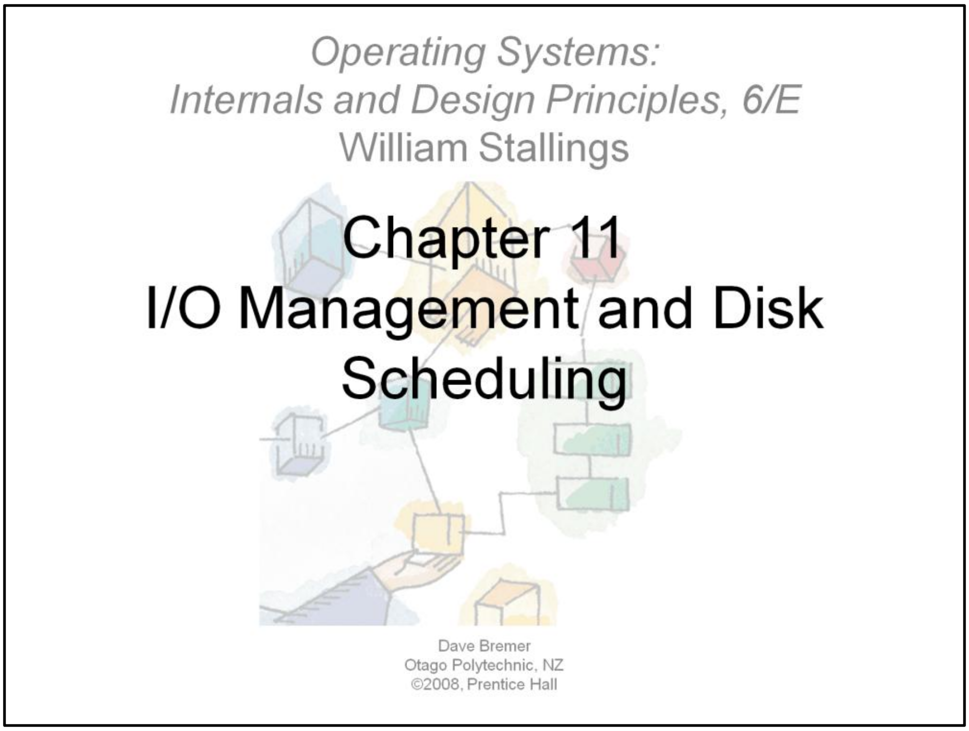
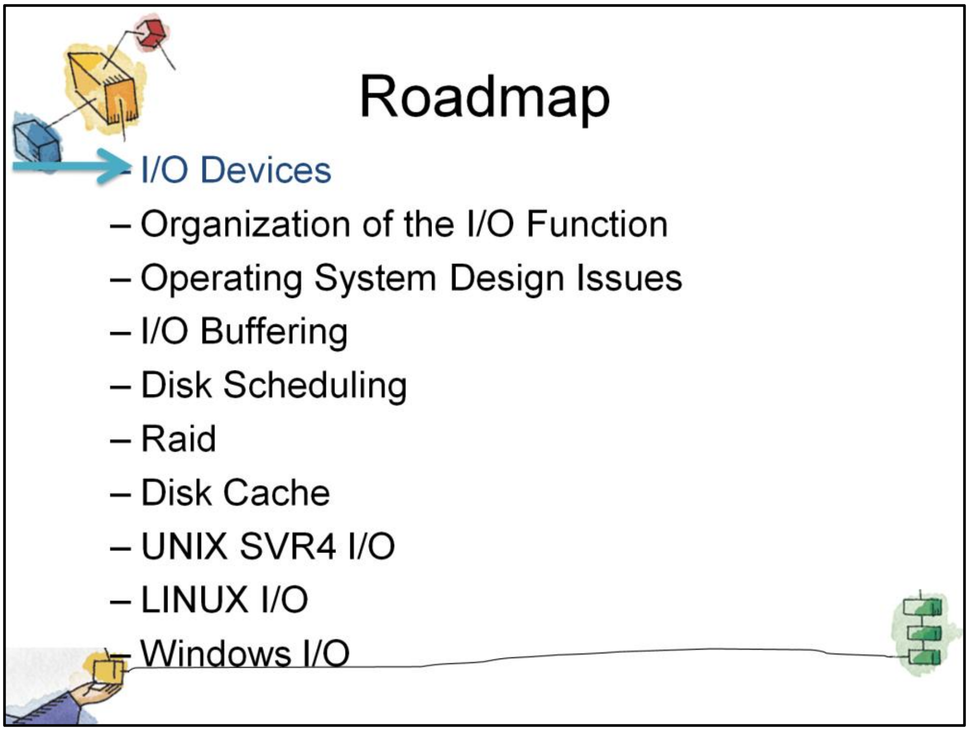


Operating Systems:
Internals and Design Principles, 6/E
William Stallings

Chapter 11
I/O Management and Disk Scheduling

Dave Bremer
Otago Polytechnic, NZ

These slides are intended to help a teacher develop a presentation. This PowerPoint covers the entire chapter and includes too many slides for a single delivery. Professors are encouraged to adapt this presentation in ways which are best suited for their students and environment.



Roadmap

- I/O Devices
- Organization of the I/O Function
- Operating System Design Issues
- I/O Buffering
- Disk Scheduling
- Raid
- Disk Cache
- UNIX SVR4 I/O
- LINUX I/O
- Windows I/O

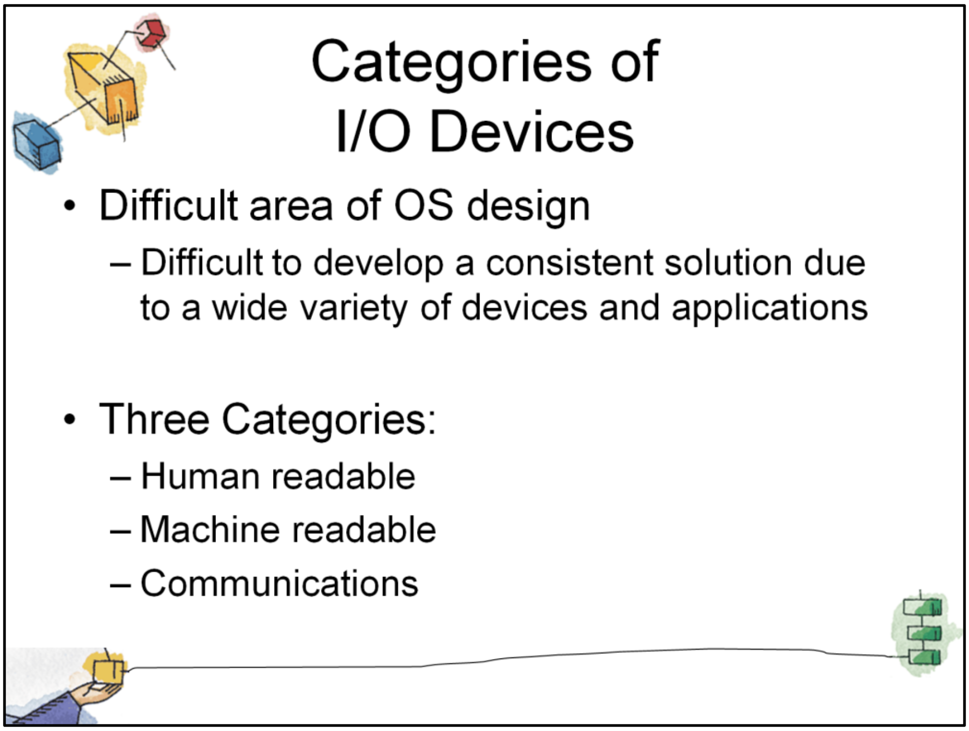
Beginning with a brief discussion of I/O devices and the organization of the I/O functions.

Next examine operating system design issues, including design objectives, and the way in which the I/O function can be structured.

Then I/O buffering is examined;

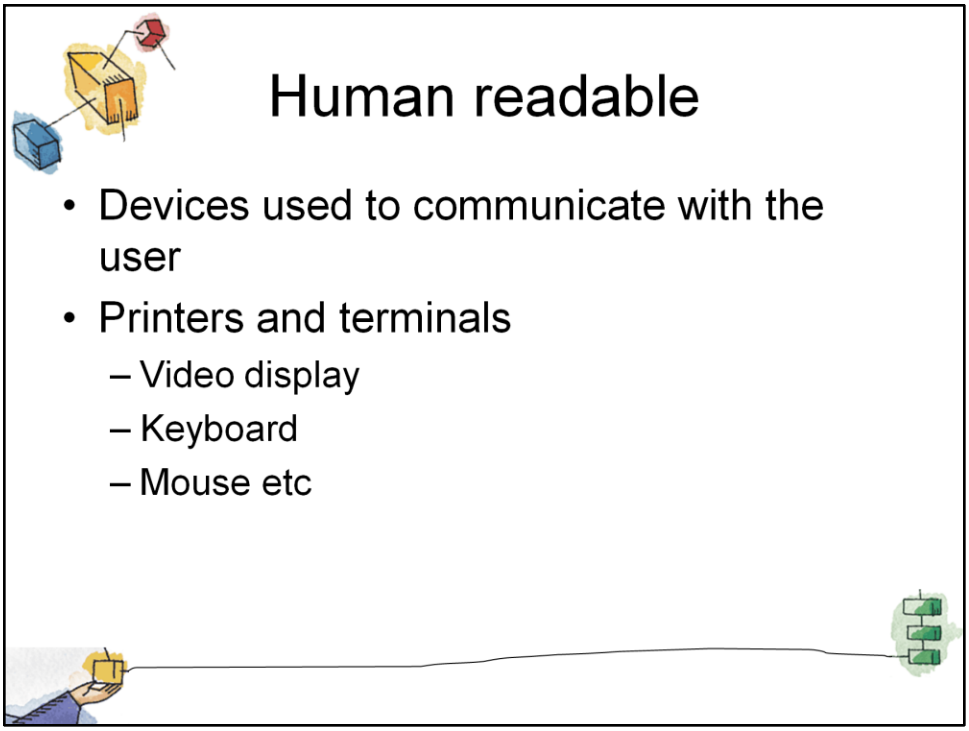
The next sections of the chapter are devoted to magnetic disk I/O.

- We begin by developing a model of disk I/O performance and then examine several techniques that can be used to enhance performance.



Categories of I/O Devices

- Difficult area of OS design
  - Difficult to develop a consistent solution due to a wide variety of devices and applications
- Three Categories:
  - Human readable
  - Machine readable
  - Communications

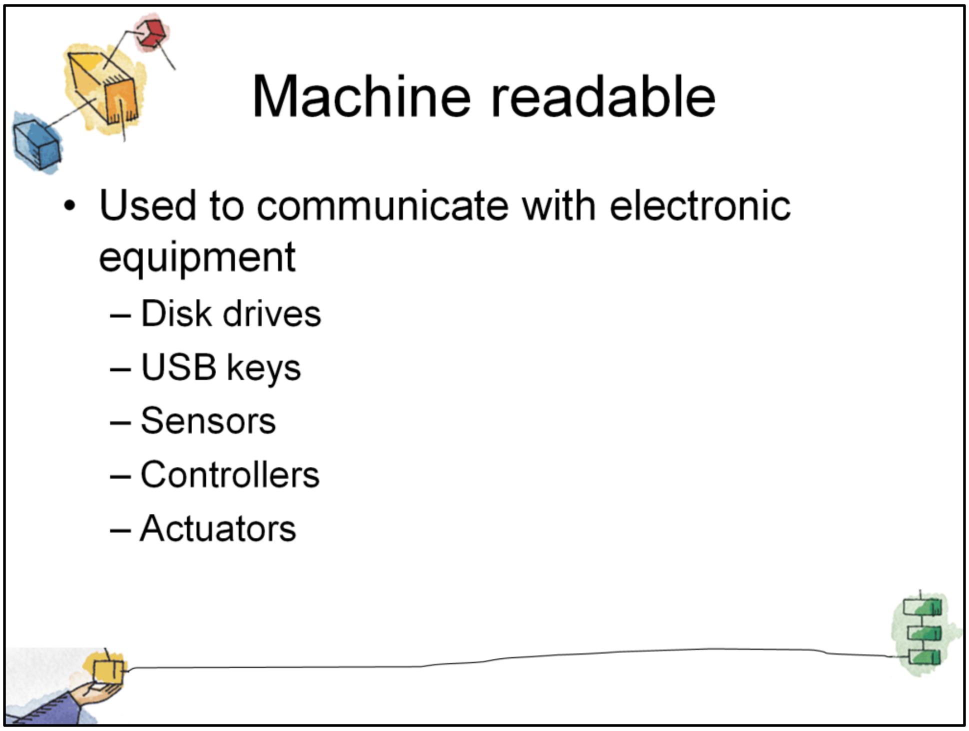


Human readable

- Devices used to communicate with the user
- Printers and terminals
  - Video display
  - Keyboard
  - Mouse etc

Suitable for communicating with the computer user.

- Examples include printers and terminals, the latter consisting of video display, keyboard, and perhaps other devices such as a mouse.

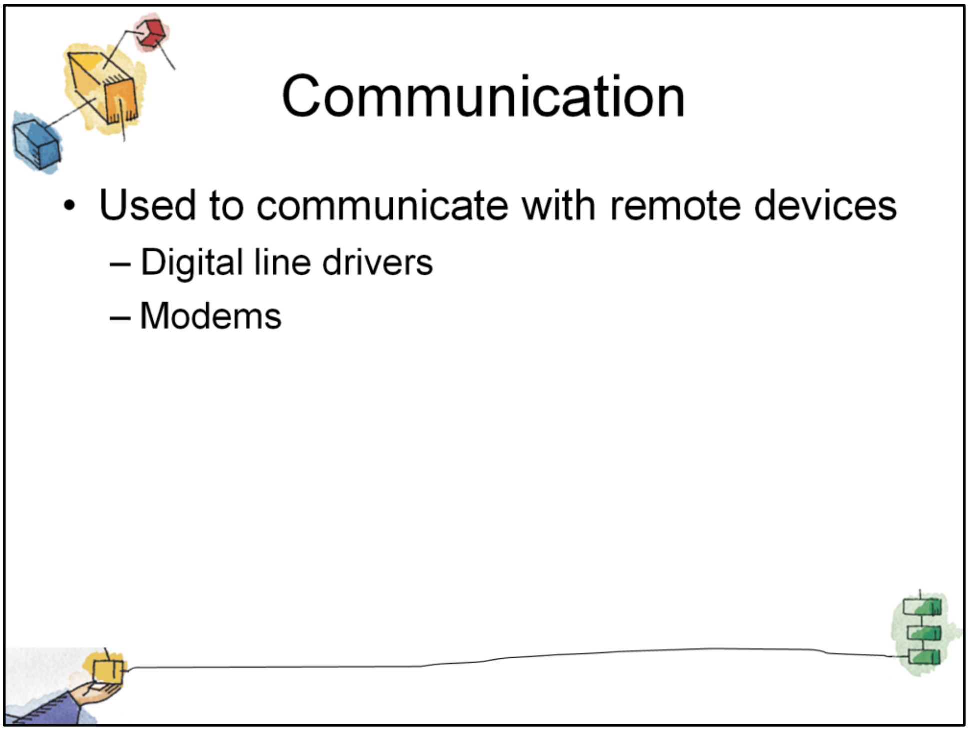


Machine readable

- Used to communicate with electronic equipment
  - Disk drives
  - USB keys
  - Sensors
  - Controllers
  - Actuators

Suitable for communicating with electronic equipment.

Examples are disk drives, USB keys, sensors, controllers, and actuators.

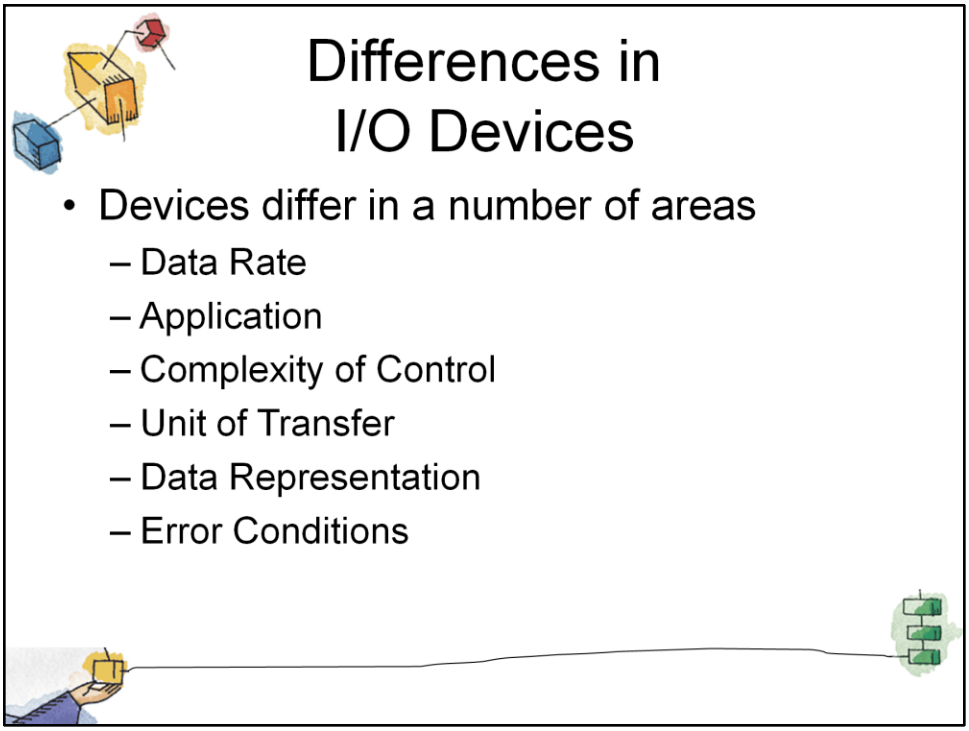


Communication

- Used to communicate with remote devices
  - Digital line drivers
  - Modems

Suitable for communicating with remote devices.

- Examples are digital line drivers and modems.

