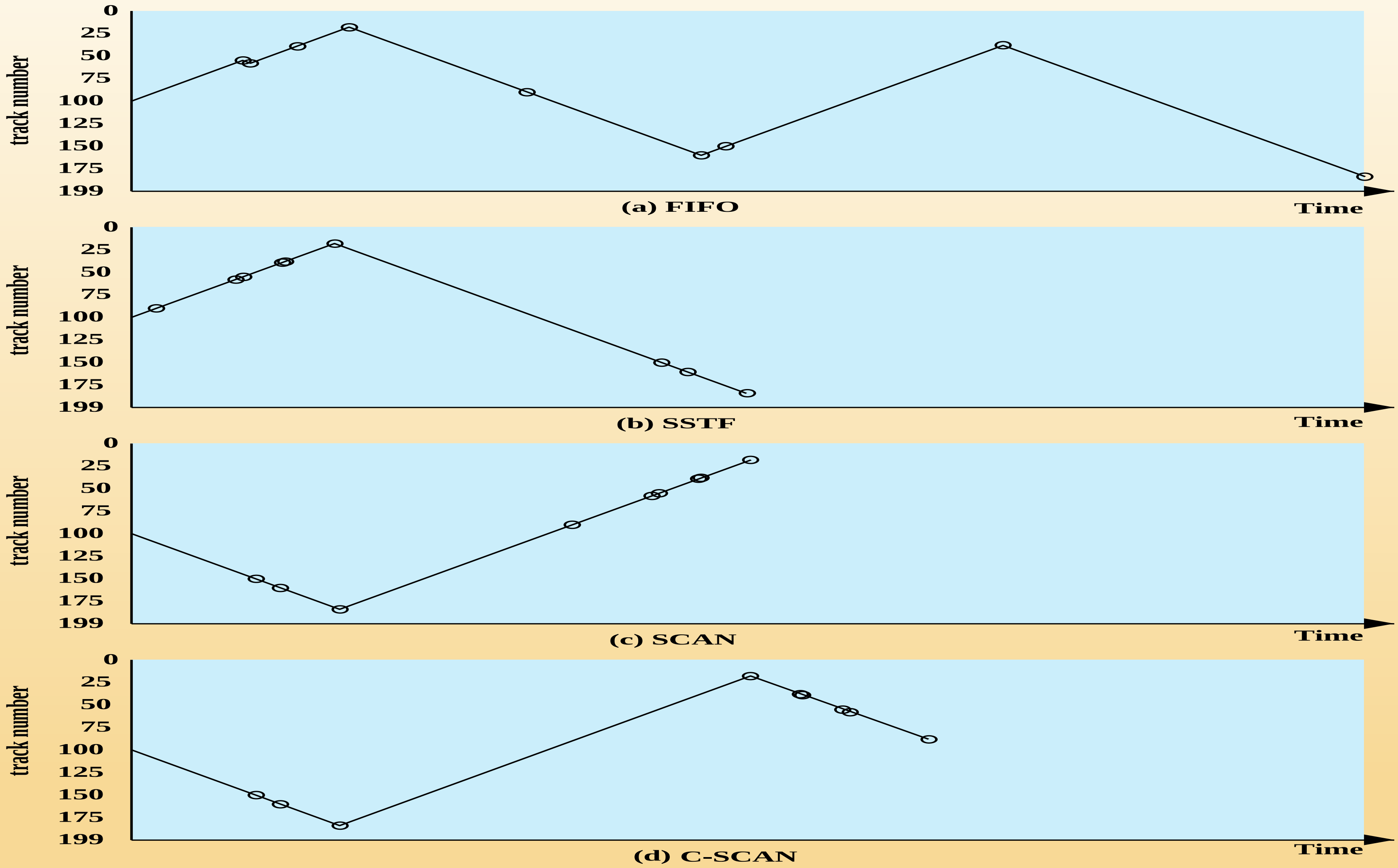
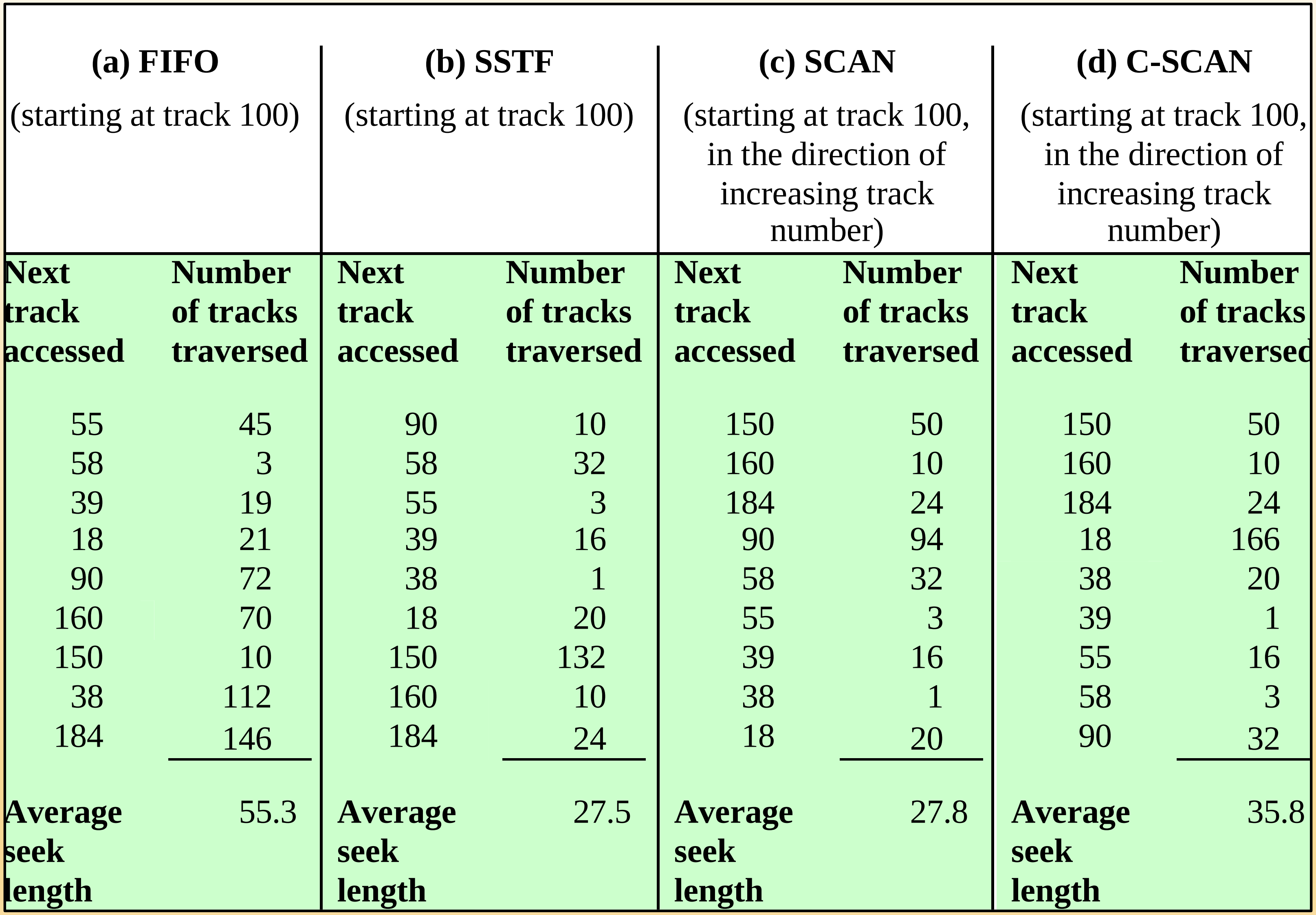


**Figure 11.7 Comparison of Disk Scheduling Algorithms (see Table 11.3)**

The requested tracks, in the order
received by the disk scheduler, are 55, 58, 39, 18, 90, 160, 150, 38, 184.

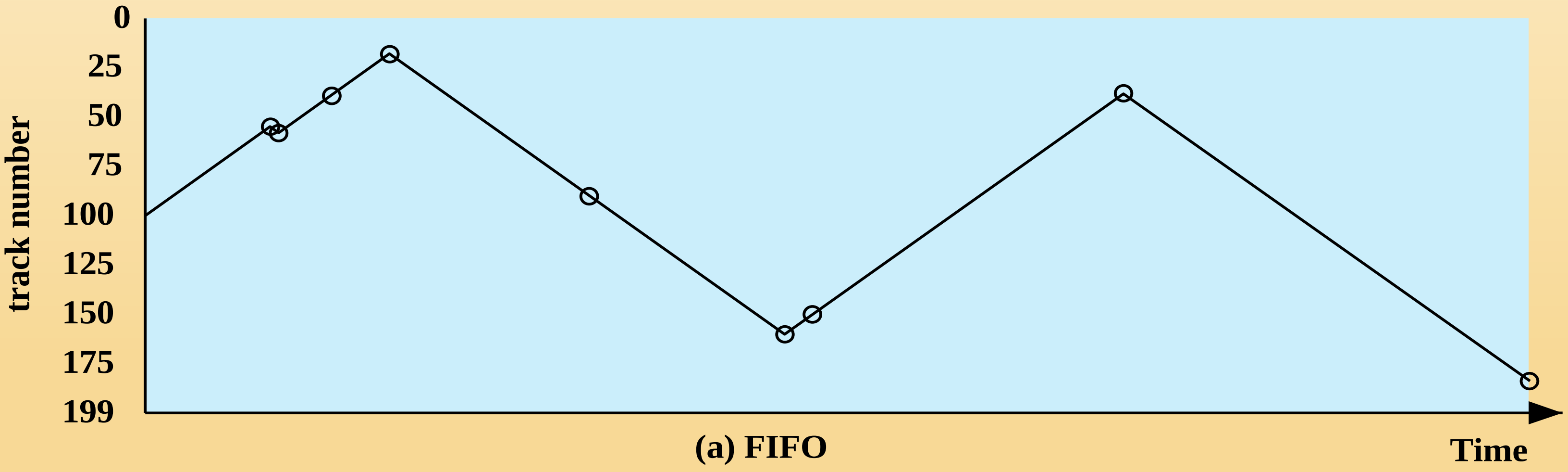


| **(a) FIFO**<br>(starting at track 100) | | **(b) SSTF**<br>(starting at track 100) | | **(c) SCAN**<br>(starting at track 100, in the direction of increasing track number) | | **(d) C-SCAN**<br>(starting at track 100, in the direction of increasing track number) | |
|---|---|---|---|---|---|---|---|
| **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 11.2 Comparison of Disk Scheduling Algorithms**

# First-In, First-Out (FIFO)

- Processes in sequential order
- Fair to all processes
- Approximates random scheduling in performance if there are many processes competing for the disk



(a) FIFO

The requested tracks, in the order
received by the disk scheduler, are 55, 58, 39, 18, 90, 160, 150, 38, 184.

| Name | Description | Remarks |
|---|---|---|
| **Selection according to requestor** | | |
| Random | 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 |

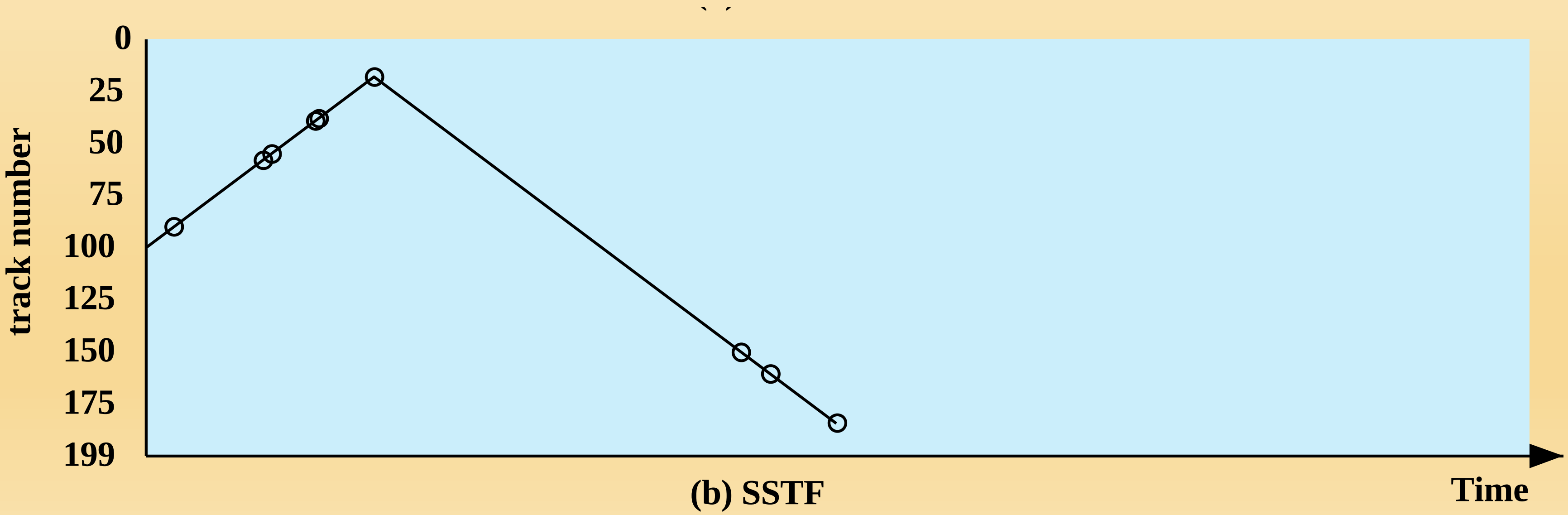
**Table 11.3 Disk Scheduling Algorithms**

# Priority (PRI)

- Control of the scheduling is outside the control of disk management software
- Goal is not to optimize disk utilization but to meet other objectives
- Short batch jobs and interactive jobs are given higher priority
- Provides good interactive response time
- Longer jobs may have to wait an excessively long time
- A poor policy for database systems

# Shortest Service Time First (SSTF)

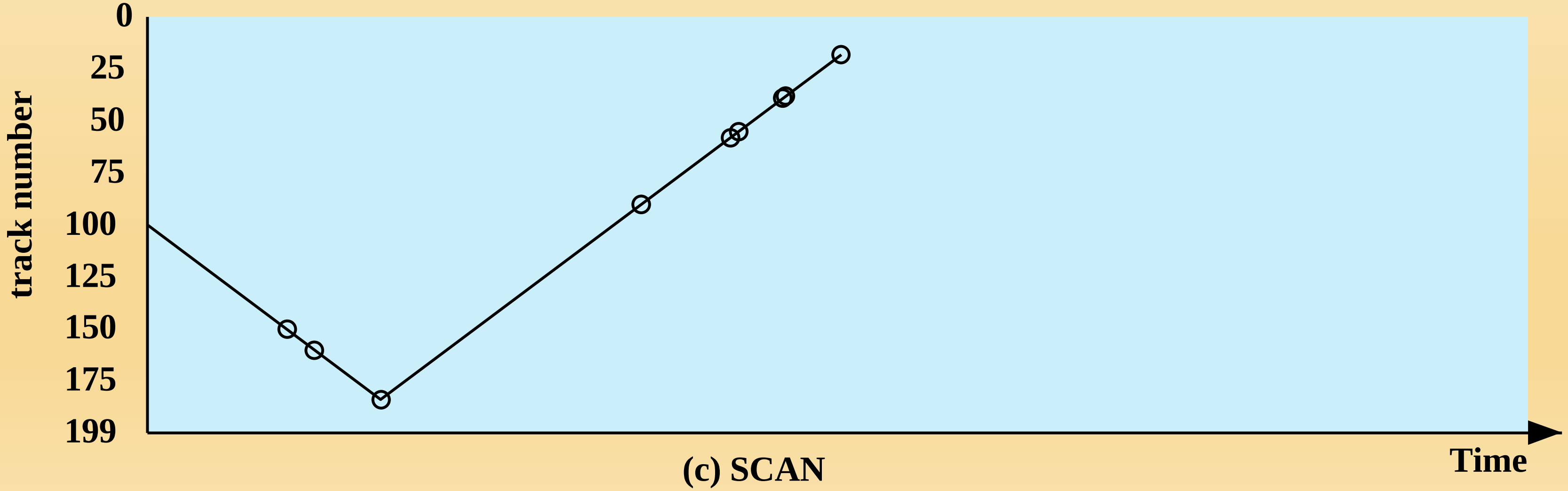
- Select the disk I/O request that requires the least movement of the disk arm from its current position
- Always choose the minimum seek time



(b) SSTF

# SCAN

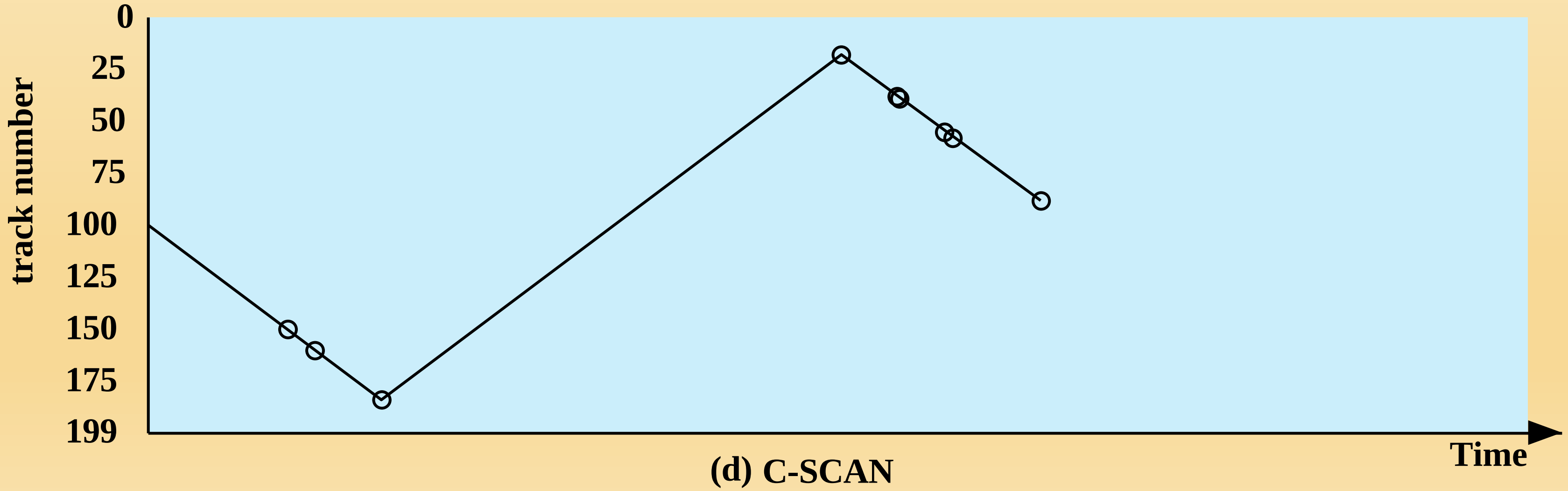
- Also known as the elevator algorithm
- Arm moves in one direction only
  - Satisfies all outstanding requests until it reaches the last track in that direction then the direction is reversed
- Favors jobs whose requests are for tracks nearest to both innermost and outermost tracks and favors the latest-arriving jobs



(c) SCAN

# C-SCAN (Circular 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



(d) C-SCAN

# N-Step-SCAN

With SSTF, SCAN, and C-SCAN, it is possible that the arm may not move for a considerable period of time. For example, if one or a few processes have high access rates to one track, they can monopolize the entire device by repeated requests to that track. High-density multisurface disks are more likely to be affected by this characteristic than lower-density disks and/or disks with only one or two surfaces. To avoid this "arm stickiness," the disk request queue can be segmented, with one segment at a time being processed completely. Two examples of this approach are *N -step-SCAN and FSCAN.*

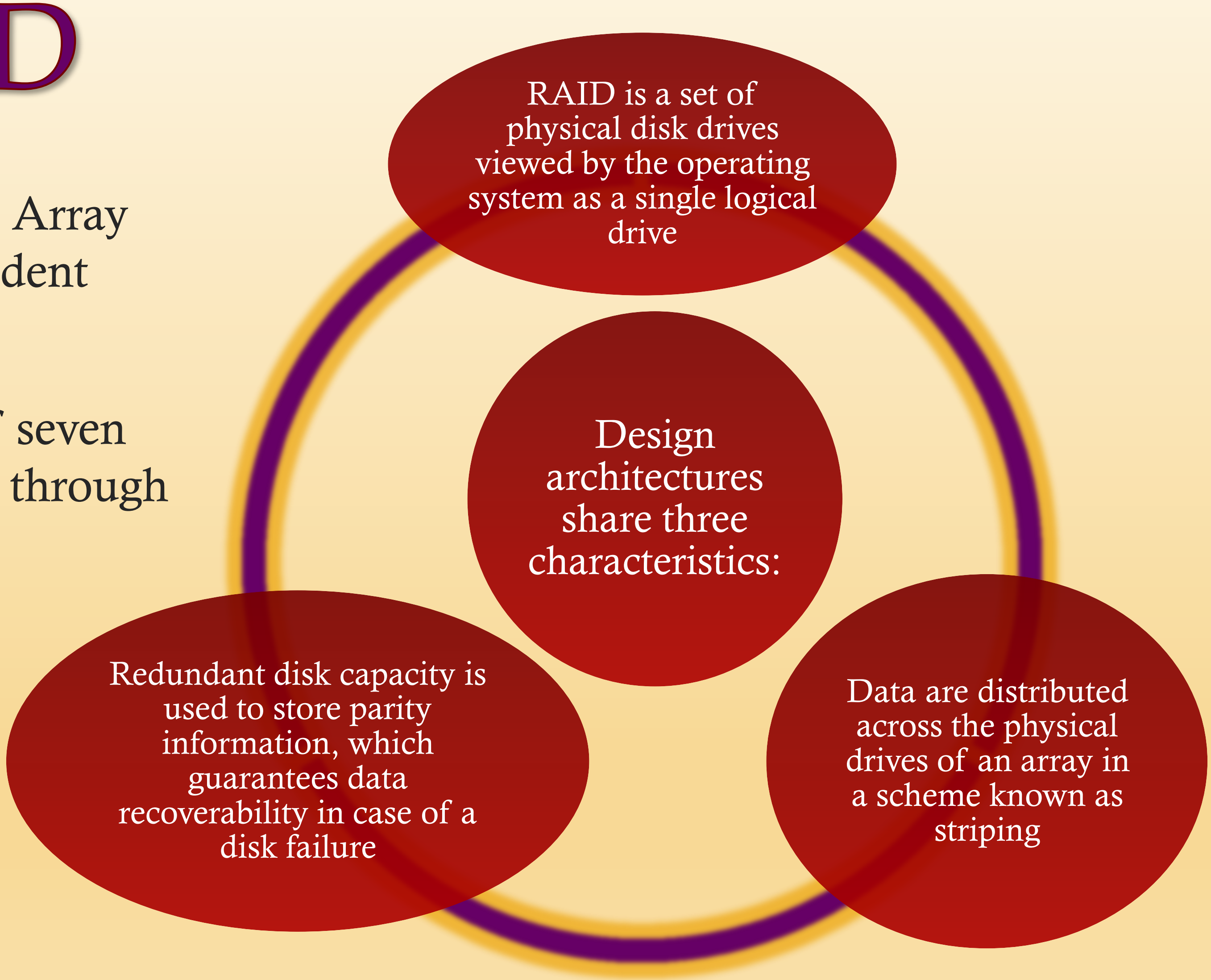
- Segments the disk request queue into subqueues of length *N*
- Subqueues are processed one at a time, using SCAN
- While a queue is being processed new requests must be added to some other queue
- If fewer than *N* requests are available at the end of a scan, all of them are processed with the next scan