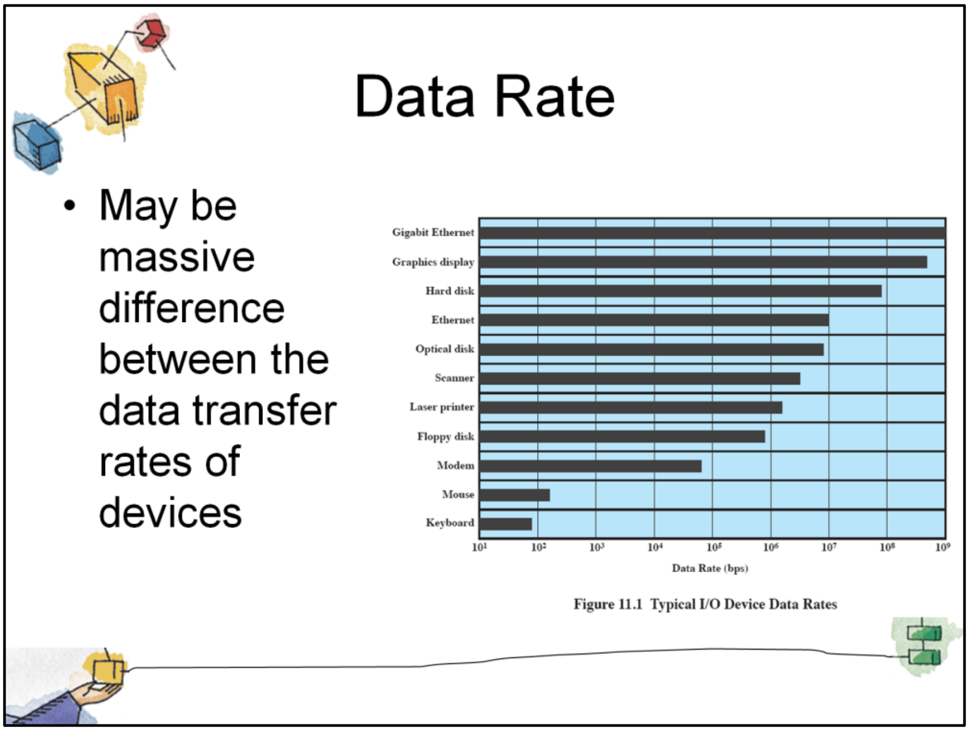


Differences in I/O Devices

- Devices differ in a number of areas
  - Data Rate
  - Application
  - Complexity of Control
  - Unit of Transfer
  - Data Representation
  - Error Conditions

Each of these are covered in subsequent slides

This diversity makes a uniform and consistent approach to I/O, both from the point of view of the operating system and from the point of view of user processes, difficult to achieve.

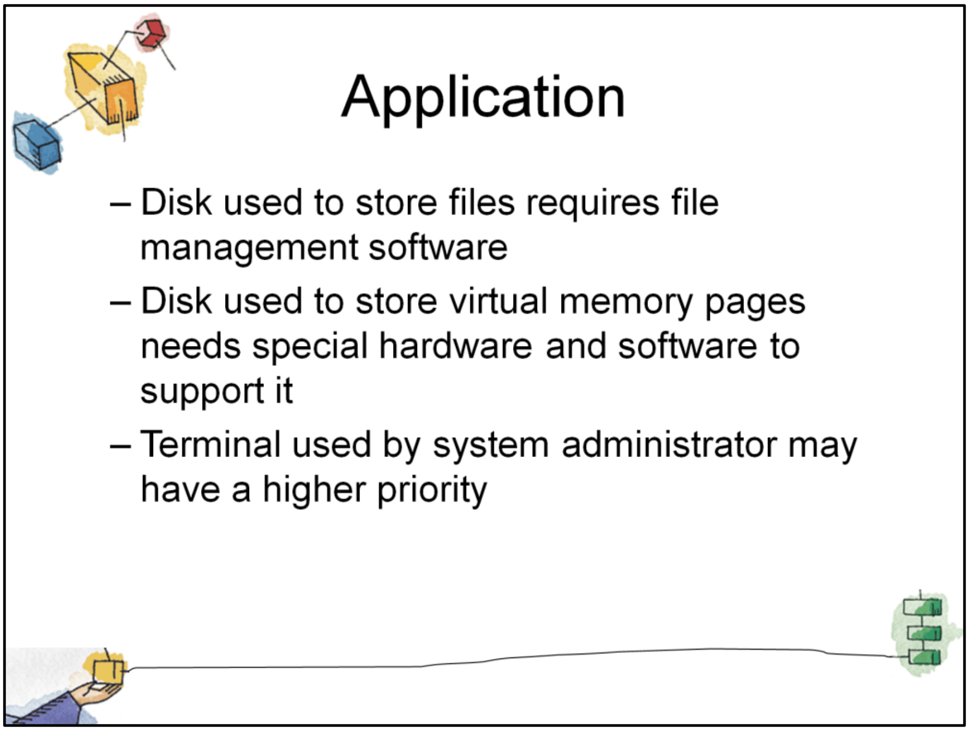


Data Rate

- May be massive difference between the data transfer rates of devices

Figure 11.1 Typical I/O Device Data Rates

There may be differences of several orders of magnitude between the data transfer rates.



Application

- 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

The use to which a device is put has an influence on the software and policies in the operating system and supporting utilities.

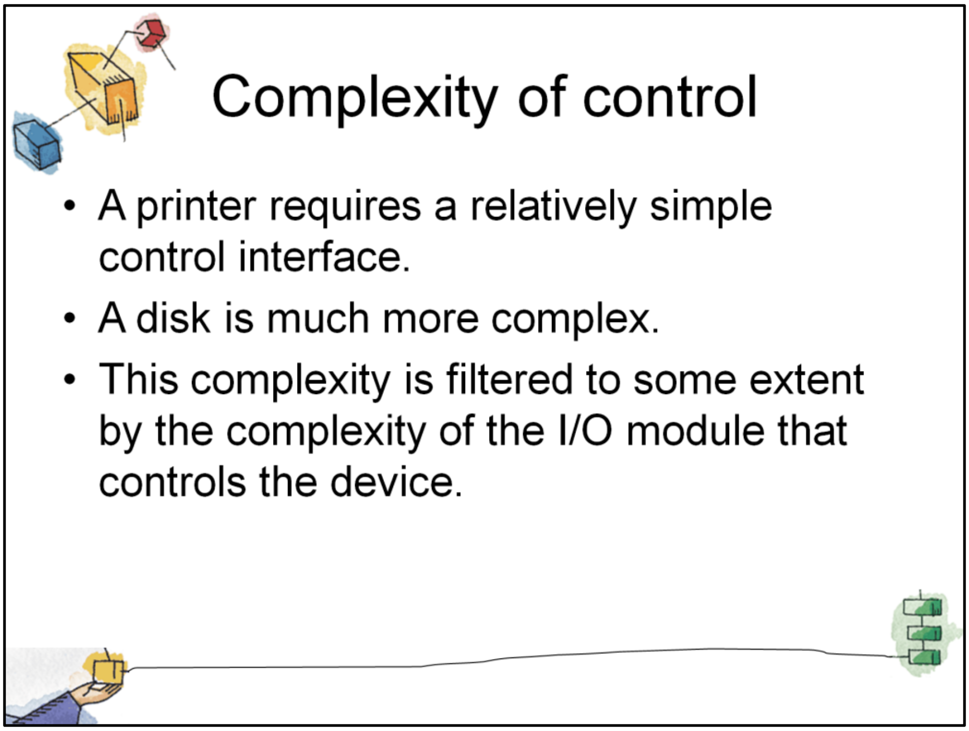
Examples:

- disk used for files requires the support of file management software.
- disk used as a backing store for pages in a virtual memory scheme depends on the use of virtual memory hardware and software.

These applications have an impact on disk scheduling algorithms.

Another example, a terminal may be used by an ordinary user or a system administrator.

- implying different privilege levels and perhaps different priorities in the operating system.

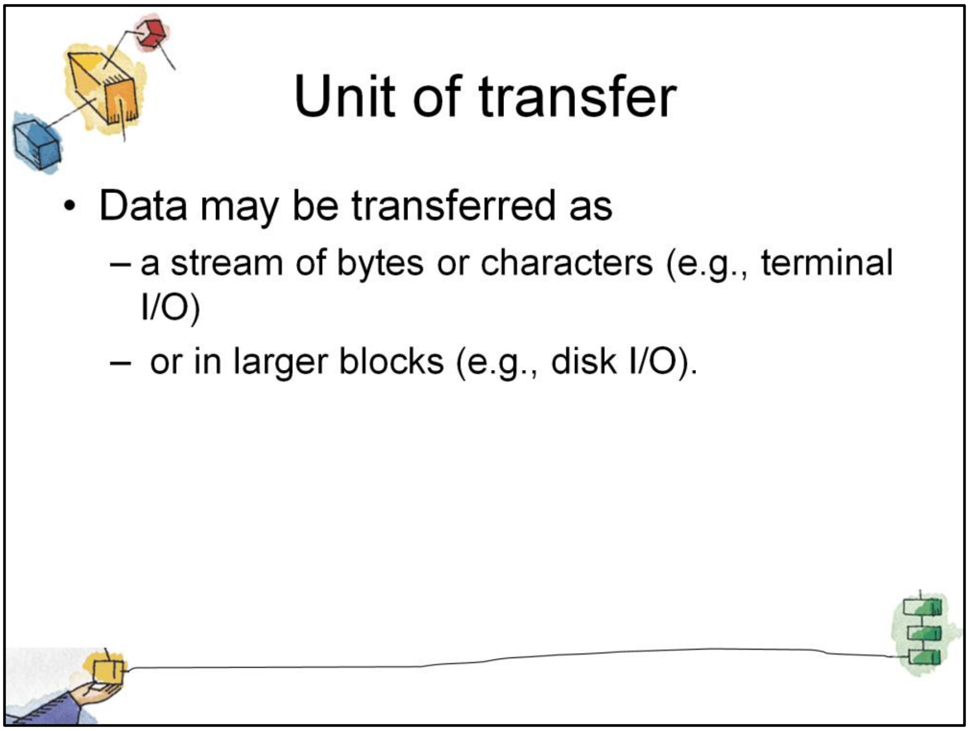


Complexity of control

- A printer requires a relatively simple control interface.
- A disk is much more complex.
- This complexity is filtered to some extent by the complexity of the I/O module that controls the device.

A printer requires a relatively simple control interface while a disk is much more complex.

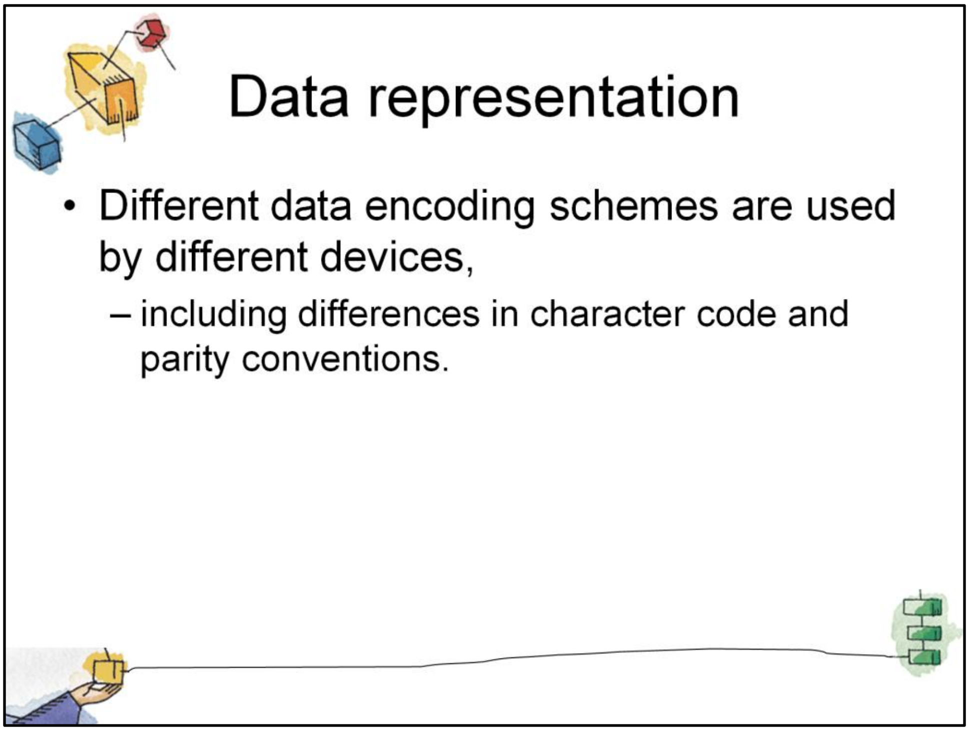
This complexity is filtered to some extent by the complexity of the I/O module that controls the device.



Unit of transfer

- Data may be transferred as
  - a stream of bytes or characters (e.g., terminal I/O)
  - or in larger blocks (e.g., disk I/O).

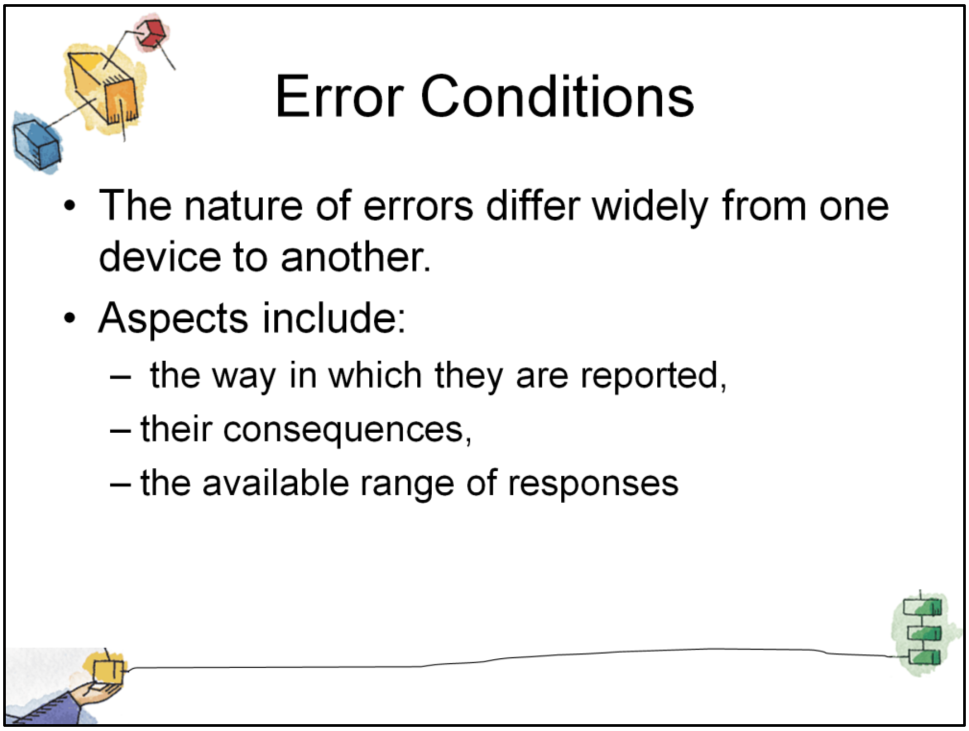
Data may be transferred as a stream of bytes or characters (e.g., terminal I/O) or in larger blocks (e.g., disk I/O).



Data representation

- Different data encoding schemes are used by different devices,
  - including differences in character code and parity conventions.

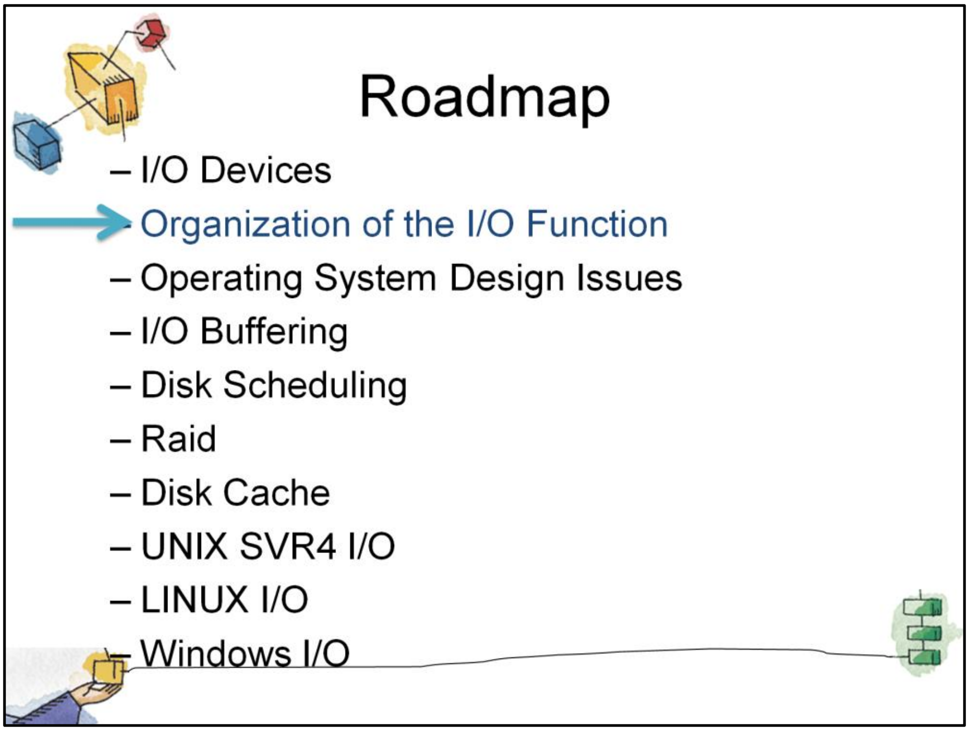
Different data encoding schemes are used by different devices, including differences in character code and parity conventions.



Error Conditions

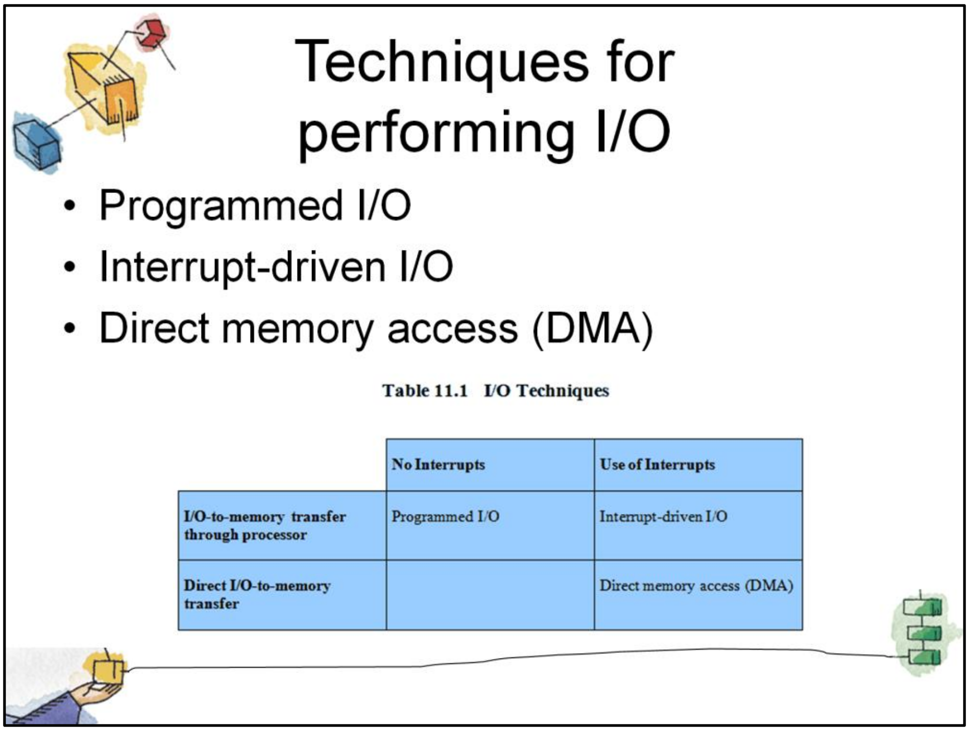
- The nature of errors differ widely from one device to another.
- Aspects include:
  - the way in which they are reported,
  - their consequences,
  - the available range of responses

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



Roadmap

- I/O Devices
- Organization of the I/O Function
- Operating System Design Issues
- I/O Buffering
- Disk Scheduling
- Raid
- Disk Cache
- UNIX SVR4 I/O
- LINUX I/O
- Windows I/O



Techniques for performing I/O

- Programmed I/O
- Interrupt-driven I/O
- Direct memory access (DMA)

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)

From section 1.7

Programmed I/O:

- 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.

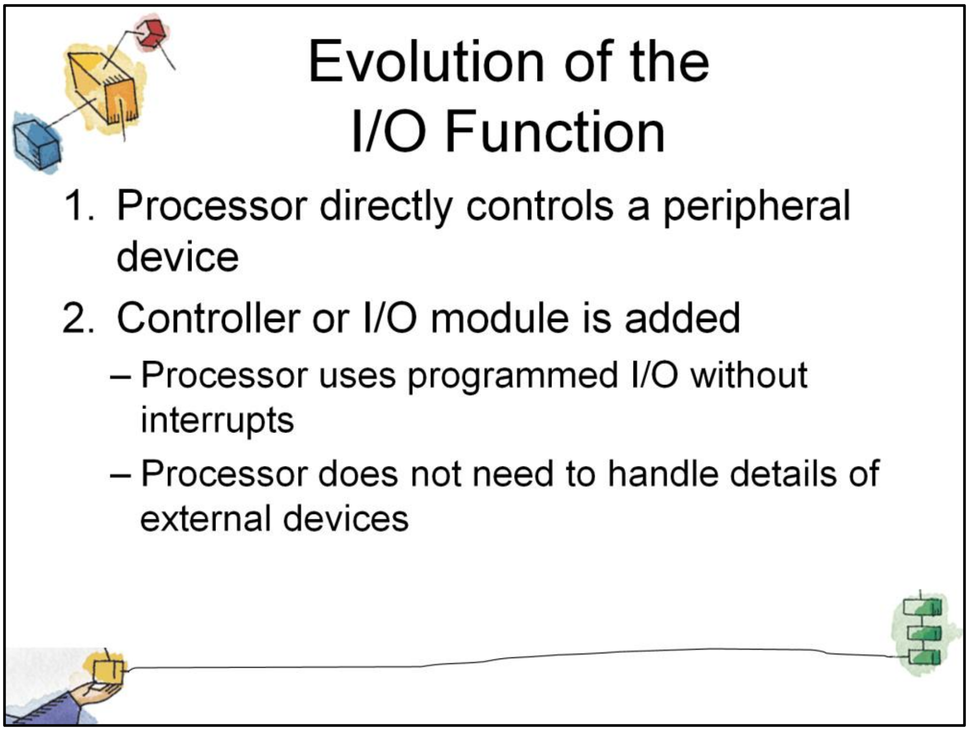
Interrupt-driven I/O:

- Processor issues an I/O command on behalf of a process.
- ***If*** the I/O instruction from the process is ***nonblocking***, then the processor continues to execute instructions from the process that issued the I/O command.
- ***If*** the I/O instruction is ***blocking***, then the next instruction that 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.
- The processor sends a request for the transfer of a block of data to the DMA module and is interrupted only after the entire block has been transferred.

Table 11.1 indicates the relationship among these three techniques.

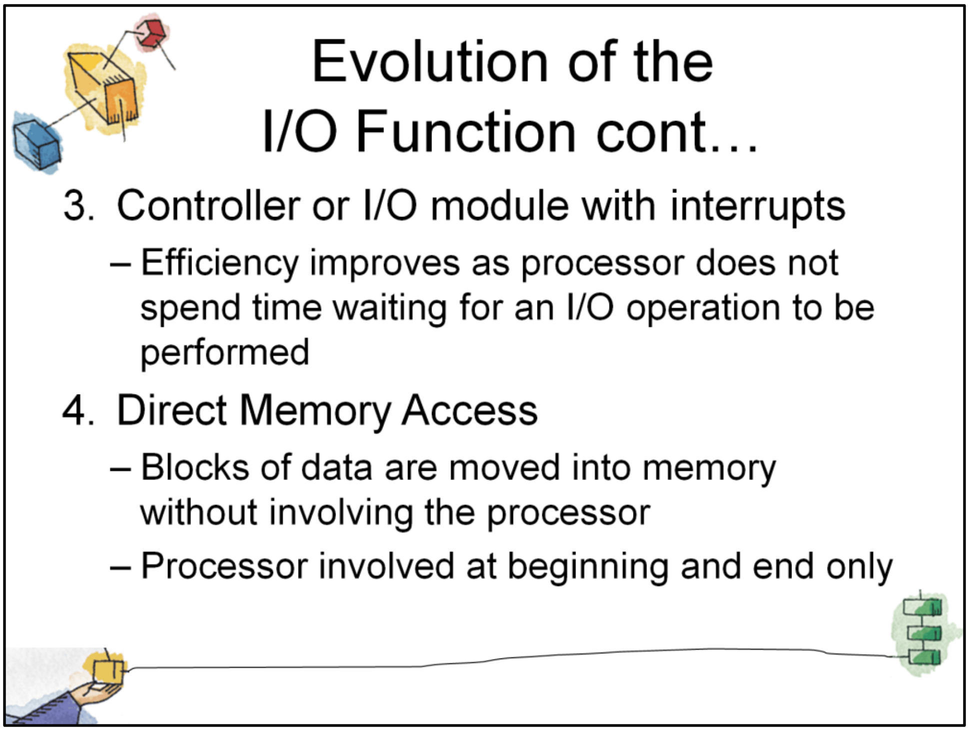


Evolution of the I/O Function

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

1. The processor directly controls a peripheral device.
   - This is seen in simple microprocessor-controlled devices.

2. A controller or I/O module is added.
   - The processor uses programmed I/O without interrupts.
   - With this step, the processor becomes somewhat divorced from the specific details of external device interfaces.



Evolution of the I/O Function cont…

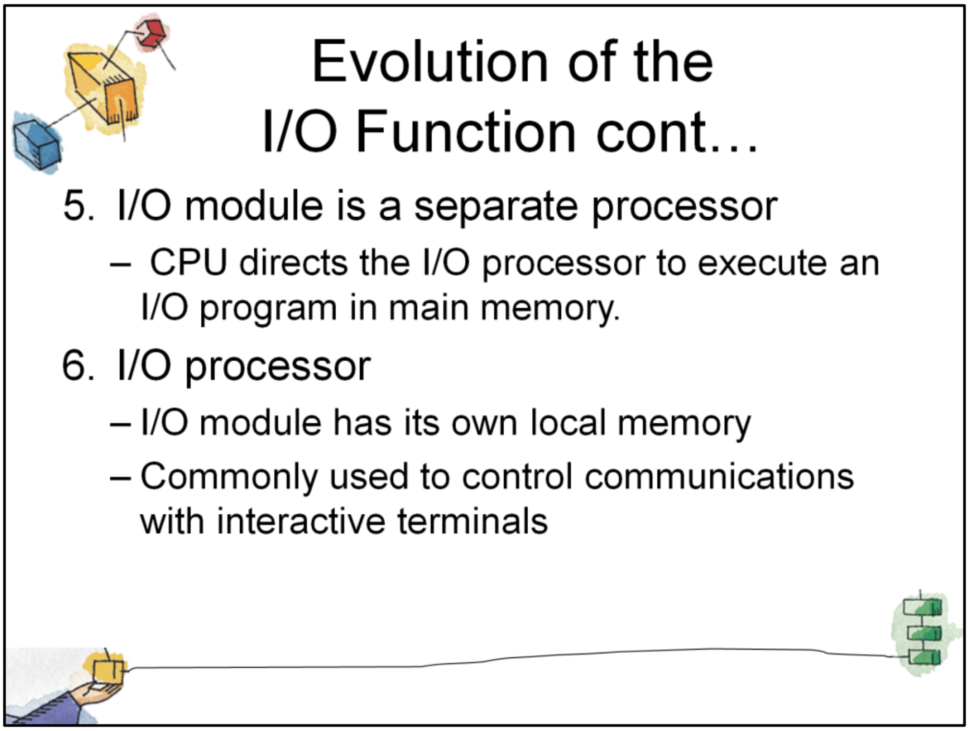
3. Controller or I/O module with interrupts
   - Efficiency improves as processor does not spend time waiting for an I/O operation to be performed
4. Direct Memory Access
   - Blocks of data are moved into memory without involving the processor
   - Processor involved at beginning and end only

3. Now interrupts are employed.

- The processor need not spend time waiting for an I/O operation to be performed, thus increasing efficiency.

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

- It can now move a block of data to or from memory without involving the processor, except at the beginning and end of the transfer.



Evolution of the I/O Function cont…

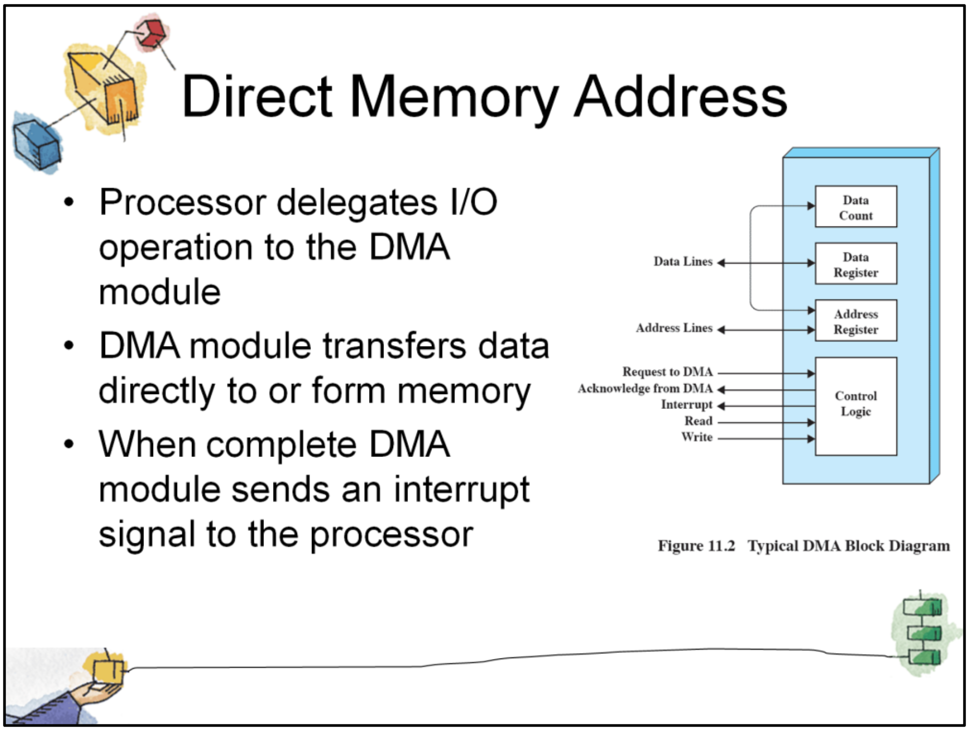
5. I/O module is a separate processor
   - CPU directs the I/O processor to execute an I/O program in main memory.
6. I/O processor
   - I/O module has its own local memory
   - Commonly used to control communications with interactive terminals

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

- CPU directs the I/O processor to execute an I/O program in main memory.
- The I/O processor fetches and executes these instructions without processor intervention.
- Allowing the processor to specify a sequence of I/O activities and to be interrupted only when the entire sequence has been performed.

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

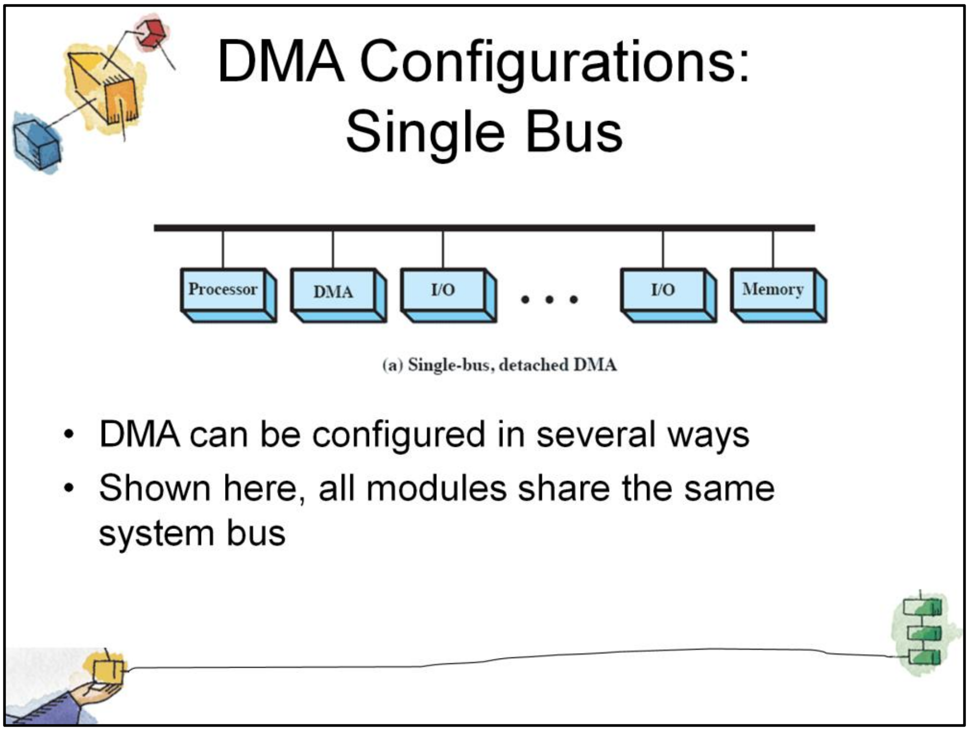
- A large set of I/O devices can be controlled, with minimal processor involvement.
- Commonly used to control communications with interactive terminals. The I/O processor takes care of most of the tasks involved in controlling the terminals.



Direct Memory Address

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

Figure 11.2 Typical DMA Block Diagram



DMA Configurations: Single Bus

Processor
DMA
I/O
• • •
I/O
Memory

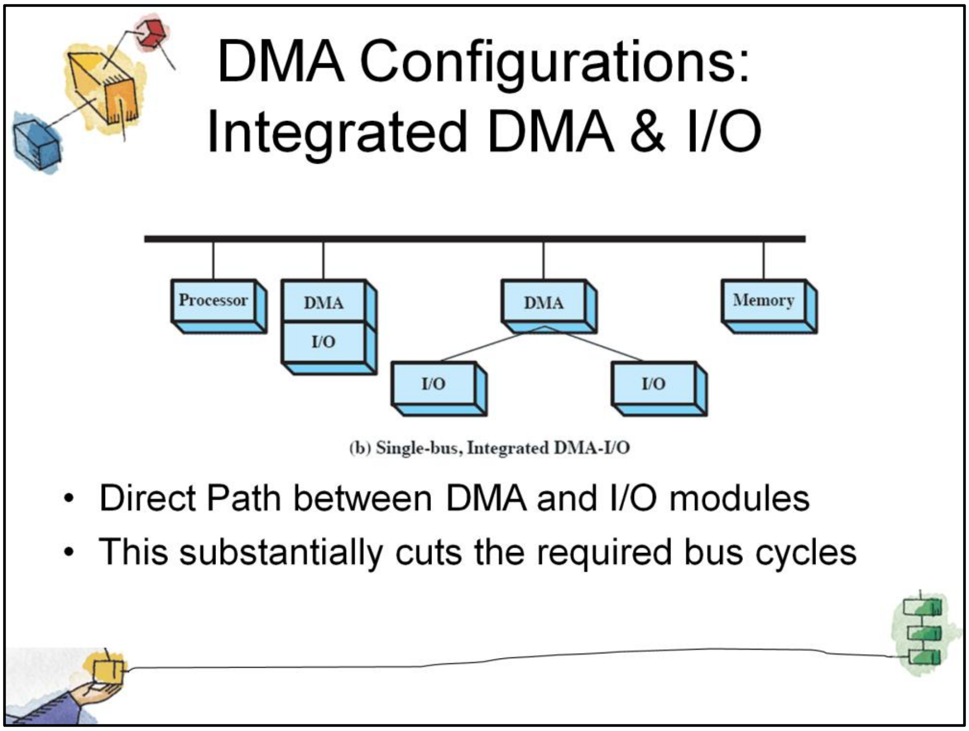
(a) Single-bus, detached DMA

- DMA can be configured in several ways
- Shown here, all modules share the same system bus

The DMA mechanism can be configured in a variety of ways.

Some possibilities are shown here In the first example, all modules share the same system bus.

- The DMA module, acting as a surrogate processor, uses programmed I/O to exchange data between memory and an I/O module through the DMA module.
- This is clearly inefficient: As with processor-controlled programmed I/O, each transfer of a word consumes two bus cycles (transfer request followed by transfer).