# FSCAN

- Uses two subqueues
- When a scan begins, all of the requests are in one of the queues, with the other empty
- During scan, all new requests are put into the other queue
- Service of new requests is deferred until all of the old requests have been processed

# RAID

- Redundant Array of Independent Disks
- Consists of seven levels, zero through six



Design architectures share three characteristics:

RAID is a 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 in a scheme known as striping

Redundant disk capacity is used to store parity information, which guarantees data recoverability in case of a disk failure

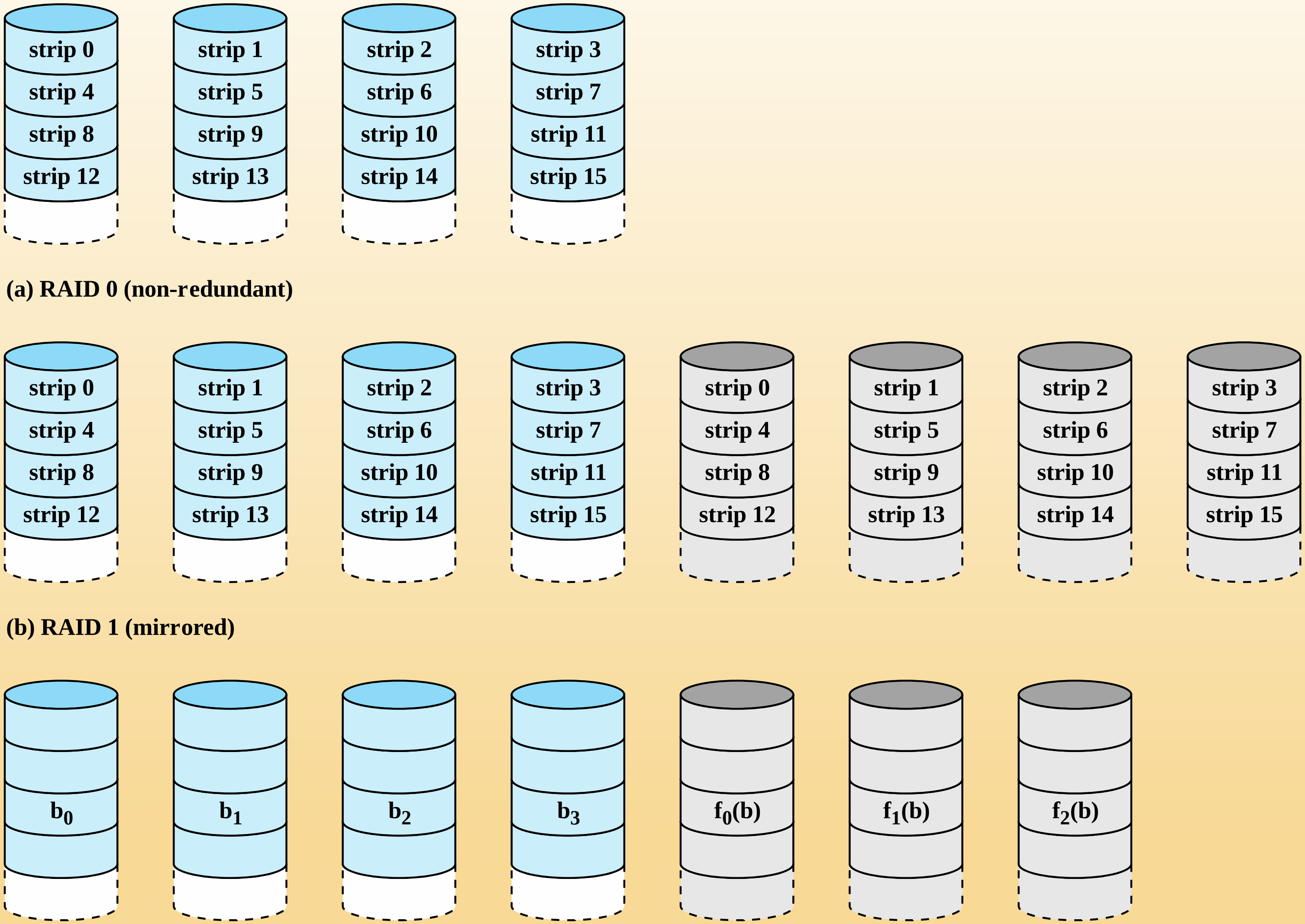
# RAID

- The term was originally coined in a paper by a group of researchers at the University of California at Berkeley
  - The paper outlined various configurations and applications and introduced the definitions of the RAID levels
- Strategy employs multiple disk drives and distributes data in such a way as to enable simultaneous access to data from multiple drives
  - Improves I/O performance and allows easier incremental increases in capacity
- The unique contribution is to address effectively the need for redundancy
- Makes use of stored parity information that enables the recovery of data lost due to a disk failure

| Category | Level | Description | Disks required | Data availability | Large I/O data transfer capacity | Small I/O request rate |
|---|---|---|---|---|---|---|
| Striping | 0 | Nonredundant | *N* | Lower than single disk | Very high | Very high for both read and write |
| Mirroring | 1 | Mirrored | 2*N* | Higher than RAID 2, 3, 4, or 5; lower than RAID 6 | Higher than single disk for read; similar to single disk for write | Up to twice that of a single disk for read; similar to single disk for write |
| Parallel access | 2 | Redundant via Hamming code | *N* + *m* | Much higher than single disk; comparable to RAID 3, 4, or 5 | Highest of all listed alternatives | Approximately twice that of a single disk |
|  | 3 | Bit-interleaved parity | *N* + 1 | Much higher than single disk; comparable to RAID 2, 4, or 5 | Highest of all listed alternatives | Approximately twice that of a single disk |
| Independent access | 4 | Block-interleaved parity | *N* + 1 | Much higher than single disk; comparable to RAID 2, 3, or 5 | Similar to RAID 0 for read; significantly lower than single disk for write | Similar to RAID 0 for read; significantly lower than single disk for write |
|  | 5 | Block-interleaved distributed parity | *N* + 1 | Much higher than single disk; comparable to RAID 2, 3, or 4 | Similar to RAID 0 for read; lower than single disk for write | Similar to RAID 0 for read; generally lower than single disk for write |
|  | 6 | Block-interleaved dual distributed parity | *N* + 2 | Highest of all listed alternatives | Similar to RAID 0 for read; lower than RAID 5 for write | Similar to RAID 0 for read; significantly lower than RAID 5 for write |

*N* = number of data disks; *m* proportional to log *N*

**Table 11.4 RAID Levels** (Page 498 in textbook)

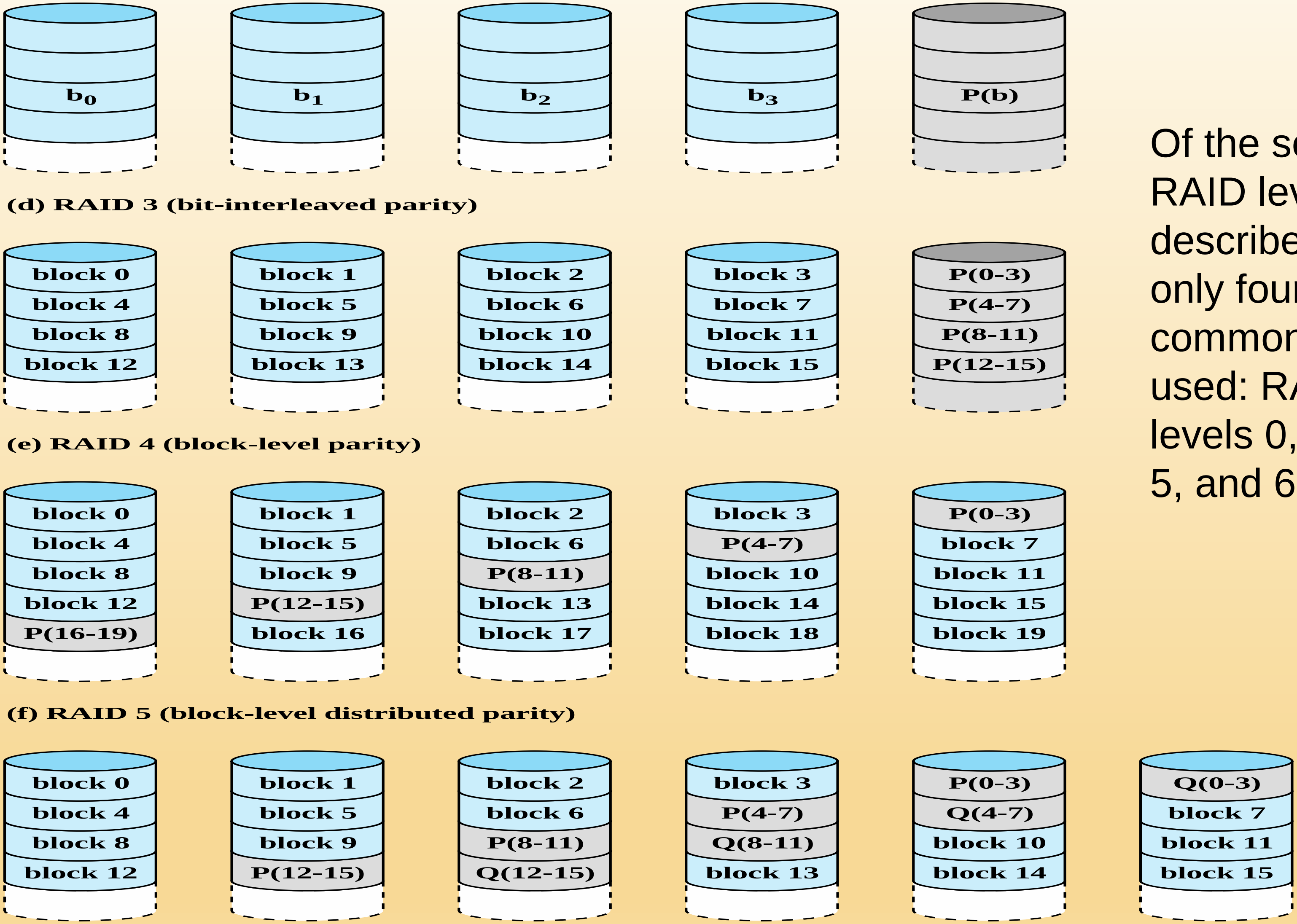


(a) RAID 0 (non-redundant)

(b) RAID 1 (mirrored)

(c) RAID 2 (redundancy through Hamming code)

**Figure 11.8 RAID Levels** (page 1 of 2)



(d) RAID 3 (bit-interleaved parity)

(e) RAID 4 (block-level parity)

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

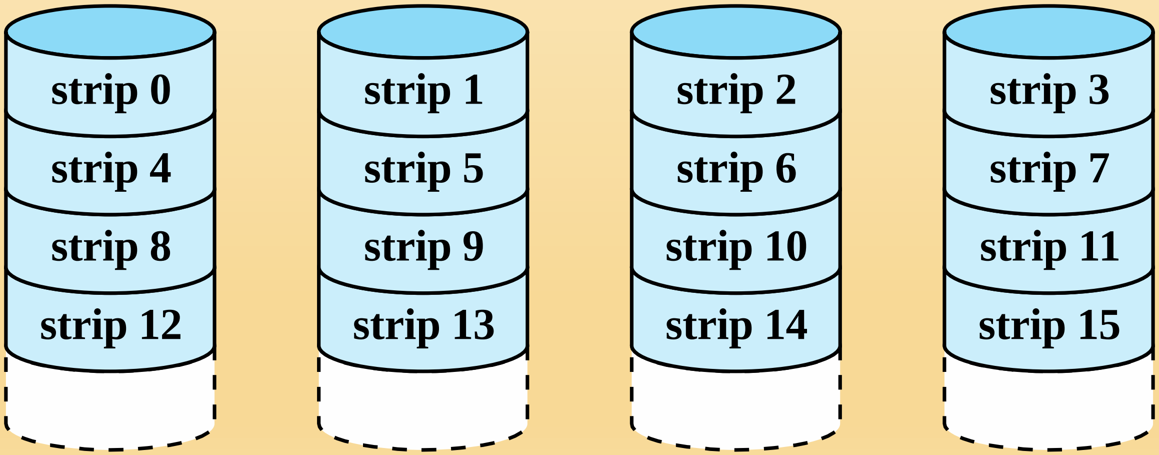
(g) RAID 6 (dual redundancy)

**Figure 11.8 RAID Levels** (page 2 of 2)

Of the seven RAID levels described, only four are commonly used: RAID levels 0, 1, 5, and 6.