DMA Configurations: Integrated DMA & I/O

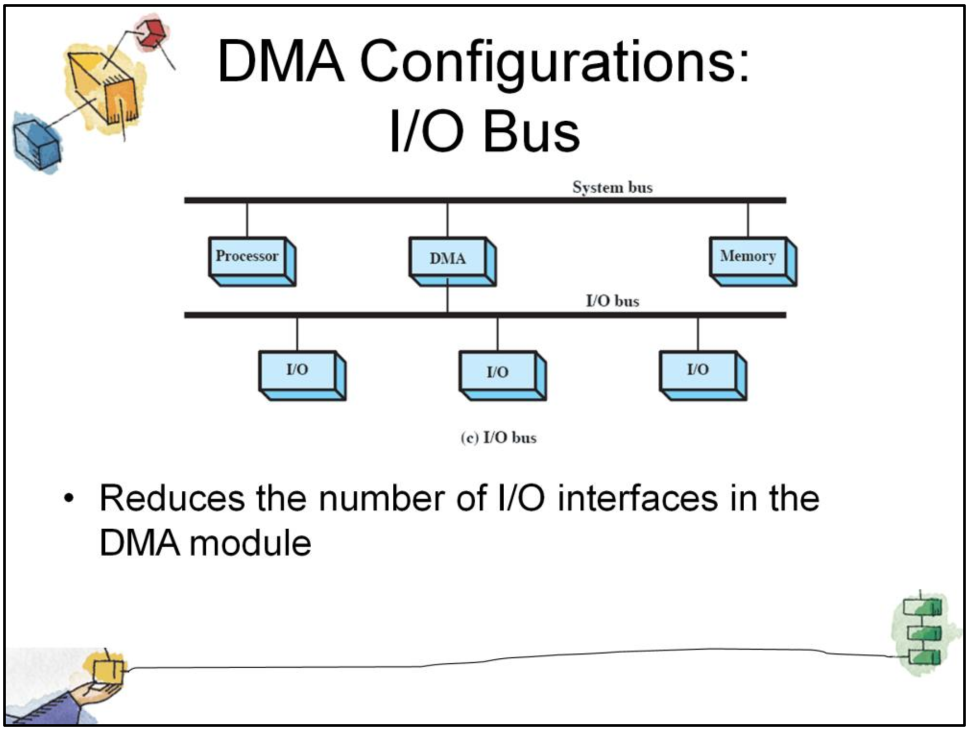
(b) Single-bus, Integrated DMA-I/O

- Direct Path between DMA and I/O modules
- This substantially cuts the required bus cycles

The number of required bus cycles can be cut substantially by integrating the DMA and I/O functions.

This means that there is a path between the DMA module and one or more I/O modules that does not include the system bus.

The DMA logic may actually be a part of an I/O module, or it may be a separate module that controls one or more I/O modules.



DMA Configurations: I/O Bus

(c) I/O bus

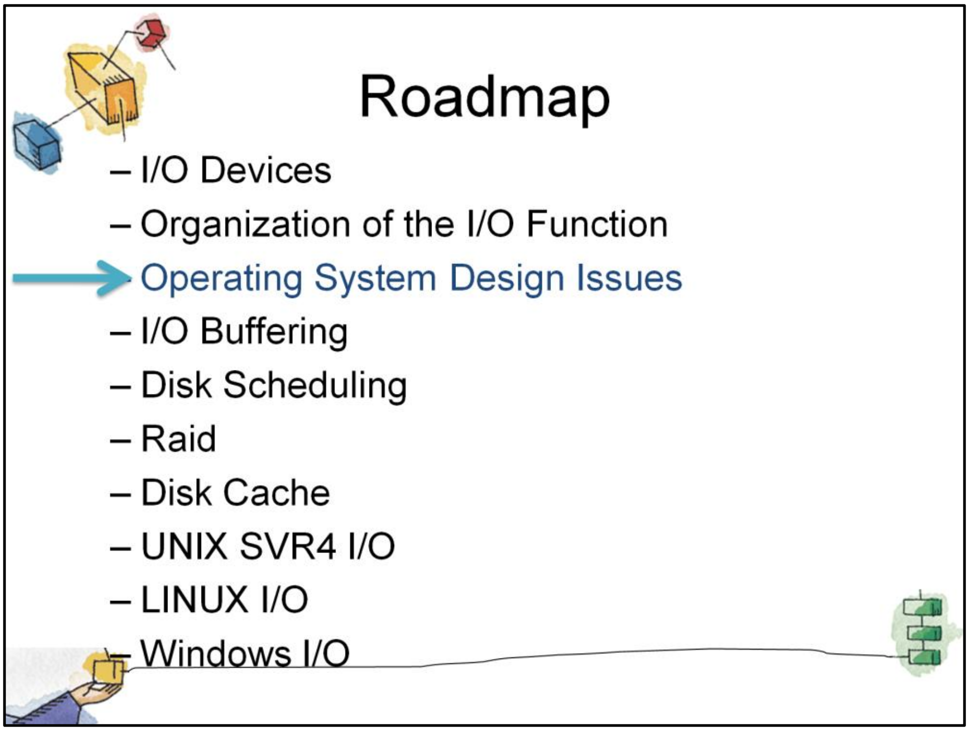
- Reduces the number of I/O interfaces in the DMA module

This concept can be taken one step further by connecting I/O modules to the DMA module using an I/O bus

This reduces the number of I/O interfaces in the DMA module to one and provides for an easily expandable configuration.

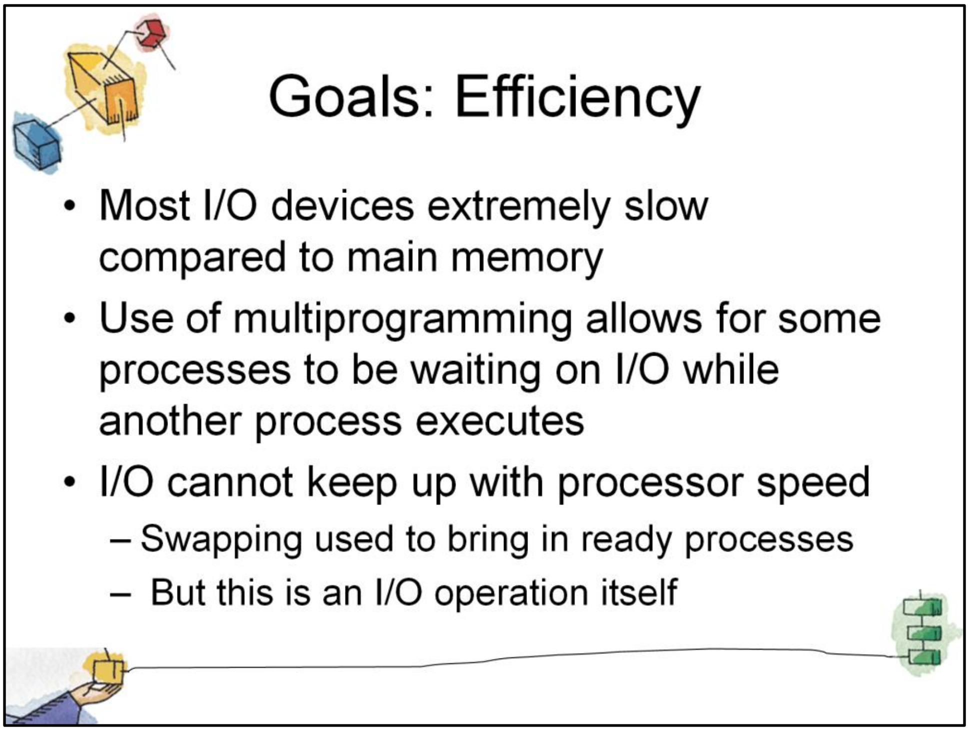
In all of these cases the system bus that the DMA module shares with the processor and main memory is used by the DMA module only to exchange data with memory and to exchange control signals with the processor.

The exchange of data between the DMA and I/O modules takes place off the system bus.



Roadmap

- I/O Devices
- Organization of the I/O Function
- → Operating System Design Issues
- I/O Buffering
- Disk Scheduling
- Raid
- Disk Cache
- UNIX SVR4 I/O
- LINUX I/O
- Windows I/O



Goals: Efficiency

- Most I/O devices extremely slow compared to main memory
- Use of multiprogramming allows for some processes to be waiting on I/O while another process executes
- I/O cannot keep up with processor speed
  - Swapping used to bring in ready processes
  - But this is an I/O operation itself

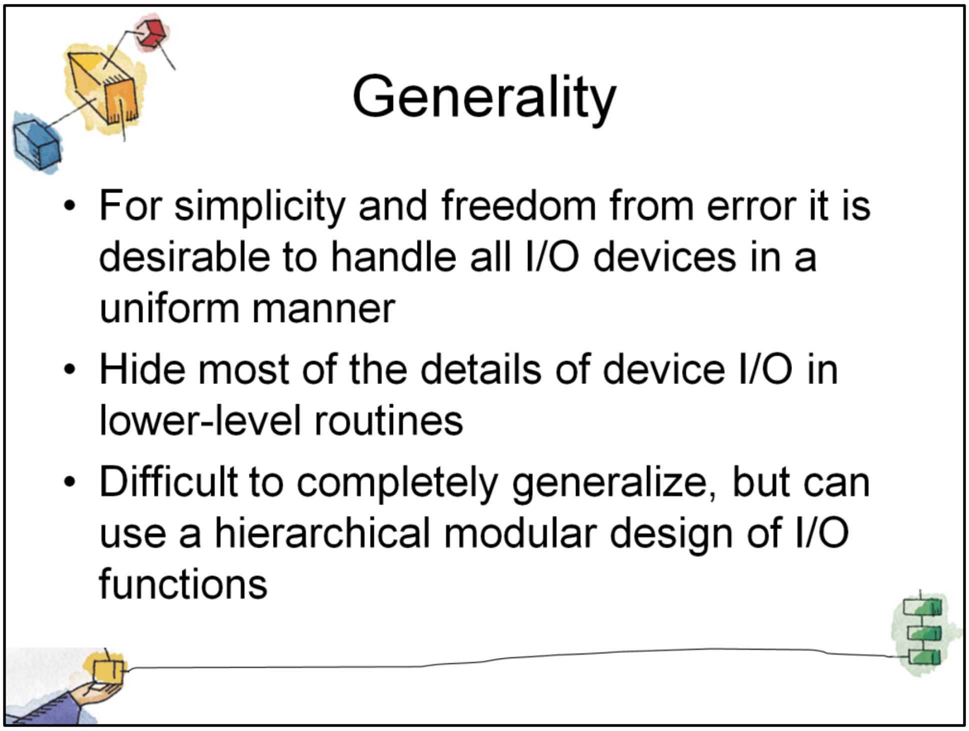
Efficiency is important because I/O operations often form a bottleneck in a computing system.

One way to tackle this problem is multiprogramming, which, as we have seen, allows some processes to be waiting on I/O operations while another process is executing.

- However, even with the vast size of main memory in today’s machines, often I/O is not keeping up with the activities of the processor.

Swapping is used to bring in additional ready processes to keep the processor busy, but this in itself is an I/O operation.

- Thus, a major effort in I/O design has been schemes for improving the efficiency of the I/O.
- The area that has received the most attention, because of its importance, is disk I/O.



Generality

- For simplicity and freedom from error it is desirable to handle all I/O devices in a uniform manner
- Hide most of the details of device I/O in lower-level routines
- Difficult to completely generalize, but can use a hierarchical modular design of I/O functions

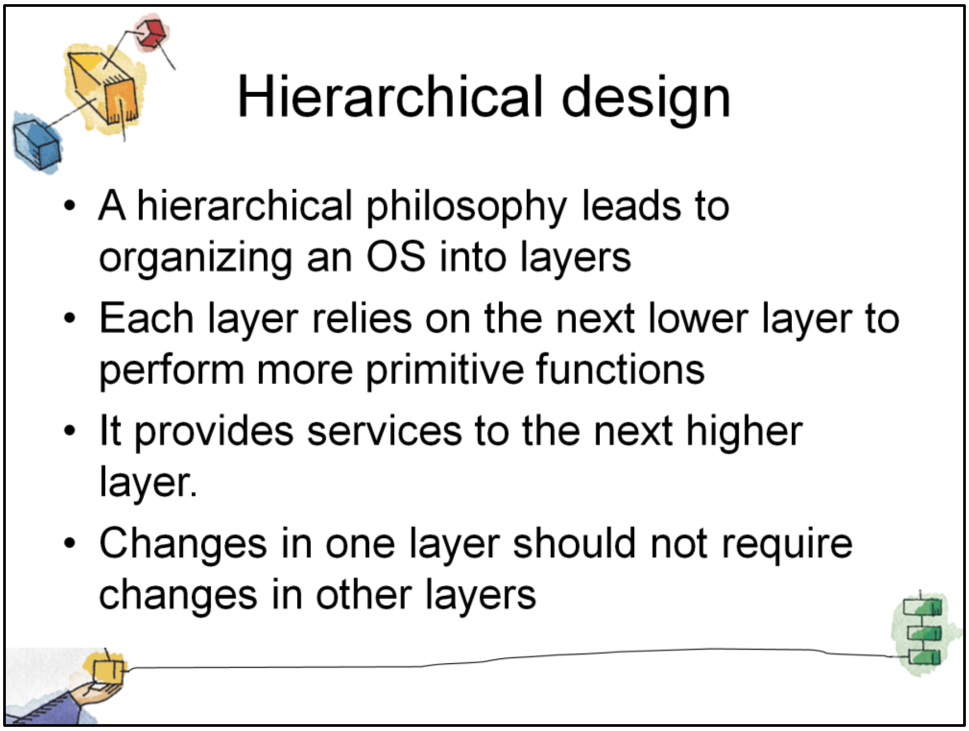
For simplicity and freedom from error, it is desirable to handle all devices in a uniform manner.

> This applies both to the way in which processes **view** I/O devices and the way in which the operating system **manages** I/O devices and operations.

Because of the diversity of device characteristics, it is difficult in practice to achieve true generality.

What can be done is to use a hierarchical, modular approach to the design of the I/O function.

> •T his hides most of the details of device I/O in lower-level routines so that user processes and upper levels of the operating system see devices in terms of general functions, such as read, write, open, close, lock, unlock.



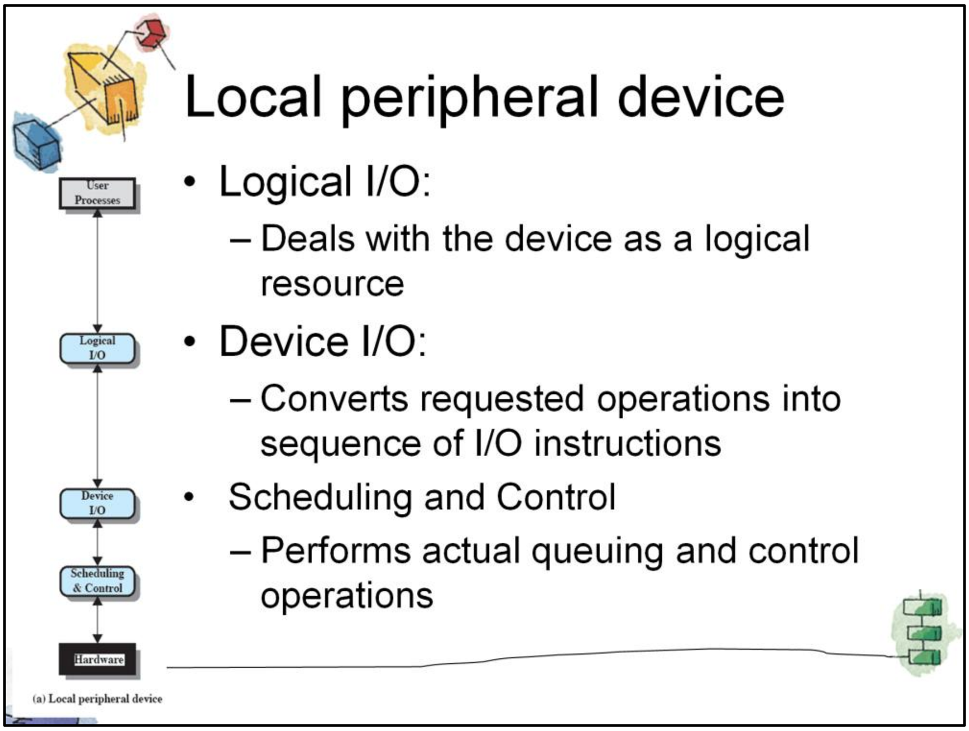
Hierarchical design

- 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

The hierarchical philosophy developed in Chapter 2 suggested that the functions of the operating system should be separated according to their complexity, their characteristic time scale, and their level of abstraction.

This approach leads to an organization of the operating system into a series of layers.

- Each layer performs a related subset of the functions required of the operating system.
- It relies on the next lower layer to perform more primitive functions and to conceal the details of those functions.
- It provides services to the next higher layer.
- Ideally, the layers should be defined so that changes in one layer do not require changes in other layers.



Local peripheral device

- Logical I/O:
  - Deals with the device as a logical resource
- Device I/O:
  - Converts requested operations into sequence of I/O instructions
- Scheduling and Control
  - Performs actual queuing and control operations

Logical I/O:

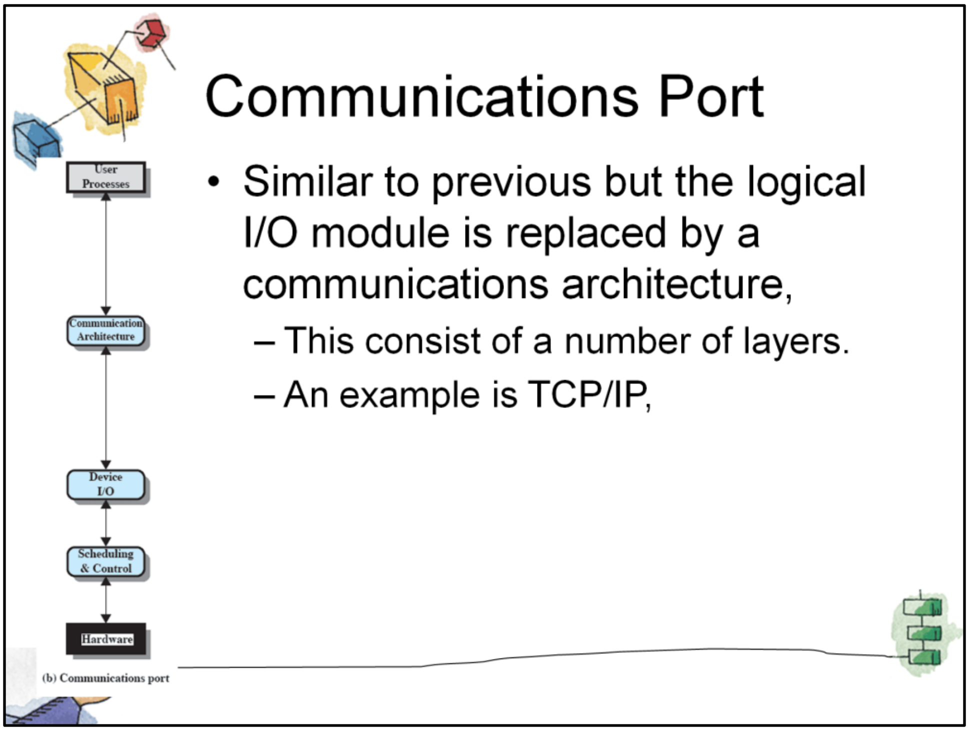
- Deals with the device as a logical resource and is not concerned with the details of actually controlling the device.
- Concerned with managing general I/O functions on behalf of user processes, allowing them to deal with the device in terms of a device identifier and simple commands such as open, close, read, write.

Device I/O:

- The requested operations and data (buffered characters, records, etc.) are converted into appropriate sequences of I/O instructions, channel commands, and controller orders.
- Buffering techniques may be used to improve utilization.

Scheduling and control:

- The actual queuing and scheduling of I/O operations occurs at this layer, as well as the control of the operations.
- Interrupts are handled at this layer and I/O status is collected and reported.
- This is the layer of software that actually interacts with the I/O module and hence the device hardware.



Communications Port

- Similar to previous but the logical I/O module is replaced by a communications architecture,
  - This consist of a number of layers.
  - An example is TCP/IP,

User Processes

Communication Architecture

Device I/O

Scheduling & Control

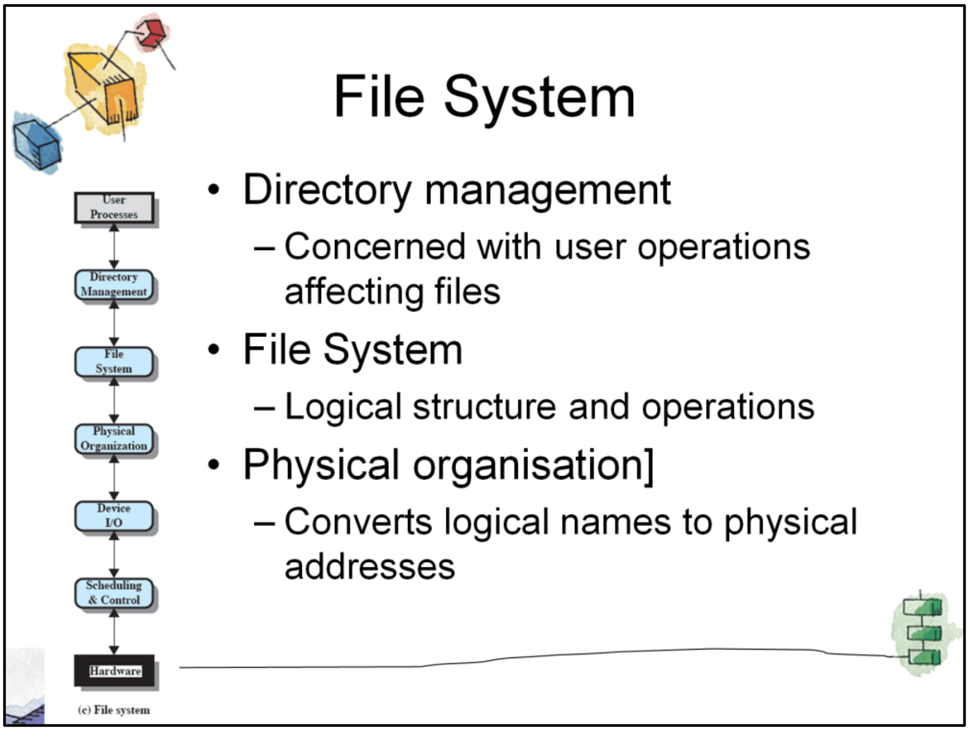
Hardware

(b) Communications port

The logical I/O module is replaced by a communications architecture,

- which may itself consist of a number of layers.

An example is TCP/IP,.



(c) File system

File System

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

Directory management:

•At this layer, symbolic file names are converted to identifiers that either reference the file directly or indirectly through a file descriptor or index table.

• Concerned with user operations that affect the directory of files, such as add, delete, and reorganize.

File system:

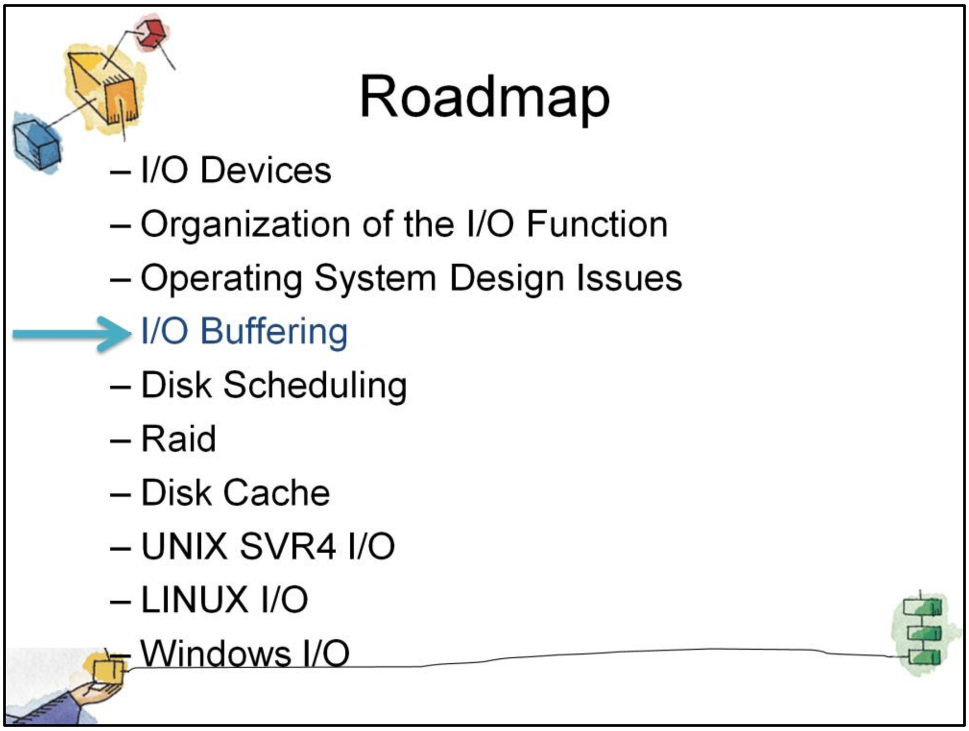
• This layer deals with the logical structure of files and with the operations that can be specified by users, such as open, close, read, write.

• Access rights are also managed at this layer.

Physical organization:

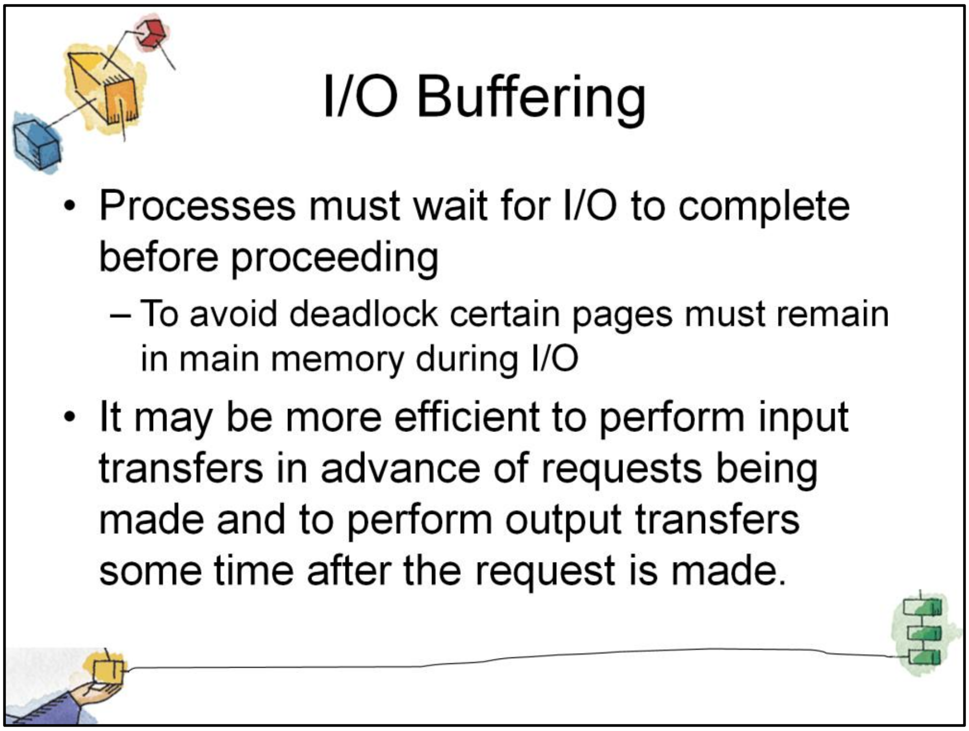
• Files and records must be converted to physical secondary storage addresses, taking into account the physical track and sector structure of the secondary storage device.

• Allocation of secondary storage space and main storage buffers is generally treated at this layer as well.



Roadmap

- I/O Devices
- Organization of the I/O Function
- Operating System Design Issues
- → I/O Buffering
- Disk Scheduling
- Raid
- Disk Cache
- UNIX SVR4 I/O
- LINUX I/O
- Windows I/O



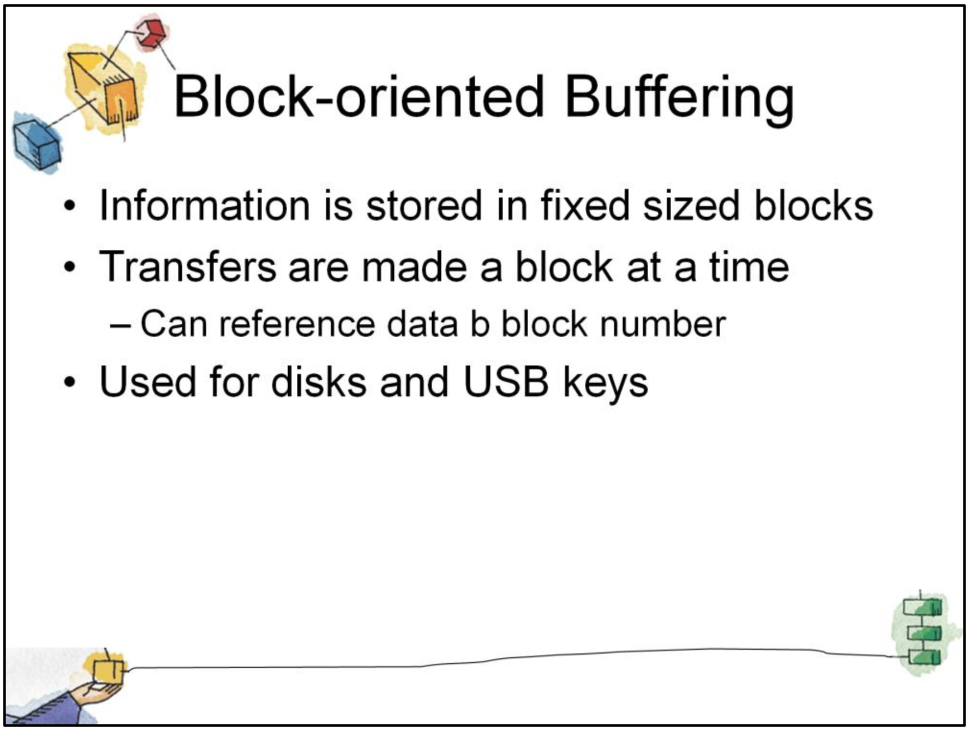
I/O Buffering

- Processes must wait for I/O to complete before proceeding
  - To avoid deadlock certain pages must remain in main memory during I/O
- It may be more efficient to perform input transfers in advance of requests being made and to perform output transfers some time after the request is made.

To avoid deadlock, the user memory involved in the I/O operation must be locked in main memory immediately before the I/O request is issued, even though the I/O operation is queued and may not be executed for some time.

- If a block is being transferred from a user process area directly to an I/O module, then the process is blocked during the transfer and the process may not be swapped out.

To avoid these 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.



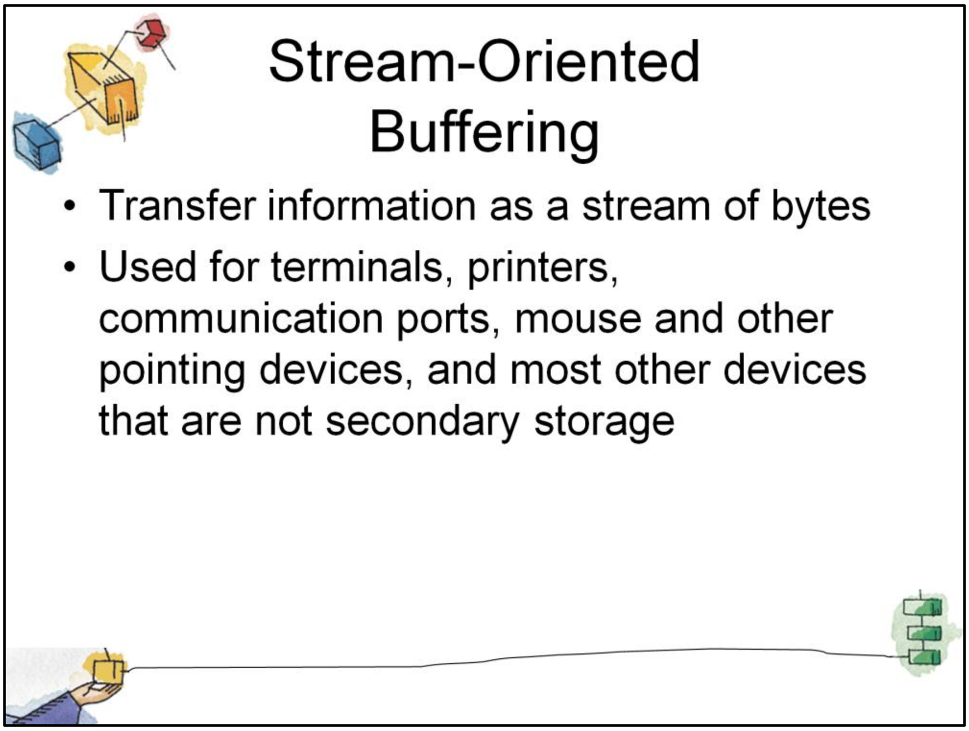
Block-oriented Buffering

- Information is stored in fixed sized blocks
- Transfers are made a block at a time
  - Can reference data b block number
- Used for disks and USB keys

A block-oriented device stores information in blocks that are usually of fixed size, and transfers are made one block at a time.

Generally, it is possible to reference data by its block number.

Disks and USB keys are examples of block-oriented devices.



Stream-Oriented Buffering

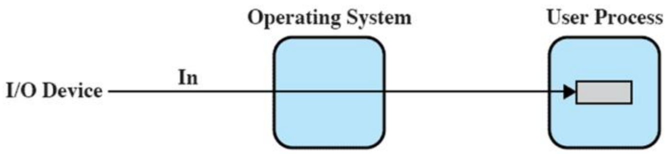
- Transfer information as a stream of bytes
- Used for terminals, printers, communication ports, mouse and other pointing devices, and most other devices that are not secondary storage

A stream-oriented device transfers data in and out as a stream of bytes, with no block structure.

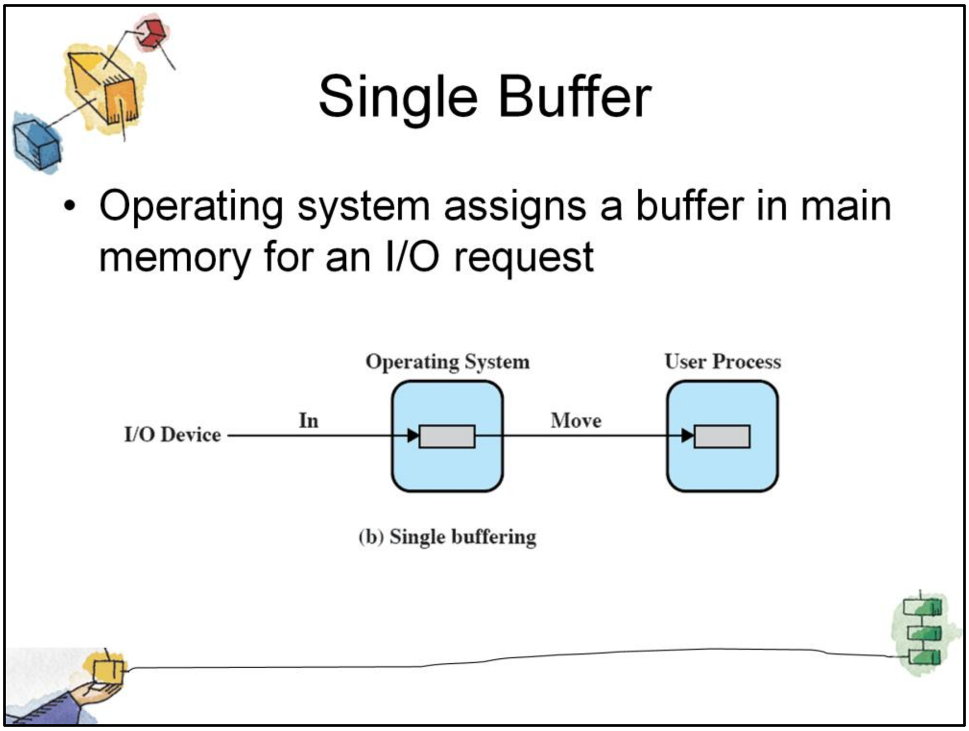
Terminals, printers, communications ports, mouse and other pointing devices, and most other devices that are not secondary storage are stream oriented.