# RAID Level 0

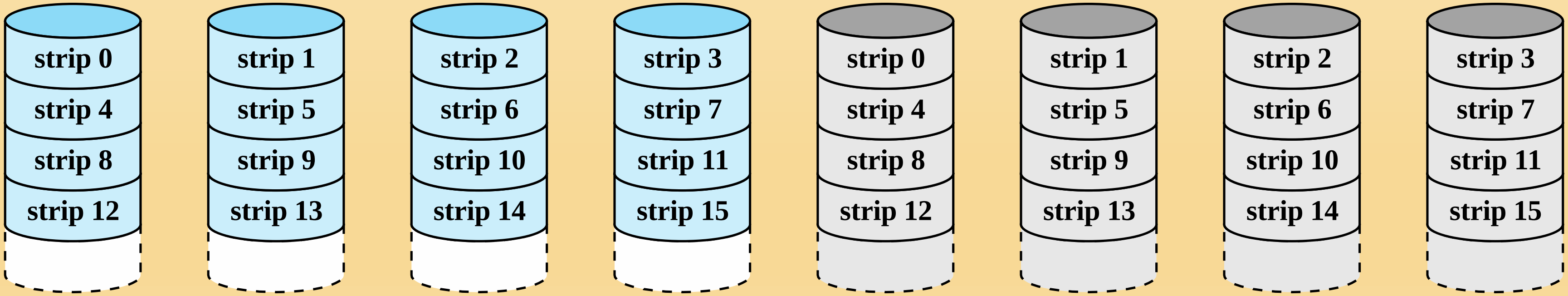
- Not a true RAID because it does not include redundancy to improve performance or provide data protection
- User and system data are distributed across all of the disks in the array
- Logical disk is divided into strips



(a) RAID 0 (non-redundant)

# RAID Level 1

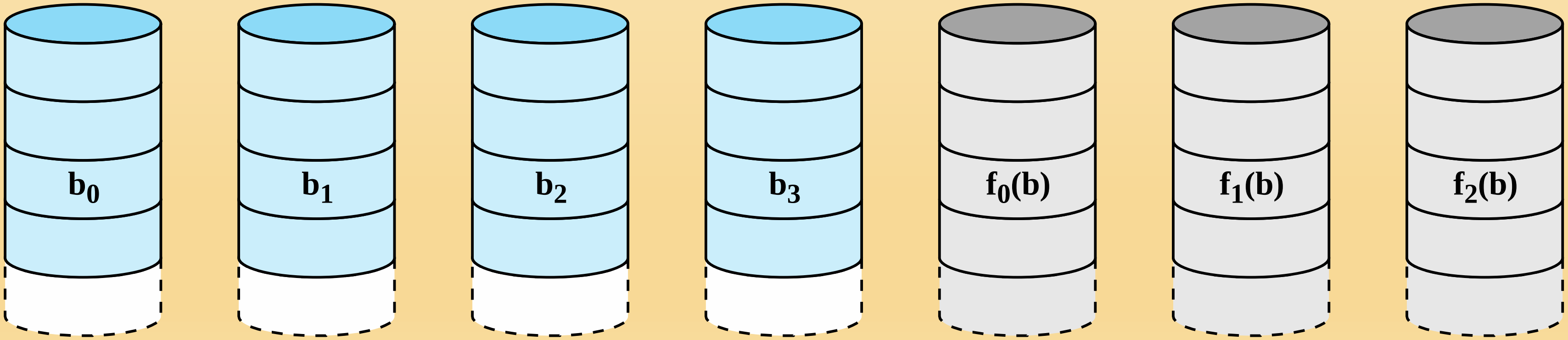
- Redundancy is achieved by the simple expedient of duplicating all the data
- There is no “write penalty”
- When a drive fails the data may still be accessed from the second drive
- Principal disadvantage is the cost



**(b) RAID 1 (mirrored)**

# RAID Level 2

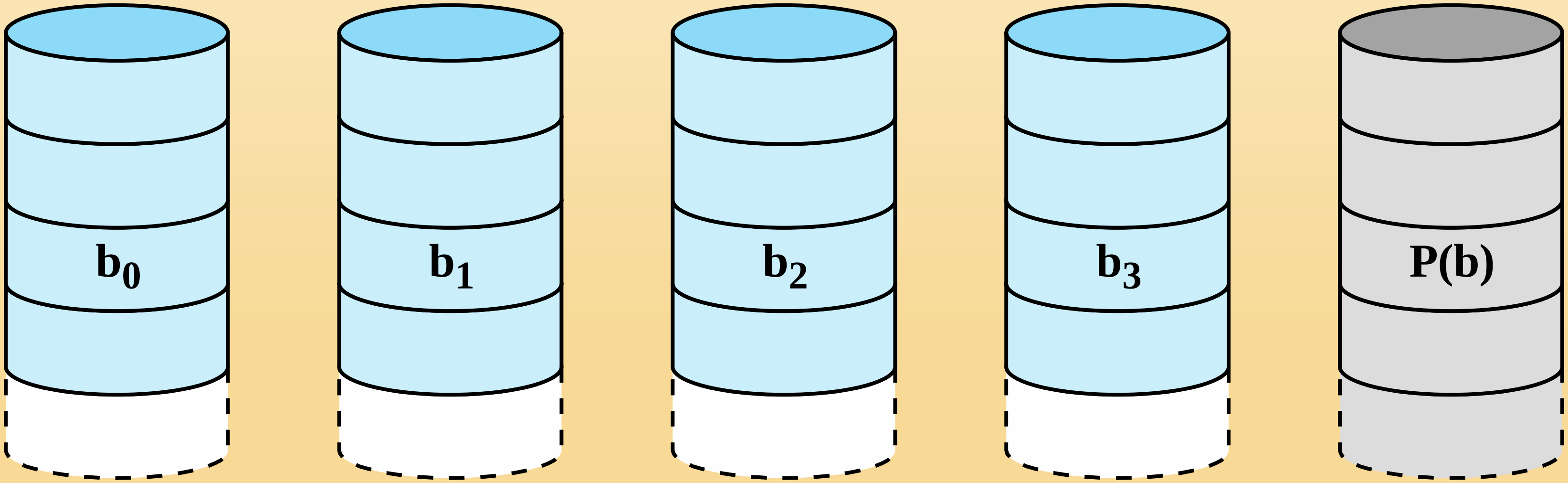
- Makes use of a parallel access technique
- Data striping is used
- Typically a Hamming code is used
- Effective choice in an environment in which many disk errors occur



**(c) RAID 2 (redundancy through Hamming code)**

# RAID Level 3

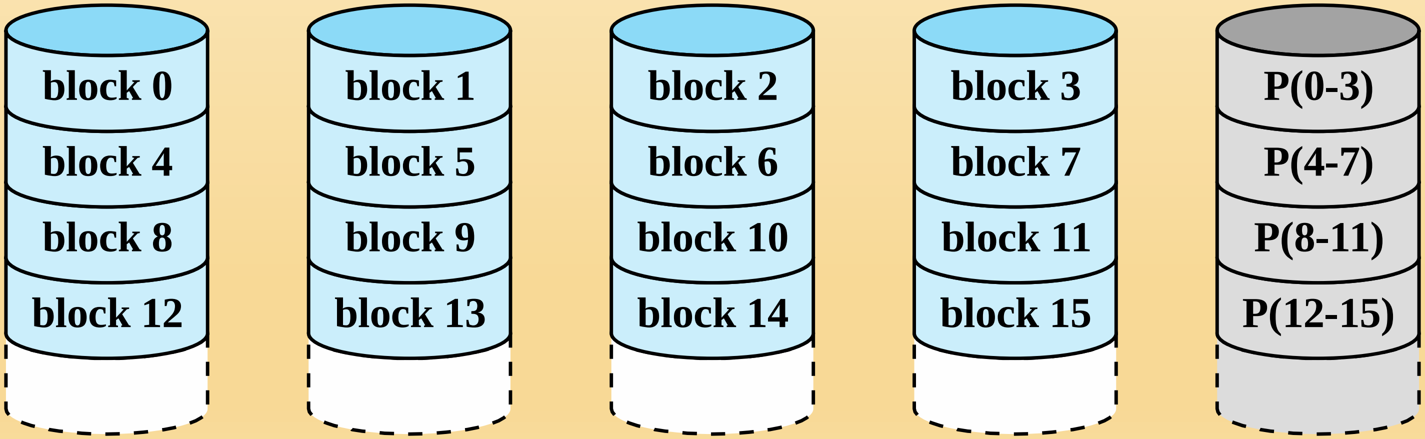
- Requires only a single redundant disk, no matter how large the disk array
- Employs parallel access, with data distributed in small strips
- Can achieve very high data transfer rates



**(d) RAID 3 (bit-interleaved parity)**

# RAID Level 4

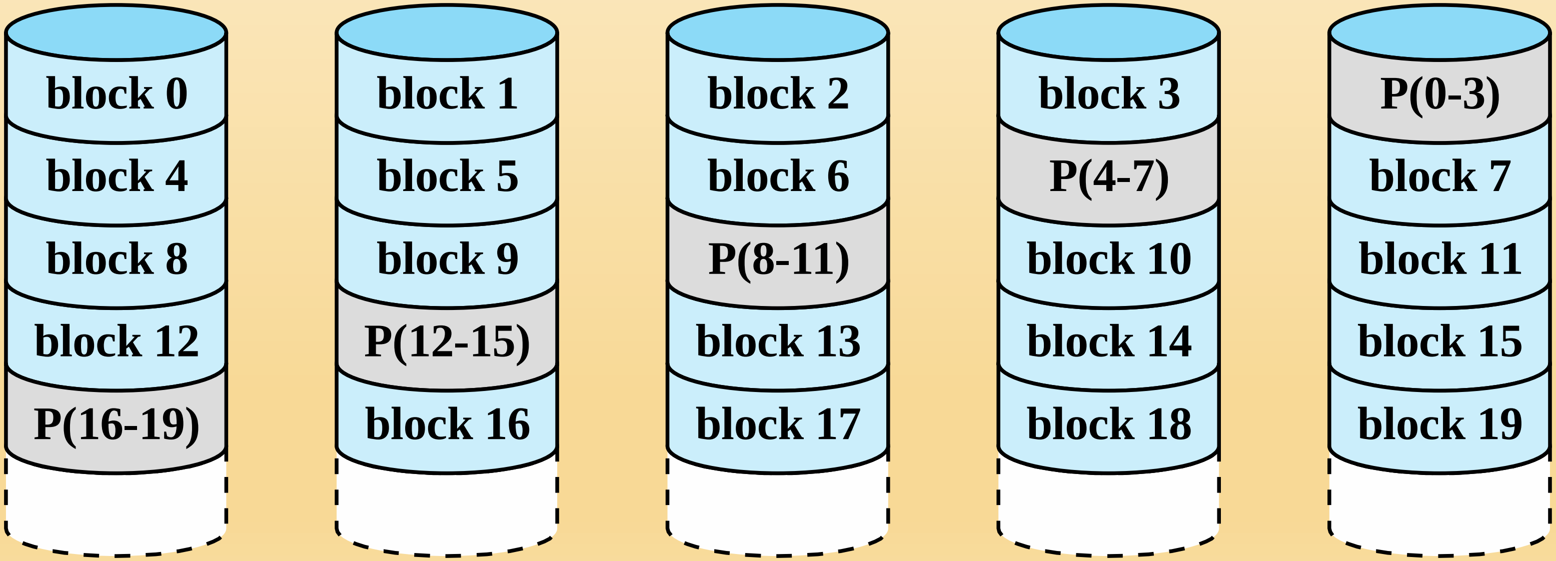
- Makes use of an independent access technique
- A bit-by-bit parity strip is calculated across corresponding strips on each data disk, and the parity bits are stored in the corresponding strip on the parity disk
- Involves a write penalty when an I/O write request of small size is performed