No Buffer

- Without a buffer, the OS directly access the device as and when it needs



(a) No buffering

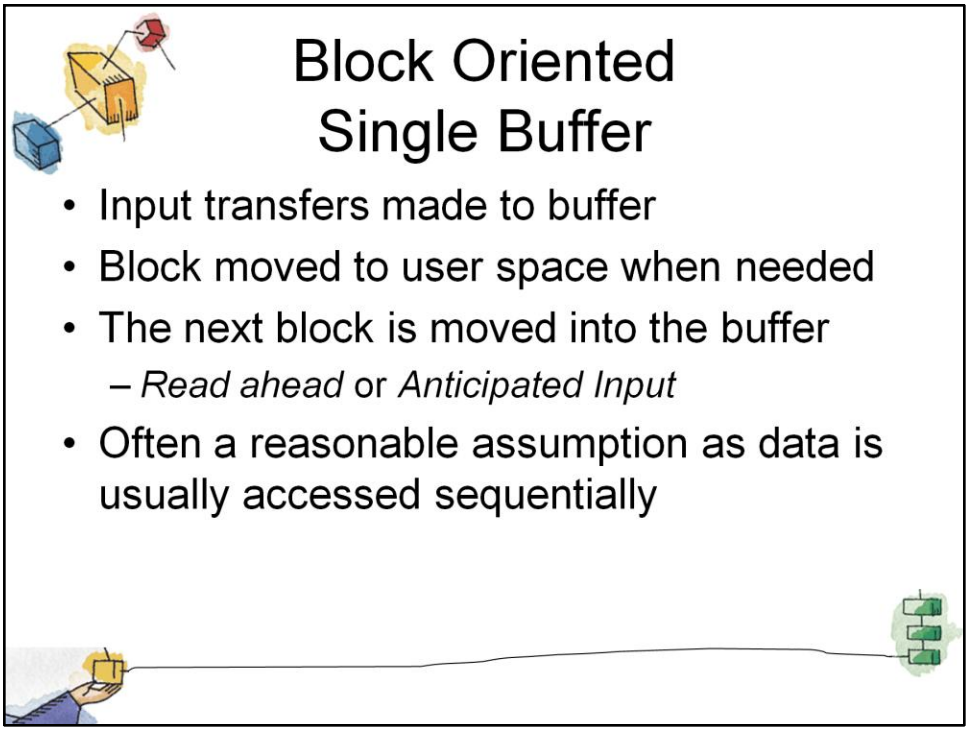


Single Buffer

- Operating system assigns a buffer in main memory for an I/O request

(b) Single buffering

When a user process issues an I/O request, the operating system assigns a buffer in the system portion of main memory to the operation.



Block Oriented Single Buffer

- Input transfers made to buffer
- Block moved to user space when needed
- The next block is moved into the buffer
  - *Read ahead* or *Anticipated Input*
- Often a reasonable assumption as data is usually accessed sequentially

For block-oriented devices,

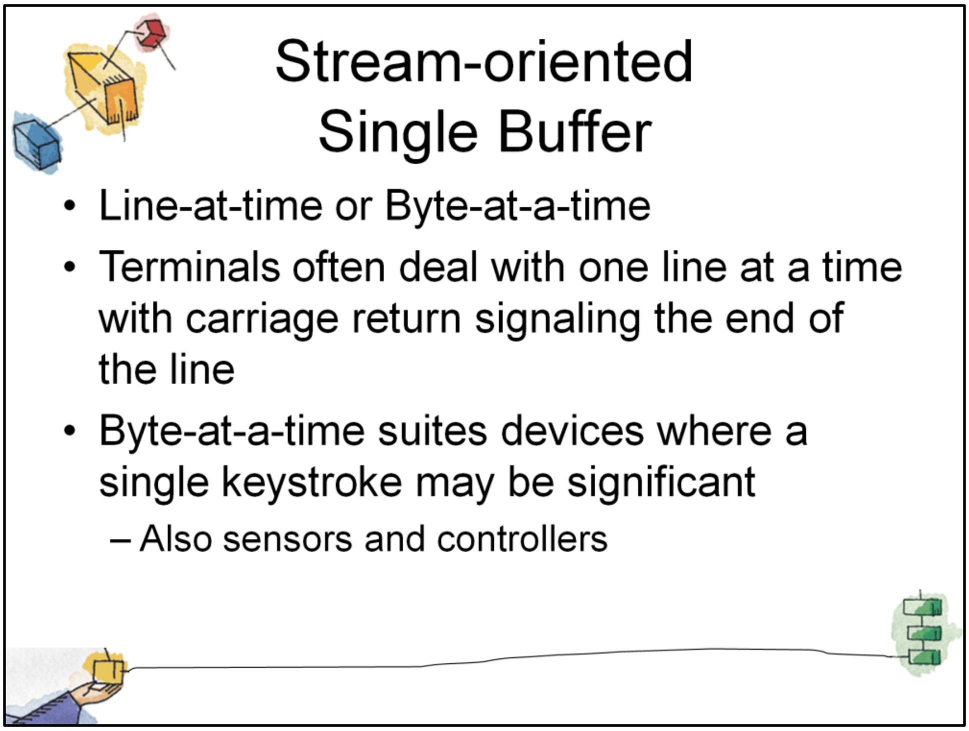
- Input transfers are made to the system buffer.
- When the transfer is complete, the process moves the block into user space and immediately requests another block.

Called **reading ahead**, or **anticipated input**;

- it is done in the expectation that the block will eventually be needed.

Often this is a reasonable assumption most of the time because data are usually accessed sequentially.

- Only at the end of a sequence of processing will a block be read in unnecessarily.

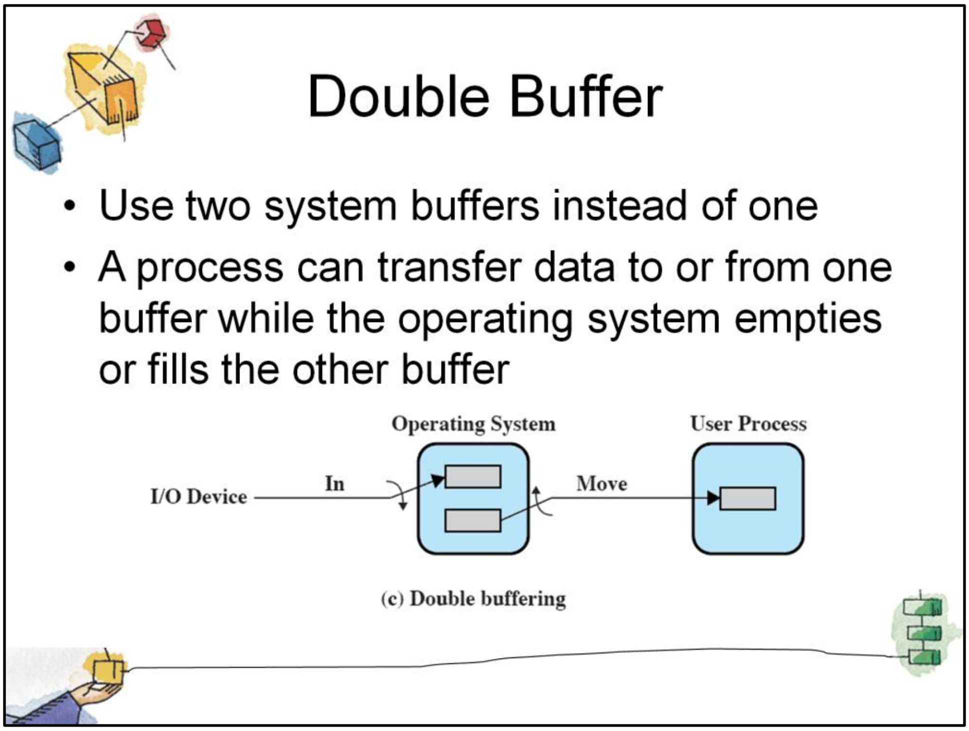


Stream-oriented Single Buffer

- Line-at-time or Byte-at-a-time
- Terminals often deal with one line at a time with carriage return signaling the end of the line
- Byte-at-a-time suites devices where a single keystroke may be significant
  - Also sensors and controllers

The single buffering scheme 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 (sometimes called dumb terminals).
- Byte-at-a-time operation is used on where each keystroke is significant, or for peripherals such as sensors and controllers.

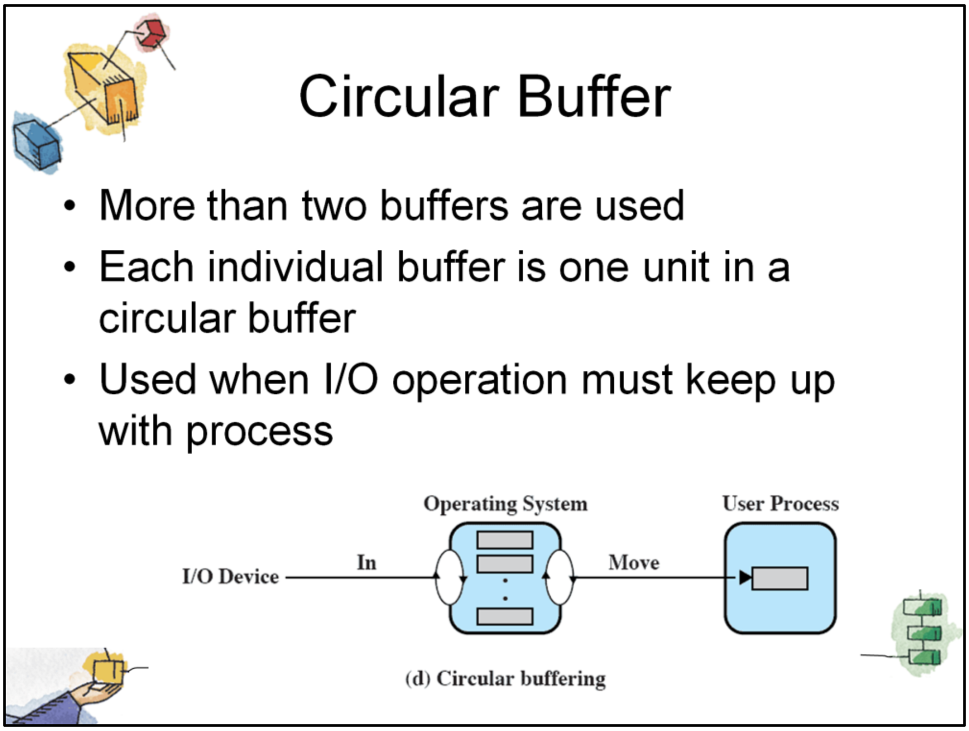


Double Buffer

- 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

A process transfers data to (or from) one buffer while the operating system empties (or fills) the other.



Circular Buffer

- More than two buffers are used
- Each individual buffer is one unit in a circular buffer
- Used when I/O operation must keep up with process

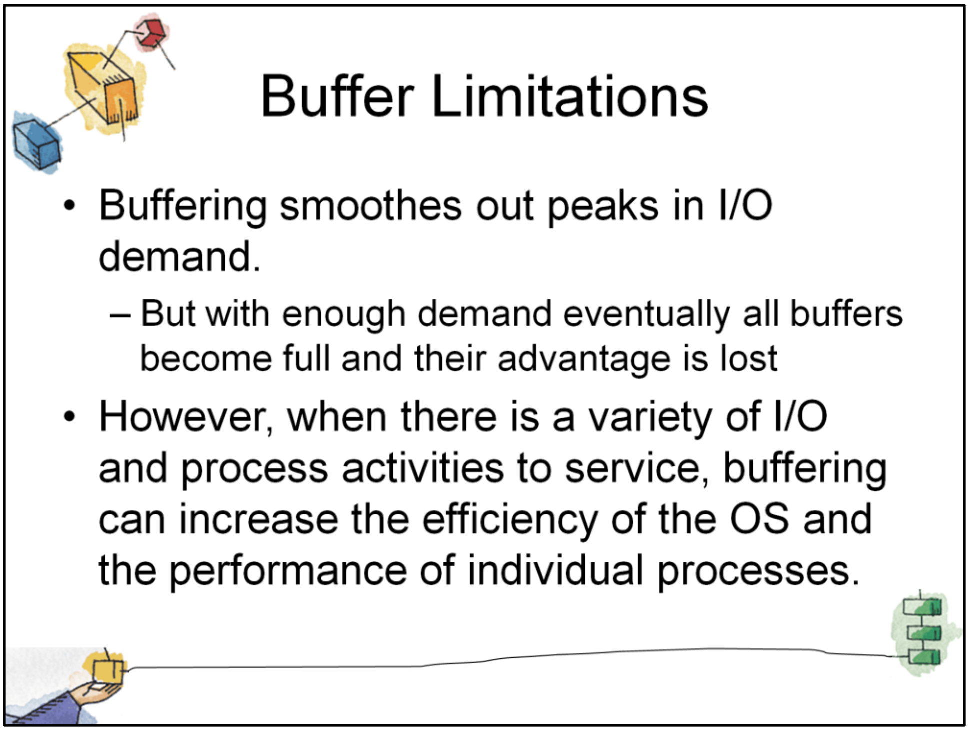
Operating System
User Process
I/O Device
In
Move

(d) Circular buffering

Double buffering may be inadequate if the process performs rapid bursts of I/O.

The problem can often be alleviated by using more than two buffers.

When more than two buffers are used, the collection of buffers is itself referred to as a circular buffer with each individual buffer being one unit in the circular buffer.



Buffer Limitations

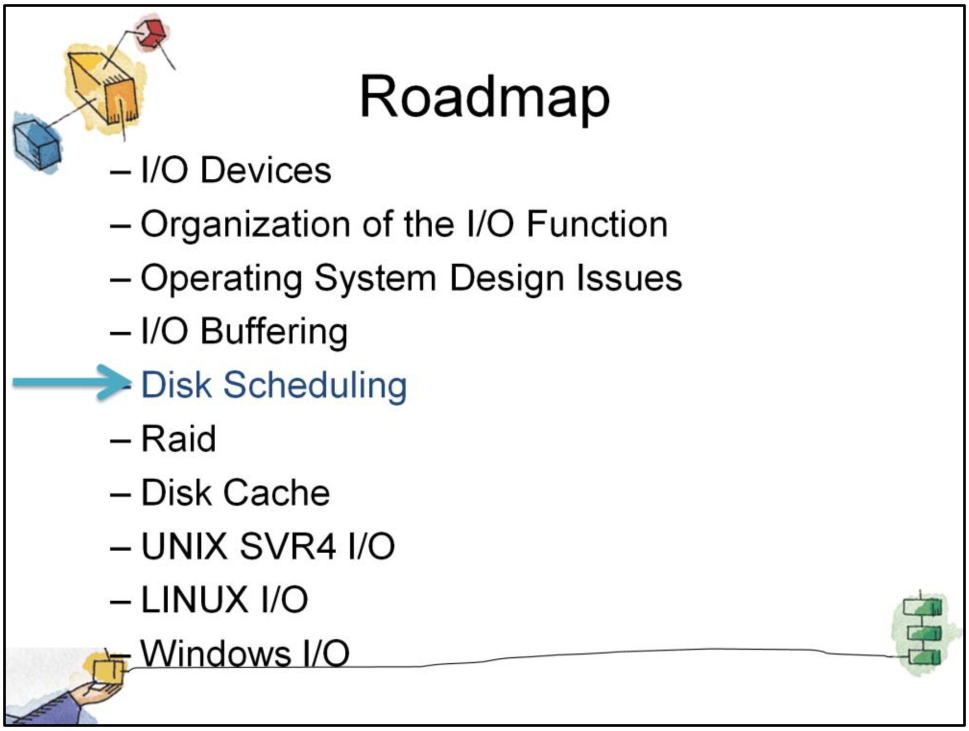
- Buffering smoothes out peaks in I/O demand.
  - But with enough demand eventually all buffers become full and their advantage is lost
- However, when there is a variety of I/O and process activities to service, buffering can increase the efficiency of the OS and the performance of individual processes.

Buffering is a technique that smoothes out peaks in I/O demand.

However, no amount of buffering will allow an I/O device to keep pace with a process indefinitely when the average demand of the process is greater than the I/O device can service.

- •Even with multiple buffers, all of the buffers will eventually fill up and the process will have to wait after processing each chunk of data.

However, in a multiprogramming environment, when there is a variety of I/O activity and a variety of process activity to service, buffering is one tool that can increase the efficiency of the operating system and the performance of individual processes.



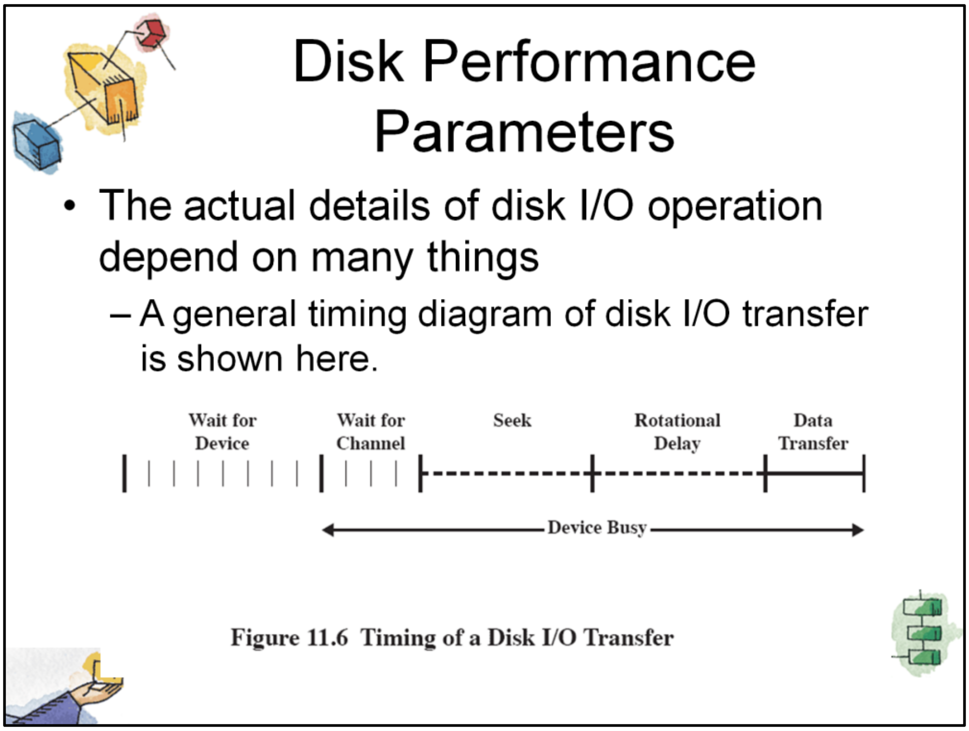
Roadmap

- – I/O Devices
- – Organization of the I/O Function
- – Operating System Design Issues
- – I/O Buffering
- Disk Scheduling
- – Raid
- – Disk Cache
- – UNIX SVR4 I/O
- – LINUX I/O
- – Windows I/O

Over the last 40 years, the increase in the speed of processors and main memory has far outstripped that for disk access, with processor and main memory speeds increasing by about two orders of magnitude compared to one order of magnitude for disk.

- The result is that disks are currently at least four orders of magnitude slower than main memory.
- Thus, the performance of disk storage subsystem is of vital concern.

In this section, we highlight some of the key issues and look at the most important approaches.