**(e) RAID 4 (block-level parity)**

# RAID Level 5

- Similar to RAID-4 but distributes the parity bits across all disks
- Typical allocation is a round-robin scheme
- Has the characteristic that the loss of any one disk does not result in data loss

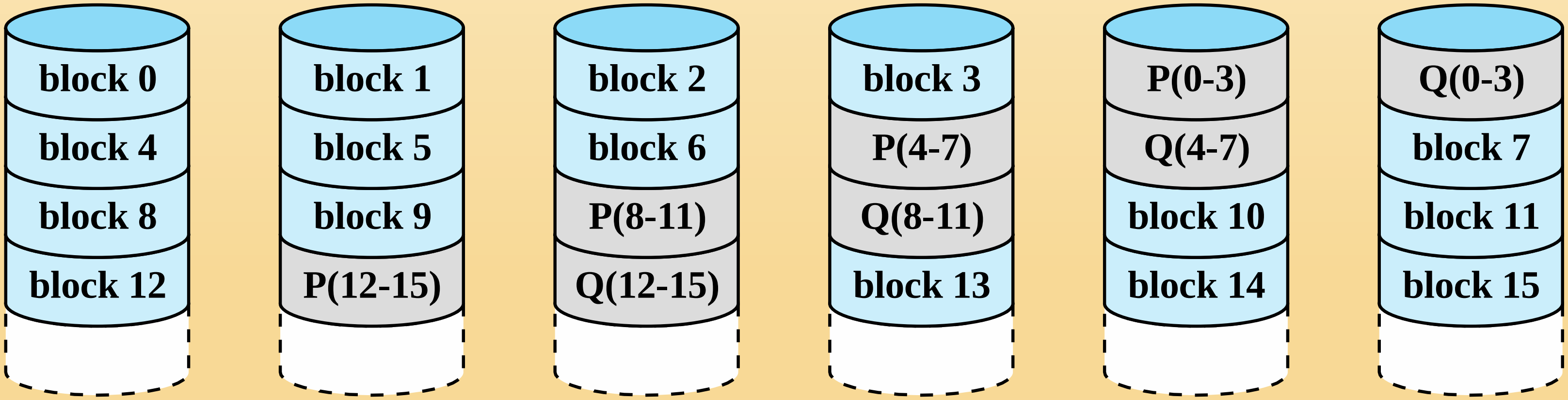


| | | | | |
|---|---|---|---|---|
| block 0 | block 1 | block 2 | block 3 | P(0-3) |
| block 4 | block 5 | block 6 | P(4-7) | block 7 |
| block 8 | block 9 | P(8-11) | block 10 | block 11 |
| block 12 | P(12-15) | block 13 | block 14 | block 15 |
| P(16-19) | block 16 | block 17 | block 18 | block 19 |

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

# RAID Level 6

- Two different parity calculations are carried out and stored in separate blocks on different disks
- Provides extremely high data availability
- Incurs a substantial write penalty because each write affects two parity blocks

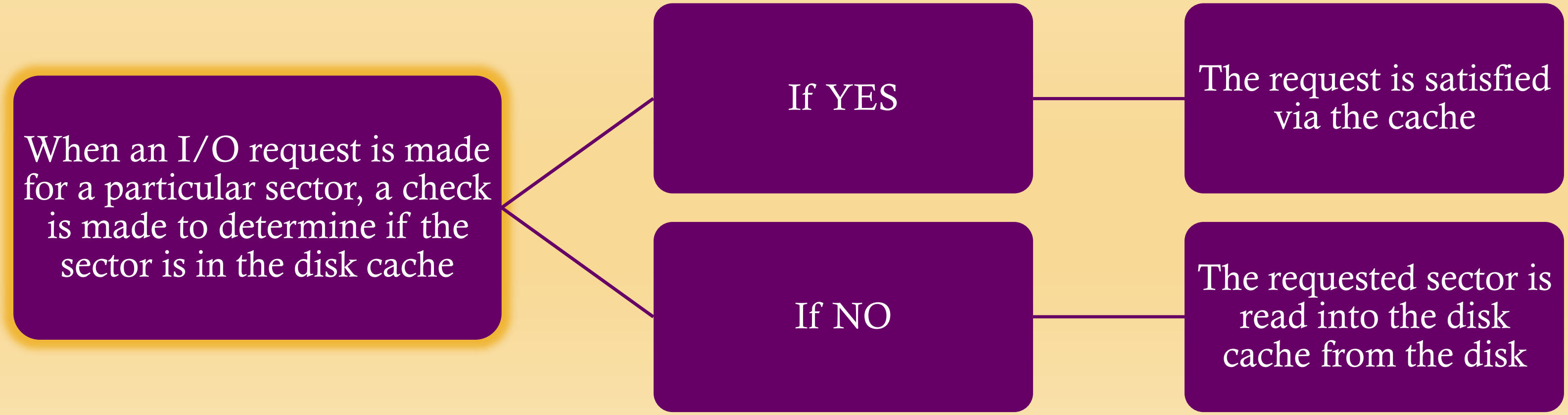


| | | | | | |
|---|---|---|---|---|---|
| block 0 | block 1 | block 2 | block 3 | P(0-3) | Q(0-3) |
| block 4 | block 5 | block 6 | P(4-7) | Q(4-7) | block 7 |
| block 8 | block 9 | P(8-11) | Q(8-11) | block 10 | block 11 |
| block 12 | P(12-15) | Q(12-15) | block 13 | block 14 | block 15 |

(g) RAID 6 (dual redundancy)

# Disk Cache

- *Cache memory* is used to apply to a memory that is smaller and faster than main memory and that is interposed between main memory and the processor
- Reduces average memory access time by exploiting the principle of locality
- *Disk cache* is a buffer in main memory for disk sectors
- Contains a copy of some of the sectors on the disk



When an I/O request is made for a particular sector, a check is made to determine if the sector is in the disk cache

If YES

The request is satisfied via the cache

If NO

The requested sector is read into the disk cache from the disk

# Least Recently Used (LRU)

- Most commonly used algorithm that deals with the design issue of replacement strategy
- The block that has been in the cache the longest with no reference to it is replaced
- A stack of pointers reference the cache
  - Most recently referenced block is on the top of the stack
  - When a block is referenced or brought into the cache, it is placed on the top of the stack

# Least Frequently Used (LFU)

- The block that has experienced the fewest references is replaced
- A counter is associated with each block
- Counter is incremented each time block is accessed
- When replacement is required, the block with the smallest count is selected

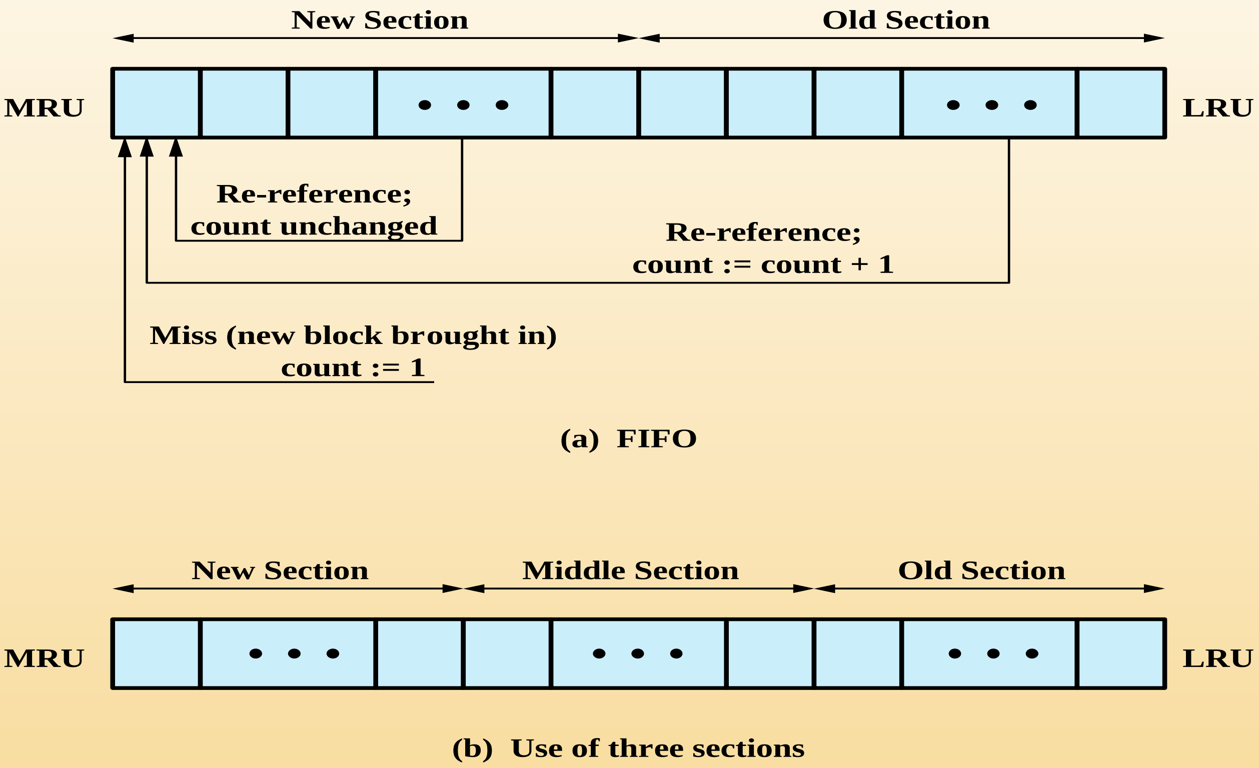


**Figure 11.9 Frequency-Based Replacement**

To overcome the difficulty with LFU, a technique known as frequency-based replacement is proposed in [ROBI90].

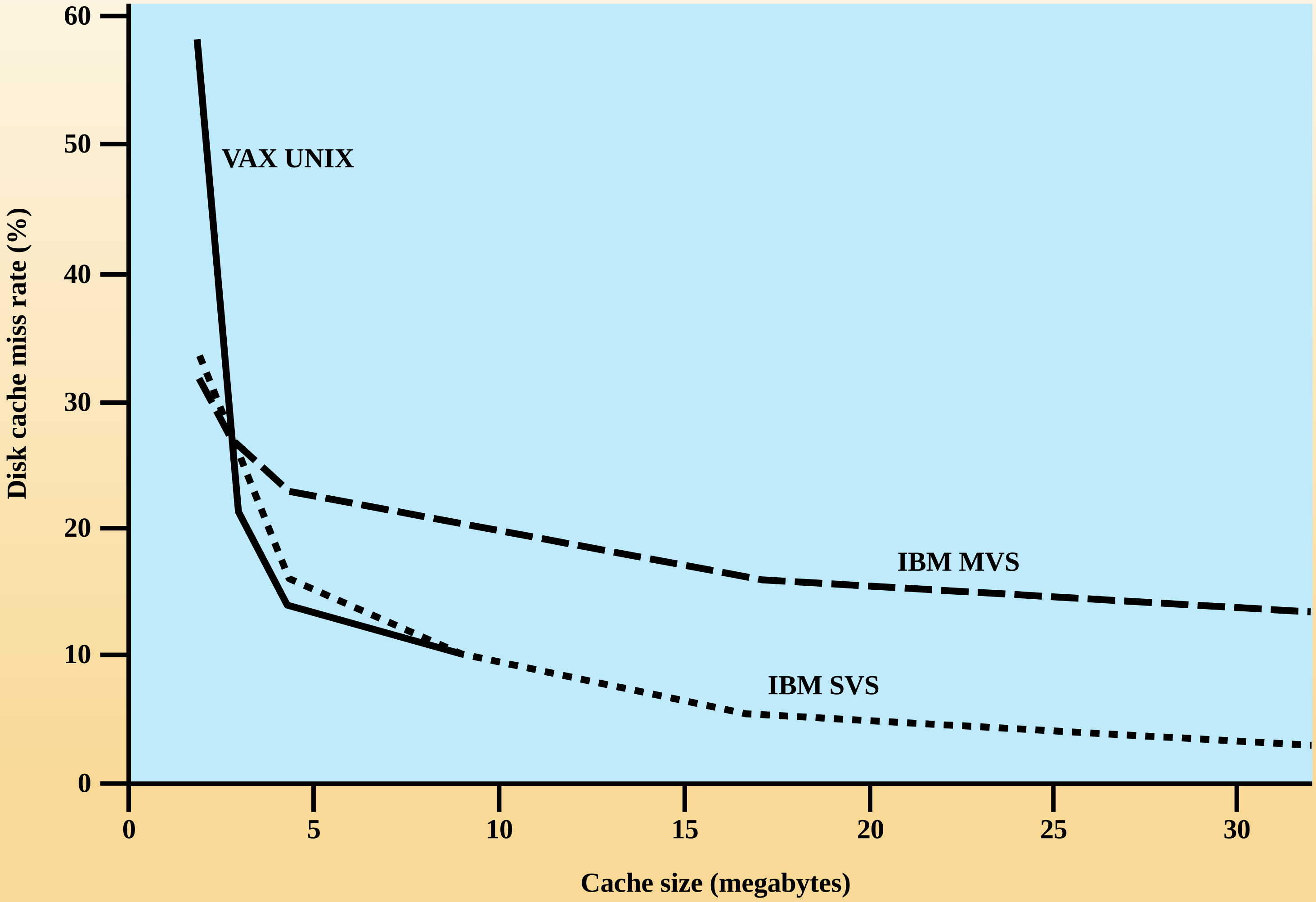


Figure 11.10 Some Disk Cache Performance Results Using LRU

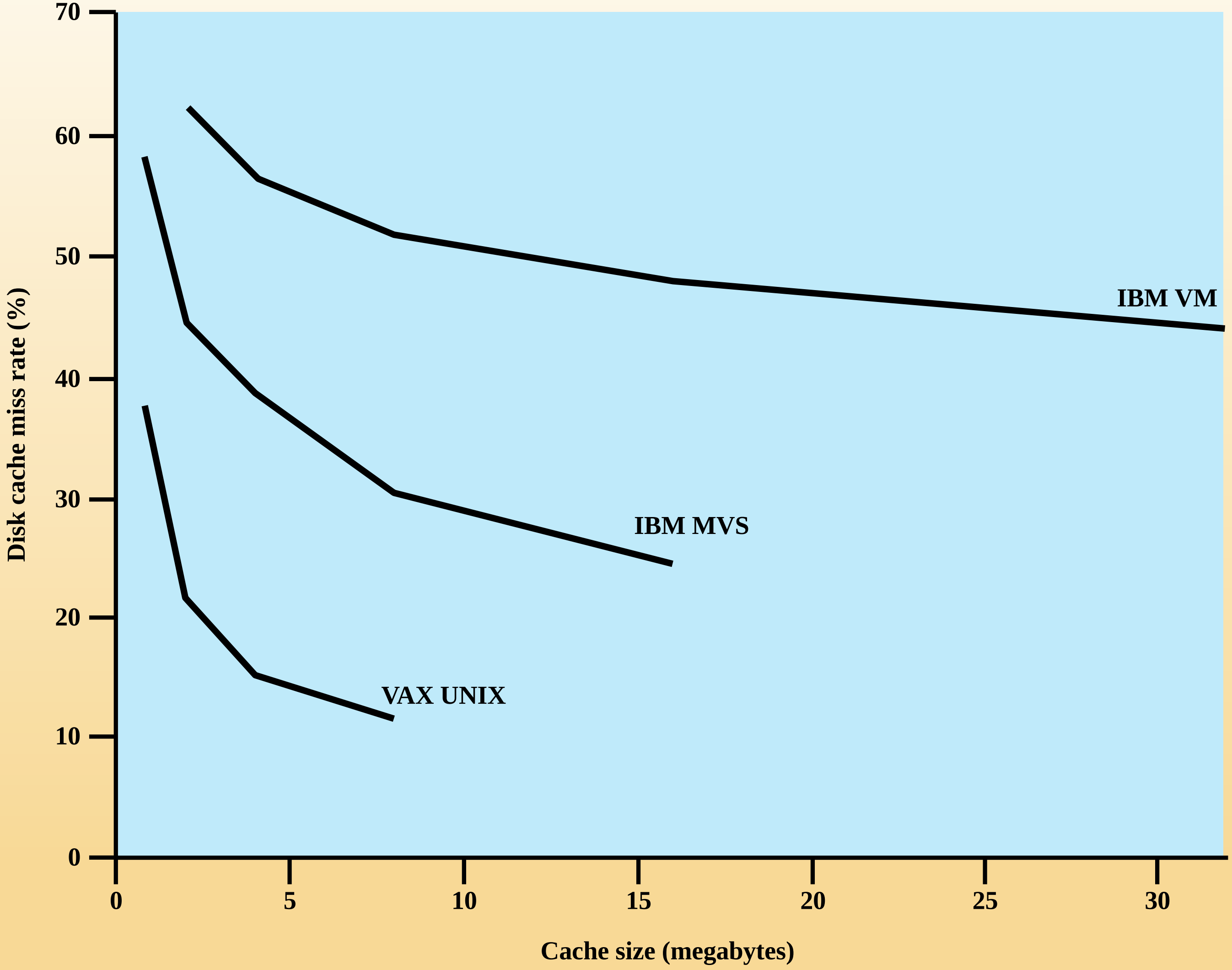


Figure 11.11 Disk Cache Performance Using Fr equency-Based Replacement [ROBI90]

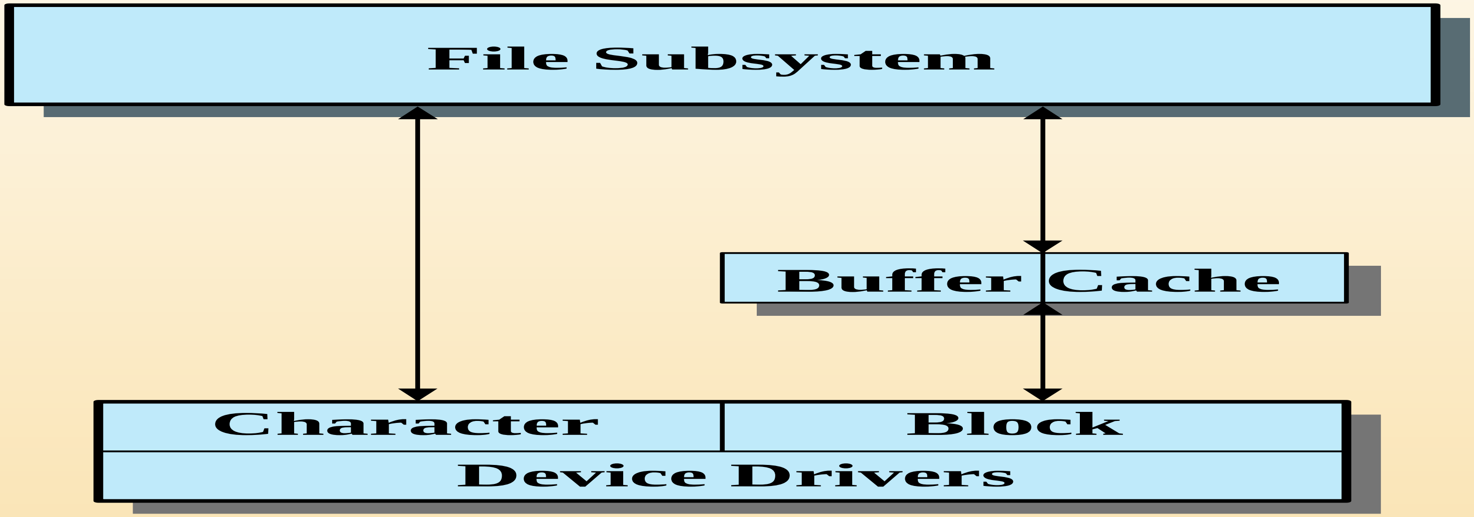


Figure 11.12 UNIX I/O Structure

In UNIX, each individual I/O device is associated with a special file. These are managed by the file system and are read and written in the same manner as user data files. This provides a clean, uniform interface to users and processes. To read from or write to a device, read and write requests are made for the special file associated with the device.

There are two types of I/O in UNIX: buffered and unbuffered. Buffered I/O passes through system buffers, whereas unbuffered I/O typically involves the DMA facility, with the transfer taking place directly between the I/O module and the process I/O area. For buffered I/O, two types of buffers are used: system buffer caches and character queues.

# UNIX Buffer Cache

- Is essentially a disk cache
  - I/O operations with disk are handled through the buffer cache
- The data transfer between the buffer cache and the user process space always occurs using DMA
  - Does not use up any processor cycles
  - Does consume bus cycles
- Three lists are maintained:
  - Free list
    - List of all slots in the cache that are available for allocation
  - Device list
    - List of all buffers currently associated with each disk
  - Driver I/O queue
    - List of buffers that are actually undergoing or waiting for I/O on a particular device