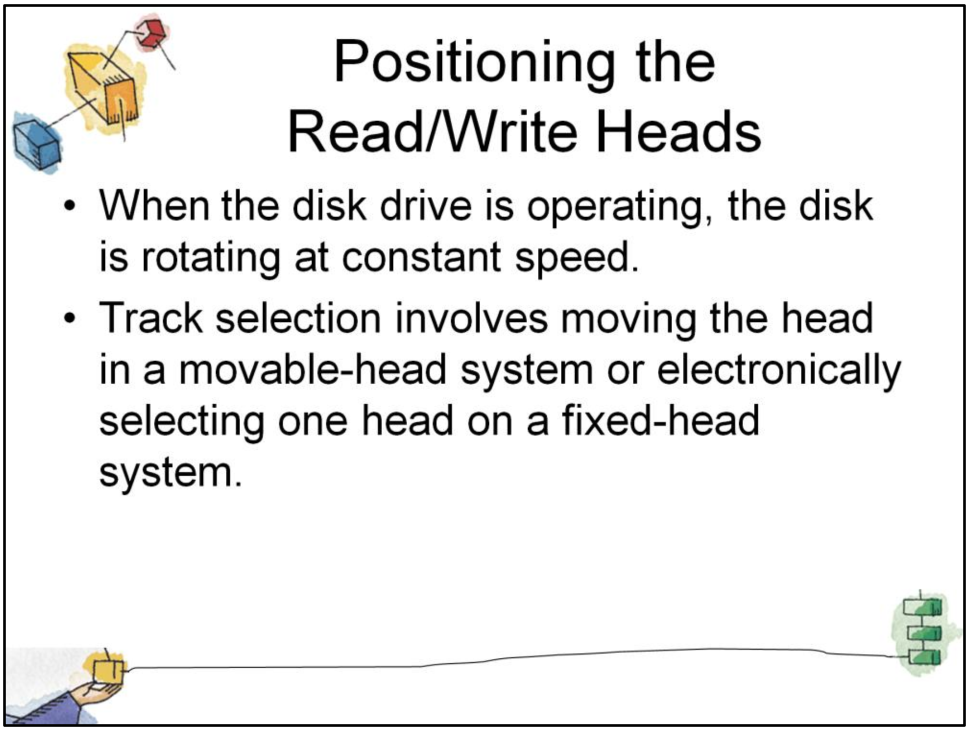
Disk Performance Parameters

- The actual details of disk I/O operation depend on many things
  - A general timing diagram of disk I/O transfer is shown here.

Figure 11.6 Timing of a Disk I/O Transfer

The actual details of disk I/O operation depend on the computer system, the operating system, and the nature of the I/O channel and disk controller hardware.

A general timing diagram of disk I/O transfer is shown in Figure 11.6.



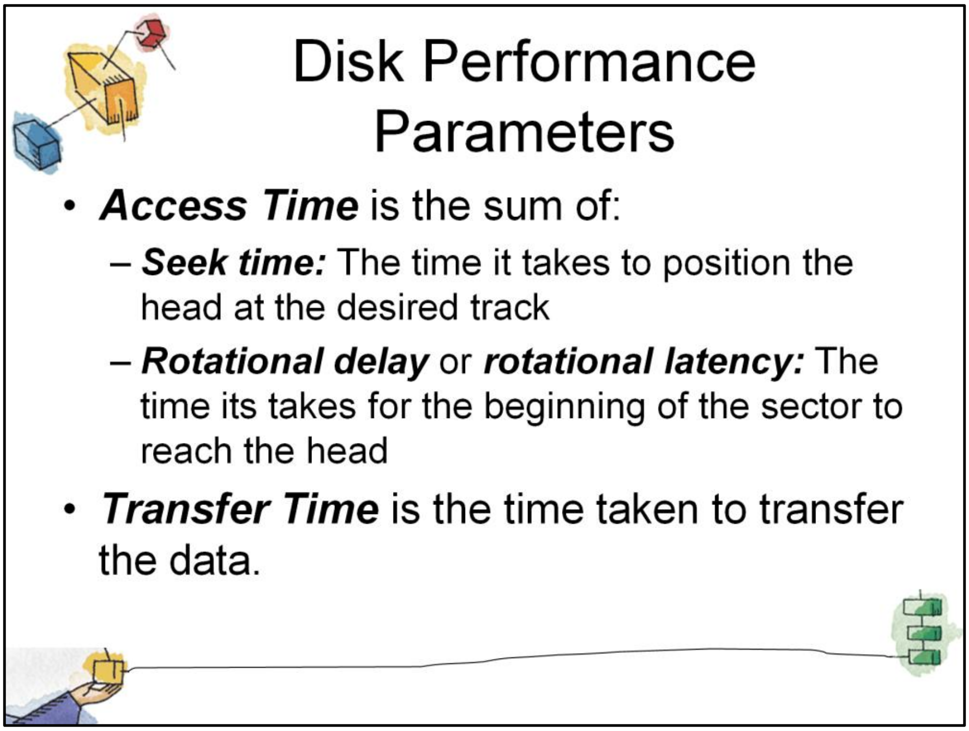
Positioning the Read/Write Heads

- 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.

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.



Disk Performance Parameters

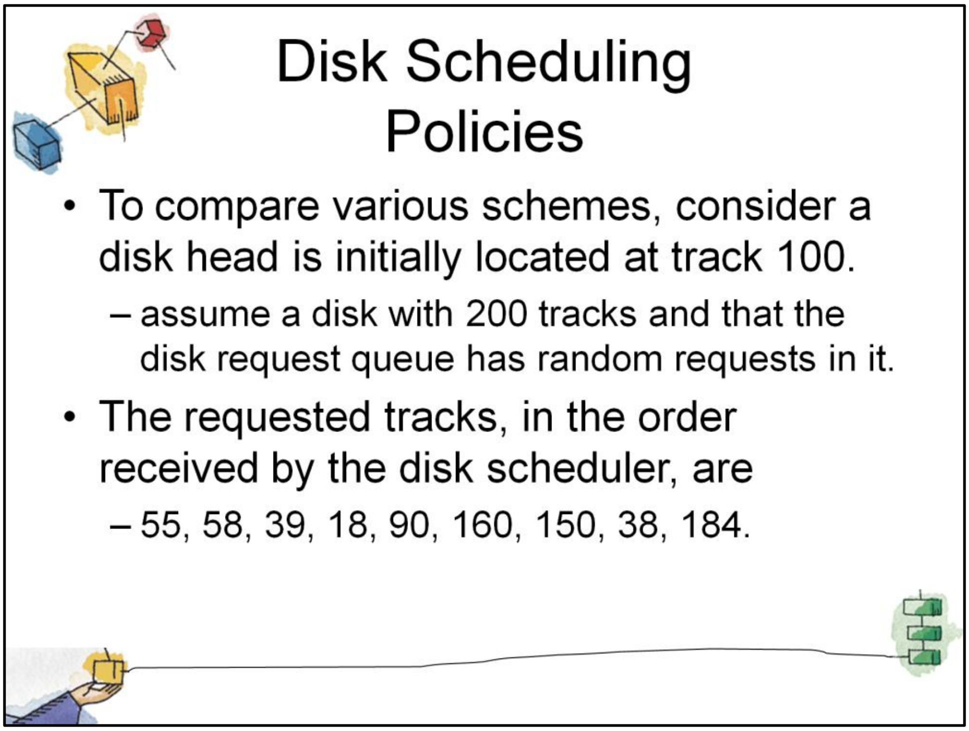
- ***Access Time*** is the sum of:
  - ***Seek time:*** The time it takes to position the head at the desired track
  - ***Rotational delay*** or ***rotational latency:*** The time its takes for the beginning of the sector to reach the head
- ***Transfer Time*** is the time taken to transfer the data.

Access Time is the sum of

- ***Seek Time*** is the time it takes to position the head at the track.
- **Rotational delay** is the time it takes for the beginning of the sector to reach the head

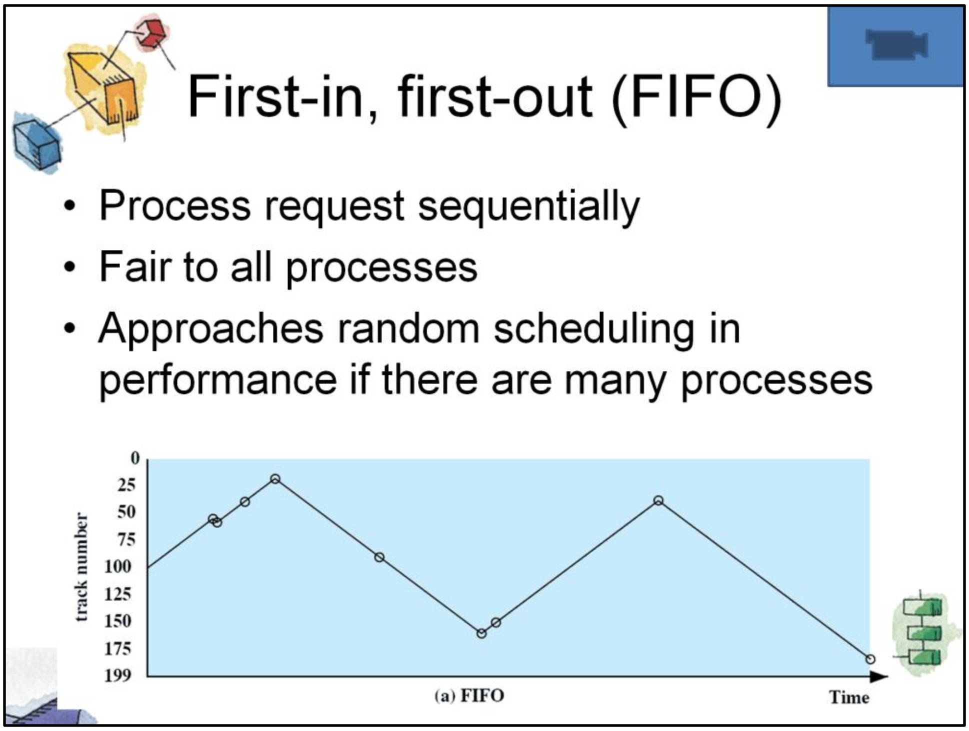
Once the head is in position, the read or write operation is then performed as the sector moves under the head;

- this is the data transfer portion of the operation; the time required for the transfer is the ***transfer time***.



Disk Scheduling Policies

- To compare various schemes, consider a disk head is initially located at track 100.
  - assume a disk with 200 tracks and that the disk request queue has random requests in it.
- The requested tracks, in the order received by the disk scheduler, are
  - 55, 58, 39, 18, 90, 160, 150, 38, 184.



First-in, first-out (FIFO)

- Process request sequentially
- Fair to all processes
- Approaches random scheduling in performance if there are many processes

Movie icon jumps to animation at
http://gaia.ecs.csus.edu/%7ezhangd/oscal/DiskApplet.html

The simplest form of scheduling is first-in-first-out (FIFO) scheduling, which processes items from the queue in sequential order.

This strategy has the advantage of being fair, because every request is honored and the requests are honored in the order received.

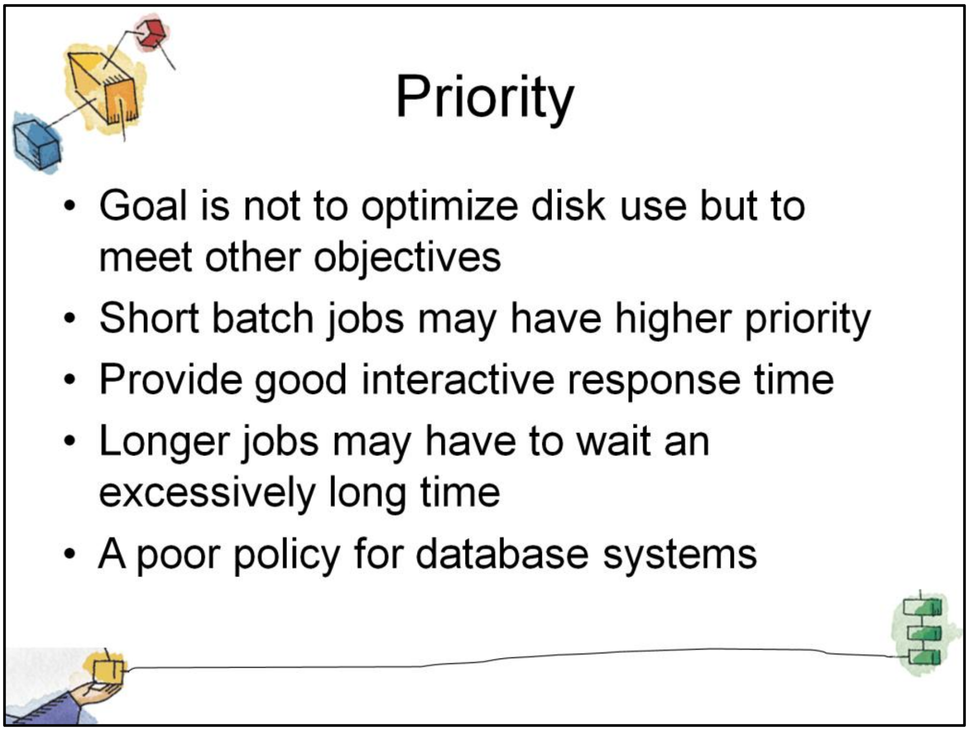
This figure illustrates the disk arm movement with FIFO.

- This graph is generated directly from the data in Table 11.2a.

As can be seen, the disk accesses are in the same order as the requests were originally received.

With FIFO, if there are only a few processes that require access and if many of the requests are to clustered file sectors, then we can hope for good performance.

- ***But***, this technique will often approximate random scheduling in performance, if there are many processes competing for the disk.



Priority

- Goal is not to optimize disk use but to meet other objectives
- Short batch jobs may have higher priority
- Provide good interactive response time
- Longer jobs may have to wait an excessively long time
- A poor policy for database systems

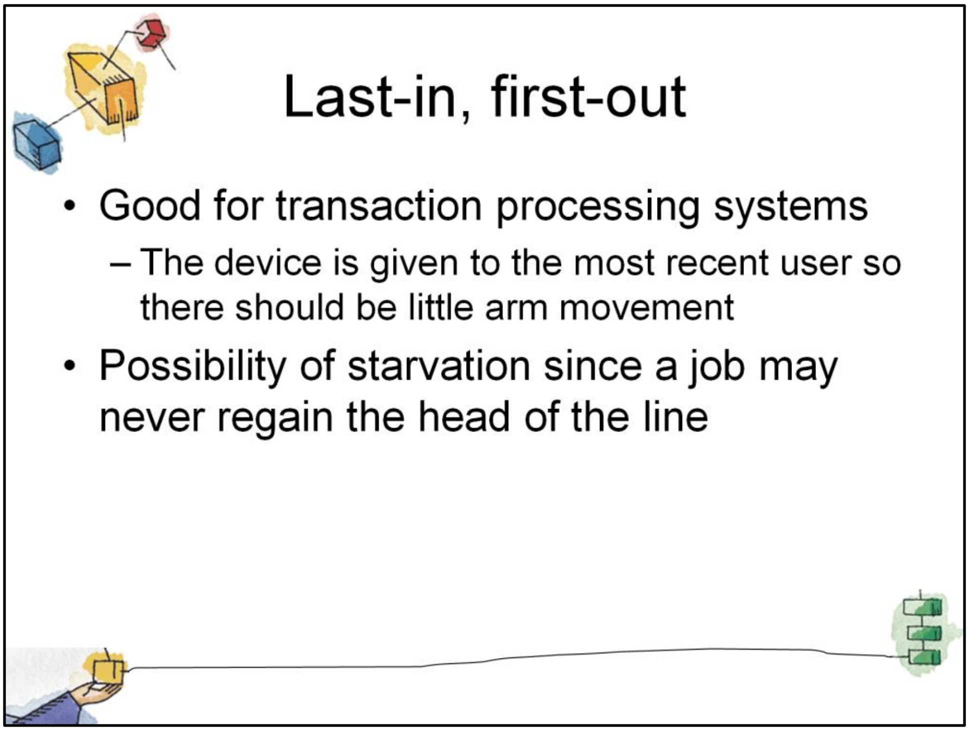
With a system based on priority (PRI), the control of the scheduling is outside the control of disk management software.

- This is not intended to optimize disk utilization but to meet other objectives within the operating system.

Often short batch jobs and interactive jobs are given higher priority than longer jobs that require longer computation.

- This allows a lot of short jobs to be flushed through the system quickly and may provide good interactive response time.
- However, longer jobs may have to wait excessively long times.
- Furthermore, such a policy could lead to countermeasures on the part of users, who split their jobs into smaller pieces to beat the system.

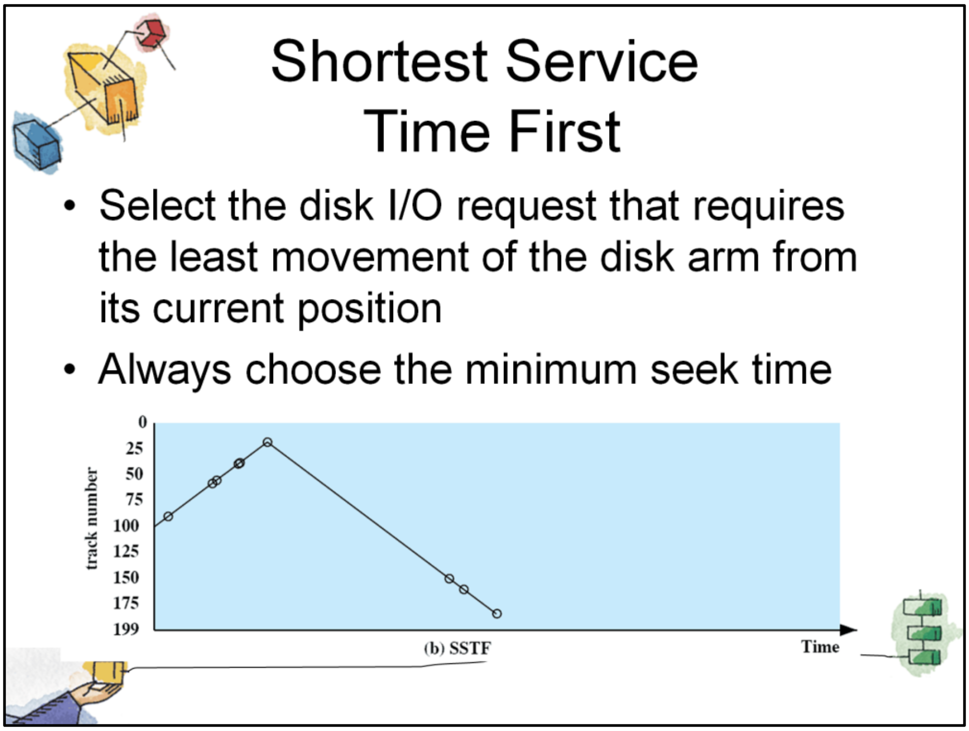
This type of policy tends to be poor for database systems.