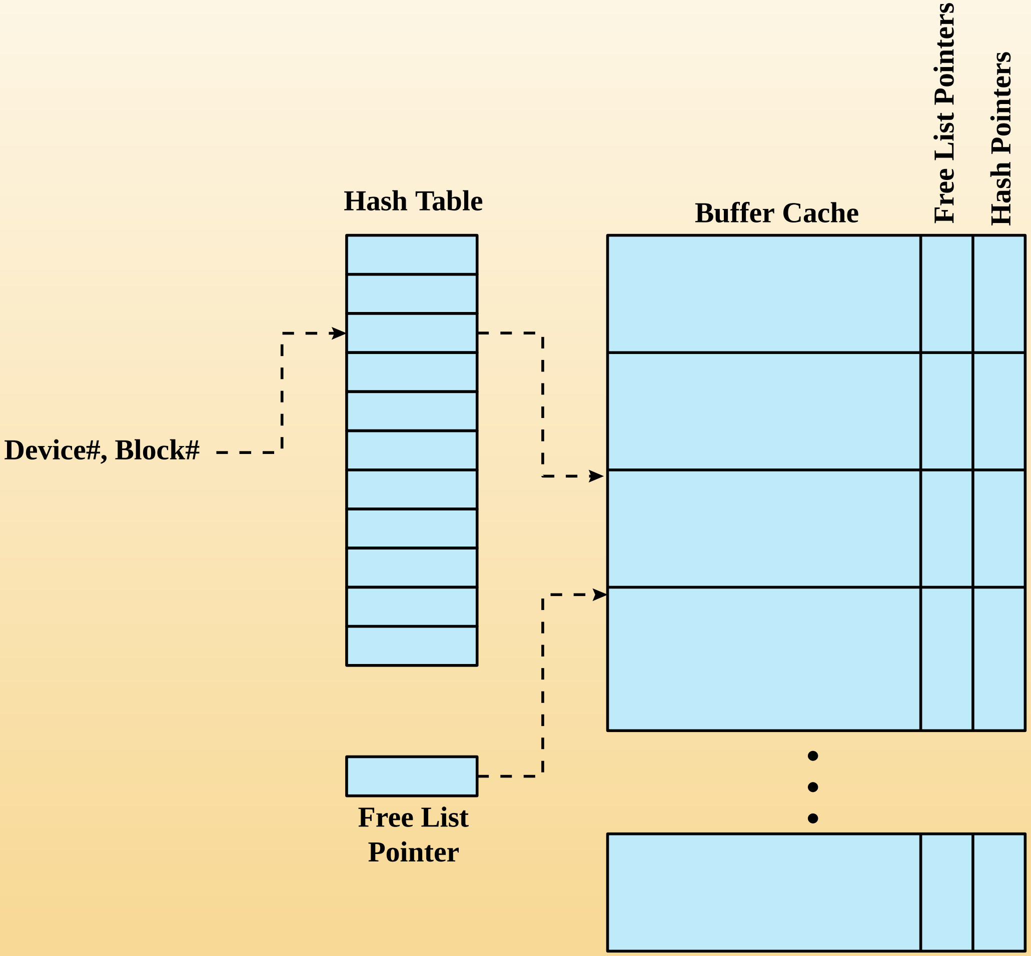
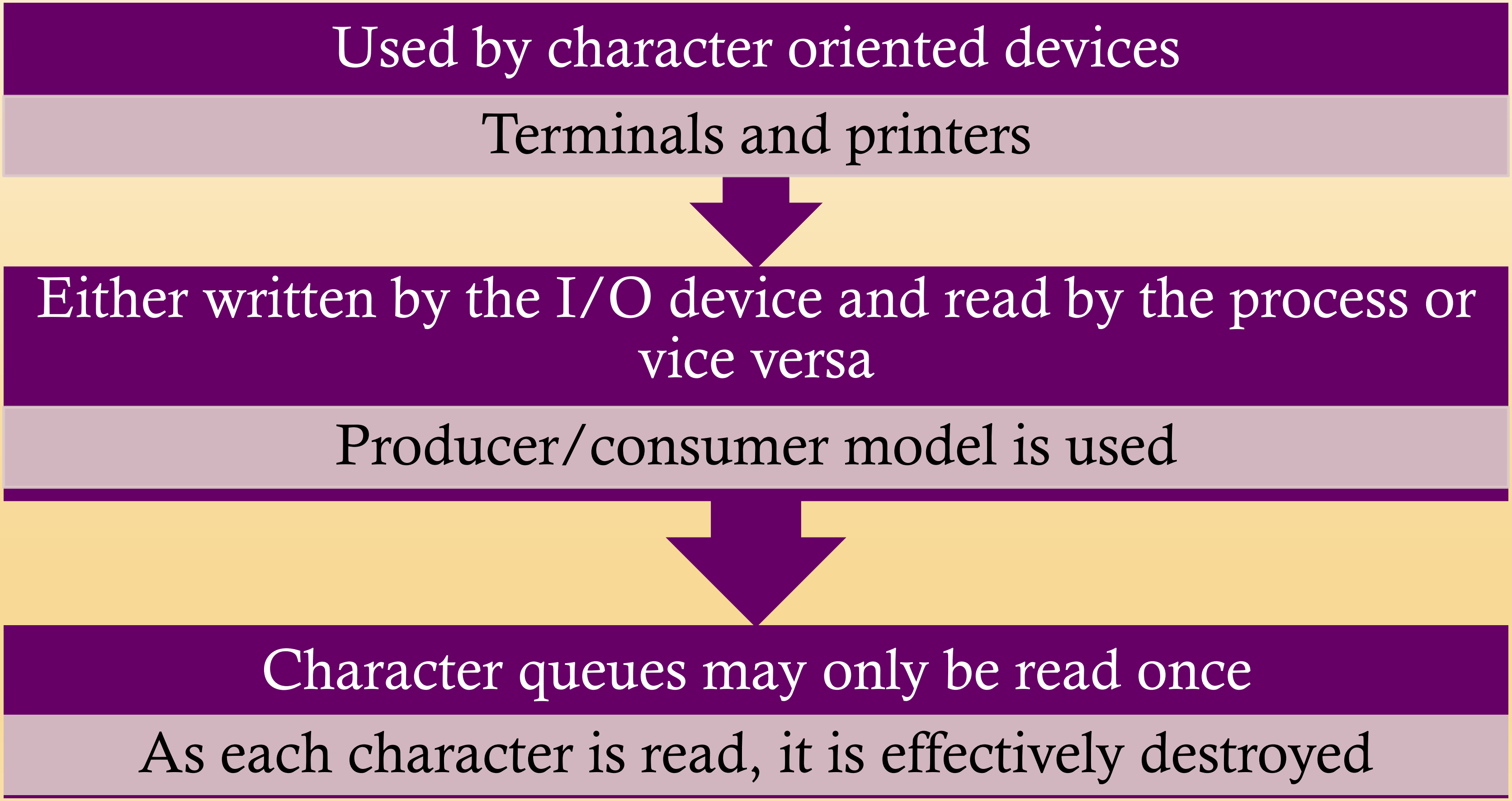


**Figure 11.13 UNIX Buffer Cache Organization**

# Character Queue



Used by character oriented devices

Terminals and printers

Either written by the I/O device and read by the process or vice versa

Producer/consumer model is used

Character queues may only be read once

As each character is read, it is effectively destroyed

# Unbuffered I/O

- Is simply DMA between device and process space
- Is always the fastest method for a process to perform I/O
- Process is locked in main memory and cannot be swapped out
- I/O device is tied up with the process for the duration of the transfer making it unavailable for other processes

|  | Unbuffered I/O | Buffer Cache | Character Queue |
|---|---|---|---|
| Disk drive | X | X |  |
| Tape drive | X | X |  |
| Terminals |  |  | X |
| Communication lines |  |  | X |
| Printers | X |  | X |

**Table 11.5 Device I/O in UNIX :** shows the types of I/O suited to each type of device

# Linux I/O

- Very similar to other UNIX implementation
- Associates a special file with each I/O device driver
- Block, character, and network devices are recognized
- Default disk scheduler in Linux 2.4 is the Linux Elevator

For Linux 2.6 the Elevator algorithm has been augmented by two additional algorithms:

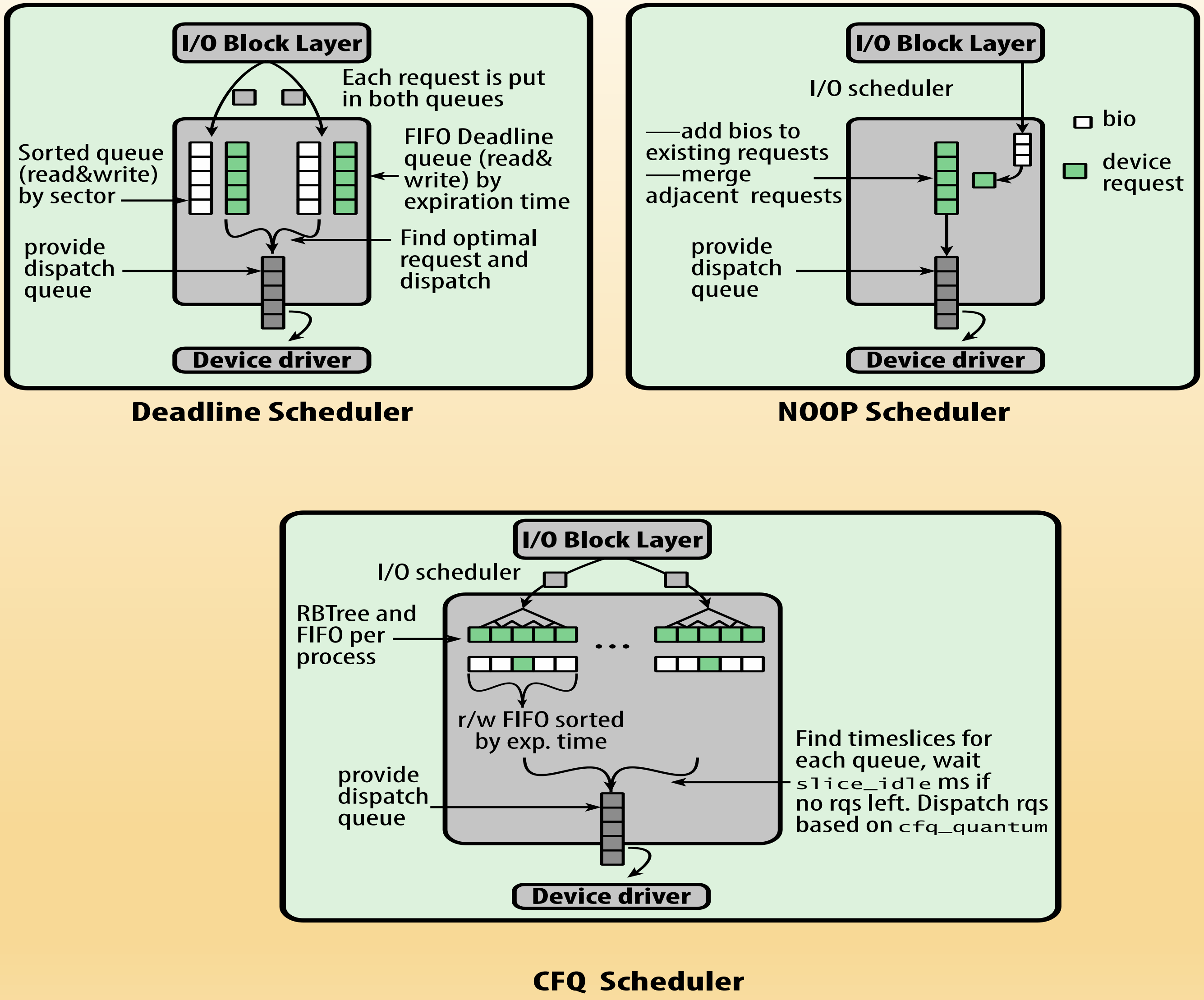
- The deadline I/O scheduler
- The anticipatory I/O scheduler

# The Elevator Scheduler

- Maintains a single queue for disk read and write requests and performs both sorting and merging functions on the queue
- When a new request is added to the queue, four operations are considered in order:
  - If the request is to the same on-disk sector or an immediately adjacent sector to a pending request in the queue, then the existing request and the new request are merged into one request
  - If a request in the queue is sufficiently old, the new request is inserted at the tail of the queue
  - If there is a suitable location, the new request is inserted in sorted order
  - If there is no suitable location, the new request is placed at the tail of the queue

# Deadline Scheduler

- Two problems manifest themselves with the elevator scheme:
  - A distant block request can be delayed for a substantial time because the queue is dynamically updated
  - A stream of write requests can block a read request for a considerable time, and thus block a process
- To overcome these problems, a new deadline I/O scheduler was developed in 2002
  - This scheduler makes use of two pairs of queues
  - In addition to each incoming request being placed in a sorted elevator queue as before, the same request is placed at the tail of a read FIFO queue for a read request or a write FIFO queue for a write request
  - When a request is satisfied, it is removed from the head of the sorted queue and also from the appropriate FIFO queue
    - However, when the item at the head of one of the FIFO queues becomes older than its expiration time, then the scheduler next dispatches from that FIFO queue, taking the expired request, plus the next few requests from the queue
    - As each request is dispatched, it is also removed from the sorted queue



I/O Block Layer

Each request is put in both queues

Sorted queue (read&write) by sector

FIFO Deadline queue (read& write) by expiration time

provide dispatch queue

Find optimal request and dispatch

Device driver

Deadline Scheduler

I/O Block Layer

I/O scheduler

add bios to existing requests

merge adjacent requests

□ bio

🟩 device request

provide dispatch queue

Device driver

NOOP Scheduler

I/O Block Layer

I/O scheduler

RBTree and FIFO per process

. . .

r/w FIFO sorted by exp. time

provide dispatch queue

Find timeslices for each queue, wait `slice_idle` ms if no rqs left. Dispatch rqs based on `cfq_quantum`

Device driver

CFQ Scheduler

**Figure 11.14 Linux I/O Schedulers**

# Anticipatory I/O Scheduler

- Elevator and deadline scheduling can be counterproductive if there are numerous synchronous read requests

- In Linux, the anticipatory scheduler is superimposed on the deadline scheduler

- When a read request is dispatched, the anticipatory scheduler causes the scheduling system to delay
  - There is a good chance that the application that issued the last read request will issue another read request to the same region of the disk
    - That request will be serviced immediately
    - Otherwise the scheduler resumes using the deadline scheduling algorithm

# The NOOP Scheduler

- This is the simplest among Linux I/O schedulers
- It is a minimal scheduler that inserts I/O requests into a FIFO queue and uses merging
- Its main uses include nondisk-based block devices such as memory devices, and specialized software or hardware environments that do their own scheduling and need only minimal support in the kernel

# Completely Fair Queuing I/O Scheduler (CFQ)

- Was developed in 2003
- Is the default I/O scheduler in Linux
- The CFQ scheduler guarantees a fair allocation of the disk I/O bandwidth among all processes
- It maintains per process I/O queues
  - Each process is assigned a single queue
  - Each queue has an allocated timeslice
  - Requests are submitted into these queues and are processed in round robin
- When the scheduler services a specific queue, and there are no more requests in that queue, it waits in idle mode for a predefined time interval for new requests, and if there are no requests, it continues to the next queue

# Linux Page Cache

- For Linux 2.4 and later there is a single unified page cache for all traffic between disk and main memory
- Benefits:
    - When it is time to write back dirty pages to disk, a collection of them can be ordered properly and written out efficiently
    - Because of the principle of temporal locality, pages in the page cache are likely to be referenced again before they are flushed from the cache, thus saving a disk I/O operation

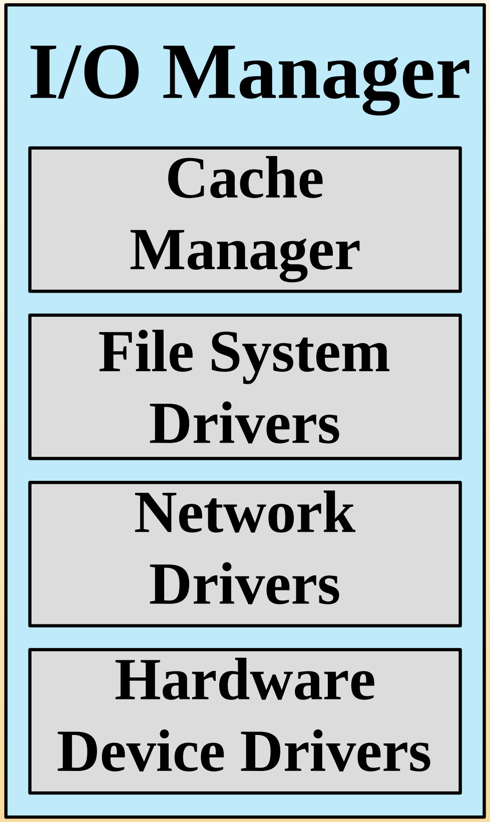
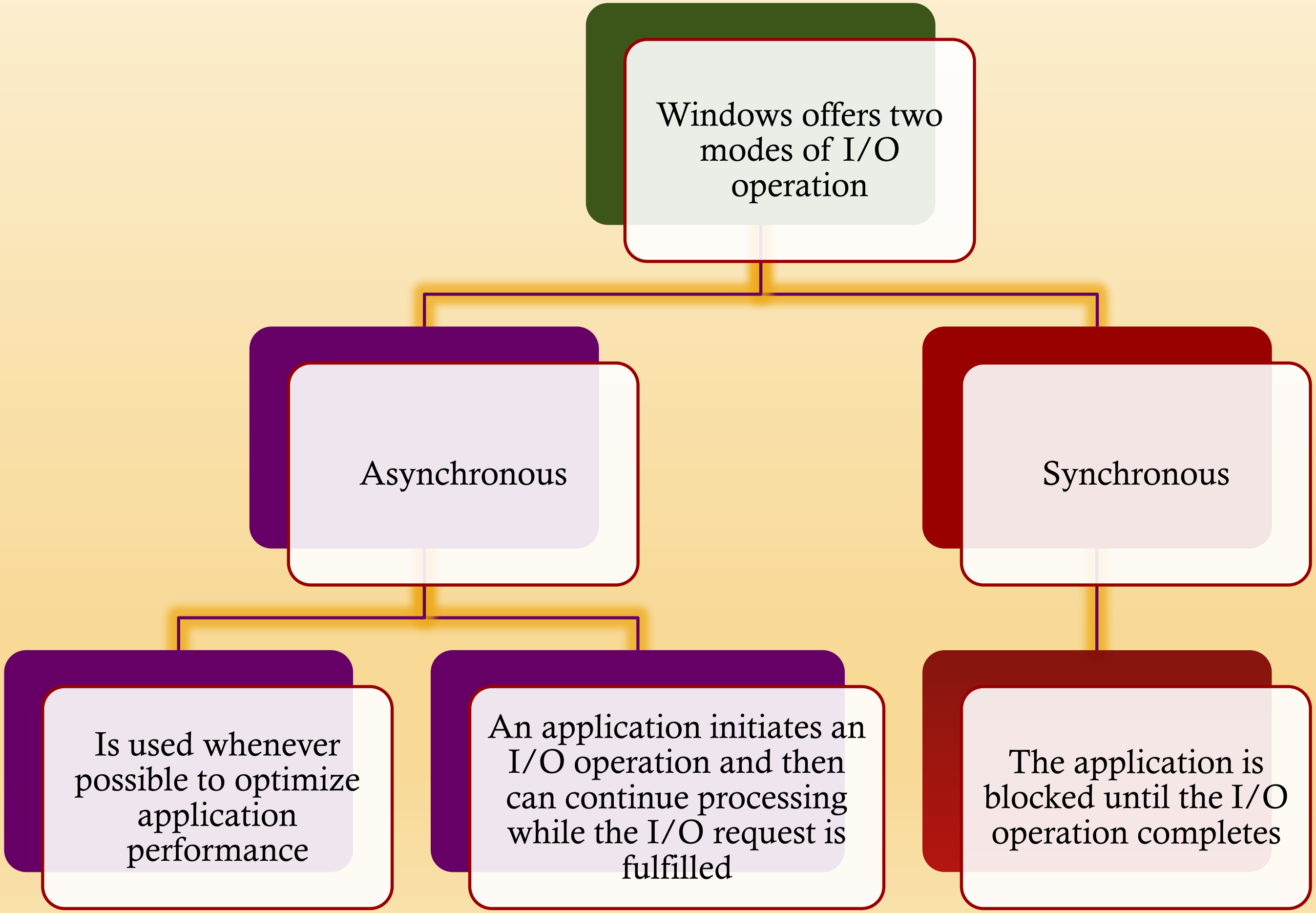


Figure 11.15 Windows I/O Manager

# Basic I/O Facilities

- **Cache Manager**
  - Maps regions of files into kernel virtual memory and then relies on the virtual memory manager to copy pages to and from the files on disk
- **File System Drivers**
  - Sends I/O requests to the software drivers that manage the hardware device adapter
- **Network Drivers**
  - Windows includes integrated networking capabilities and support for remote file systems
  - The facilities are implemented as software drivers
- **Hardware Device Drivers**
  - The source code of Windows device drivers is portable across different processor types

# Asynchronous and Synchronous I/O



Windows offers two modes of I/O operation

Asynchronous

Is used whenever possible to optimize application performance

An application initiates an I/O operation and then can continue processing while the I/O request is fulfilled

Synchronous

The application is blocked until the I/O operation completes

# I/O Completion

- Windows provides five different techniques for signaling I/O completion:

1 • Signaling the file object

2 • Signaling an event object

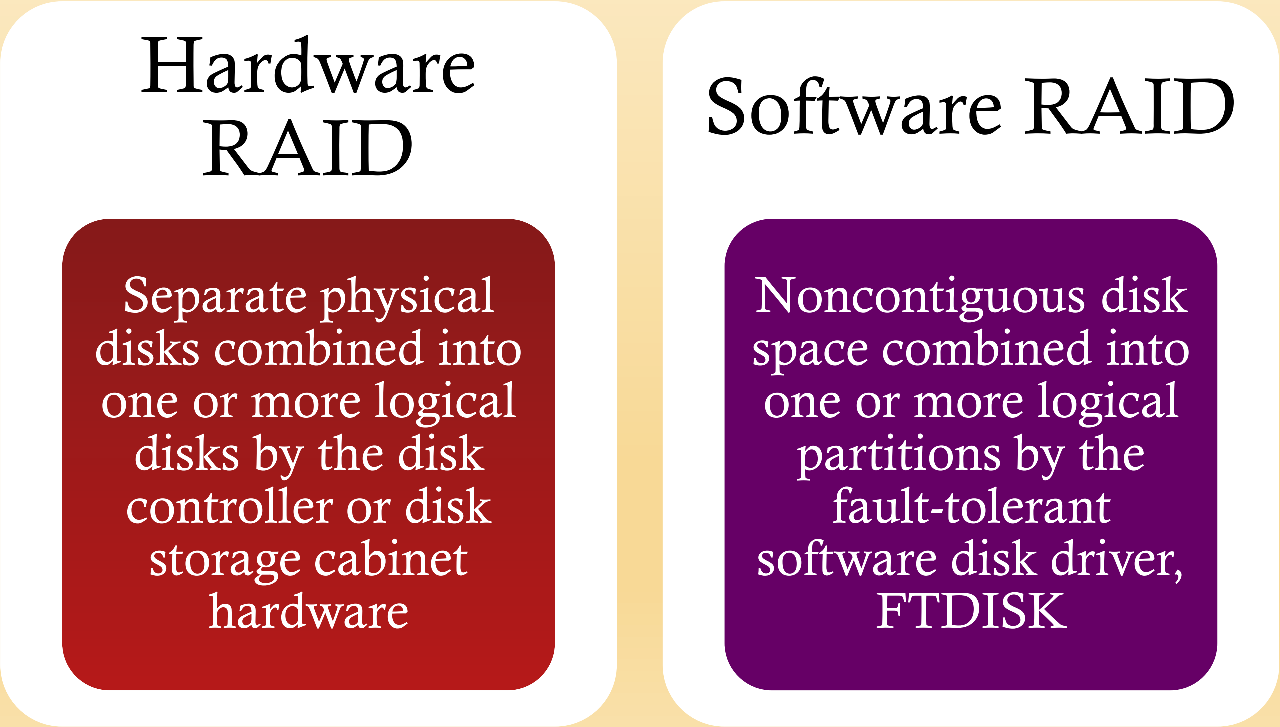
3 • Asynchronous procedure call

4 • I/O completion ports

5 • Polling

# Windows RAID Configurations

- Windows supports two sorts of RAID configurations:



Hardware RAID

Separate physical disks combined into one or more logical disks by the disk controller or disk storage cabinet hardware

Software RAID

Noncontiguous disk space combined into one or more logical partitions by the fault-tolerant software disk driver, FTDISK

# Volume Shadow Copies and Volume Encryption

- **Volume Shadow Copies**
  - Efficient way of making consistent snapshots of volumes so they can be backed up
  - Also useful for archiving files on a per-volume basis
  - Implemented by a software driver that makes copies of data on the volume before it is overwritten

- **Volume Encryption**
  - Windows uses BitLocker to encrypt entire volumes
  - More secure than encrypting individual files
  - Allows multiple interlocking layers of security

# Summary

- I/O devices
- Organization of the I/O function
  - The evolution of the I/O function
  - Direct memory access
- Operating system design issues
  - Design objectives
  - Logical structure of the I/O function
- I/O Buffering
  - Single/double/circular buffer
  - The utility of buffering
- Disk scheduling
  - Disk performance parameters
  - Disk scheduling policies
- Raid
- Raid levels 0 – 6
- Disk cache
  - Design and performance considerations
- UNIX SVR4 I/O
  - Buffer cache
  - Character queue
  - Unbuffered I/O
  - UNIX devices
- Linux I/O
  - Disk scheduling
  - Linux page cache
- Windows I/O
  - Basic I/O facilities
  - Asynchronous and Synchronous I/O
  - Software RAID
  - Volume shadow copies/encryption