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

In transaction processing systems, giving the device to the most recent user should result in little or no arm movement for moving through a sequential file.



Shortest Service Time First

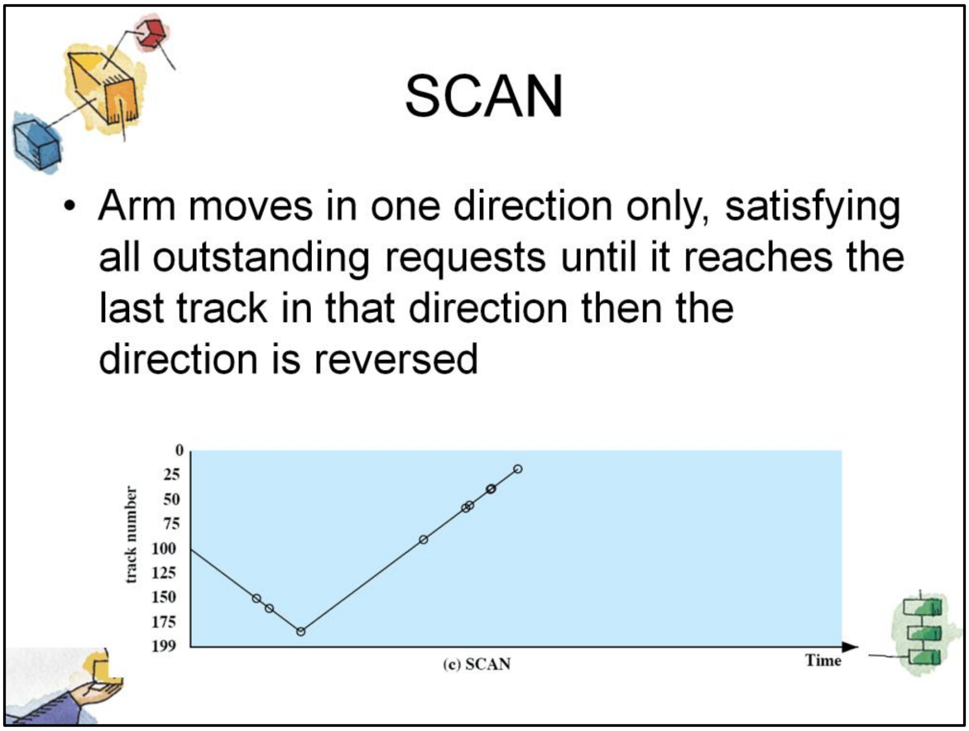
- Select the disk I/O request that requires the least movement of the disk arm from its current position
- Always choose the minimum seek time

Select the disk I/O request that requires the least movement of the disk arm from its current position.

Thus, we always choose to incur the minimum seek time.

- •Always choosing the minimum seek time does not guarantee that the average seek time over a number of arm movements will be minimum.
- •However, this should provide better performance than FIFO.

Because the arm can move in two directions, a random tie-breaking algorithm may be used to resolve cases of equal distances.



SCAN

- Arm moves in one direction only, satisfying all outstanding requests until it reaches the last track in that direction then the direction is reversed

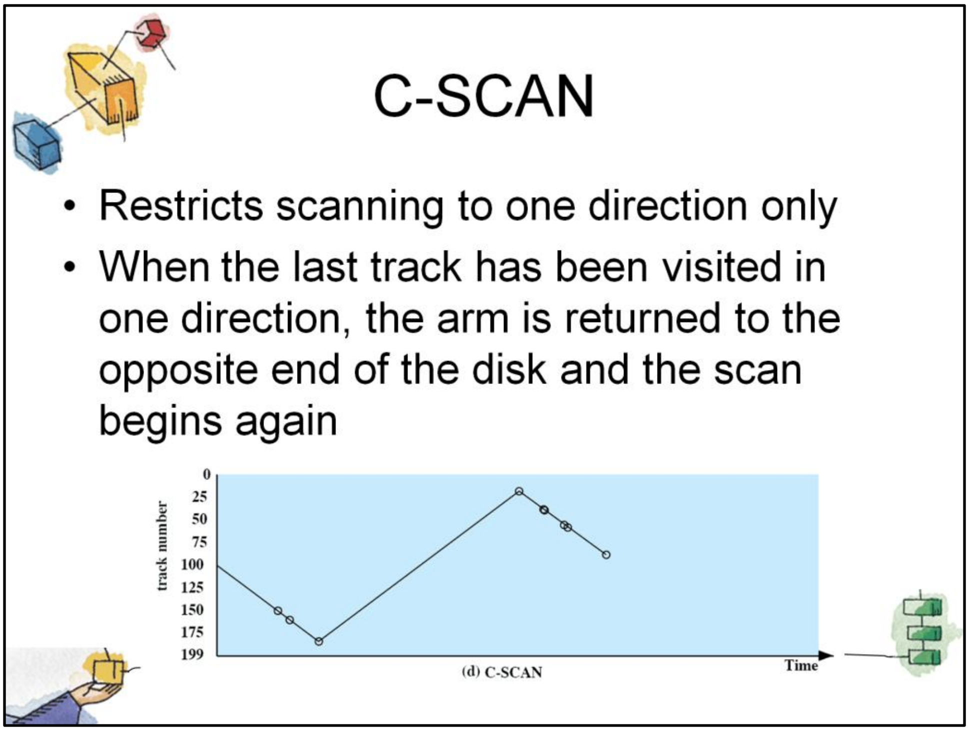
(c) SCAN

With SCAN, the arm is required to move in one direction only, satisfying all outstanding requests en route, until it reaches the last track in that direction or until there are no more requests in that direction.

- The service direction is then reversed and the scan proceeds in the opposite direction, again picking up all requests in order.

This latter refinement is sometimes referred to as the LOOK policy.

The SCAN policy favors jobs whose requests are for tracks nearest to both innermost and outermost tracks and favors the latest-arriving jobs.



C-SCAN

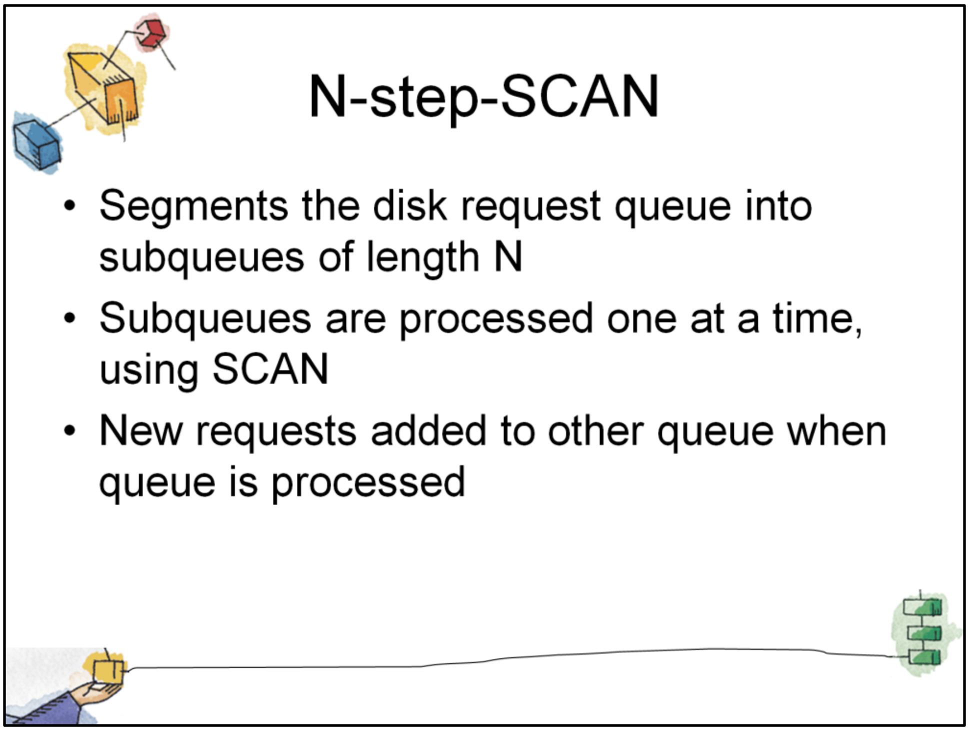
- Restricts scanning to one direction only
- When the last track has been visited in one direction, the arm is returned to the opposite end of the disk and the scan begins again

(d) C-SCAN

The C-SCAN (circular SCAN) policy restricts scanning to one direction only.

- Thus, 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.

This reduces the maximum delay experienced by new requests.



N-step-SCAN

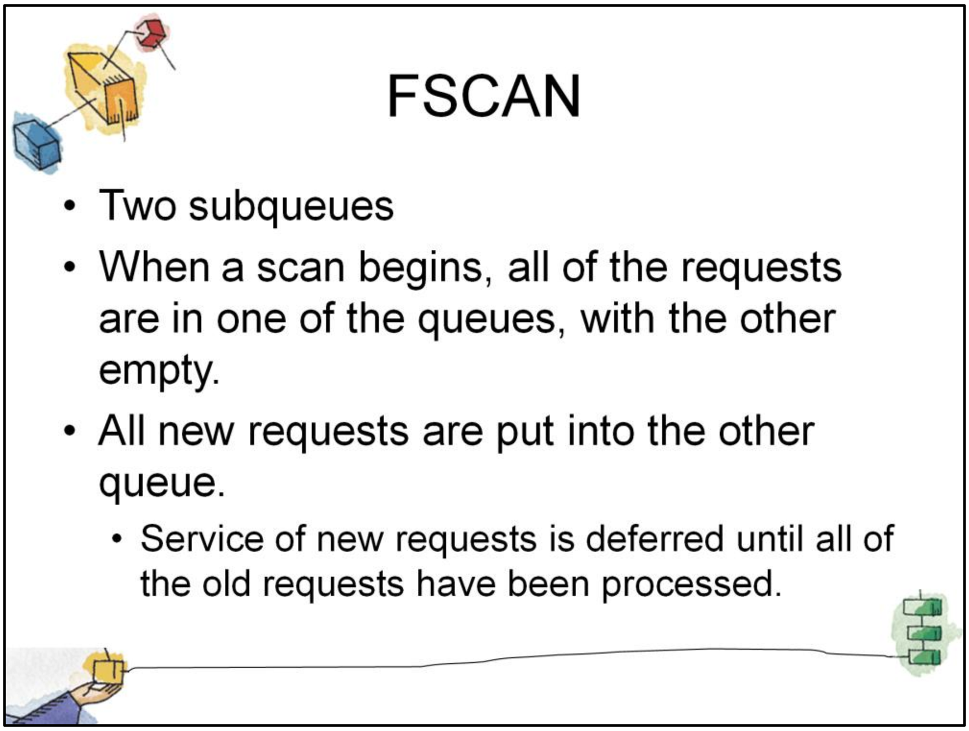
- Segments the disk request queue into subqueues of length N
- Subqueues are processed one at a time, using SCAN
- New requests added to other queue when queue is processed

The N-step-SCAN policy 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, then all of them are processed with the next scan.



FSCAN

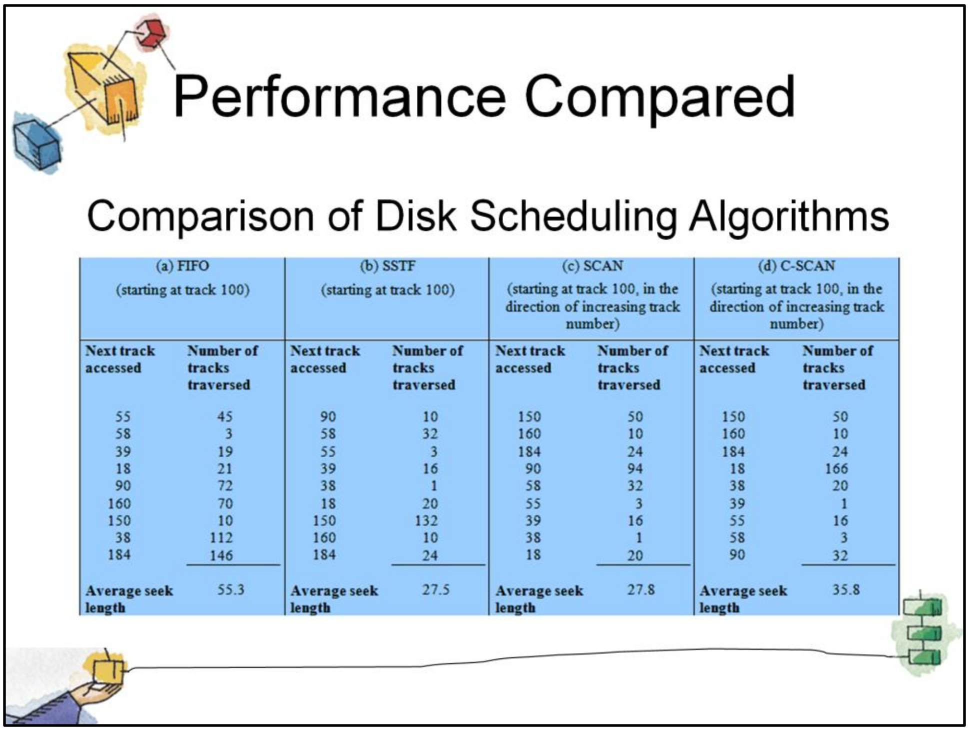
- Two subqueues
- When a scan begins, all of the requests are in one of the queues, with the other empty.
- All new requests are put into the other queue.
  - Service of new requests is deferred until all of the old requests have been processed.

FSCAN is a policy that uses two subqueues.

When a scan begins, all of the requests are in one of the queues, with the other empty.

During the scan, all new requests are put into the other queue.

- Thus, service of new requests is deferred until all of the old requests have been processed.

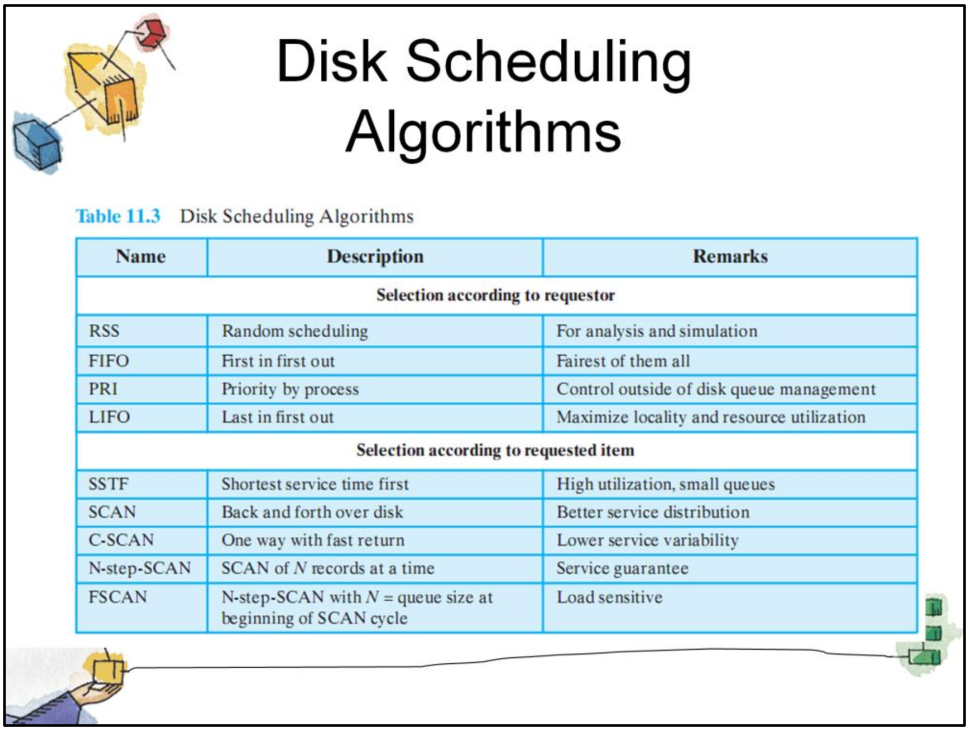


Performance Compared

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

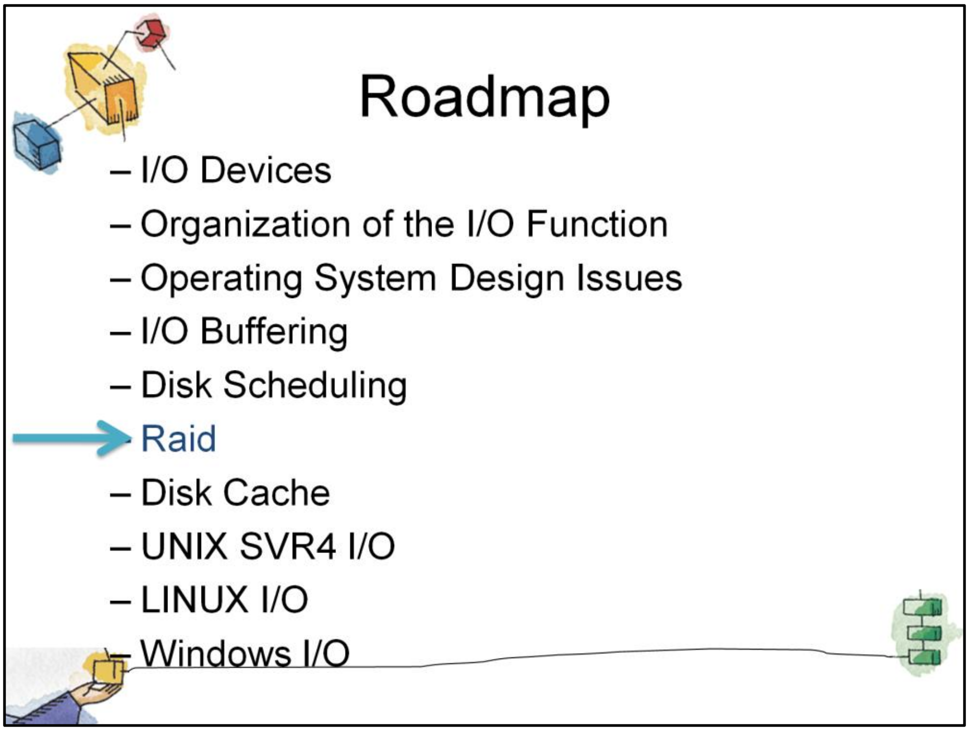
T



Disk Scheduling Algorithms

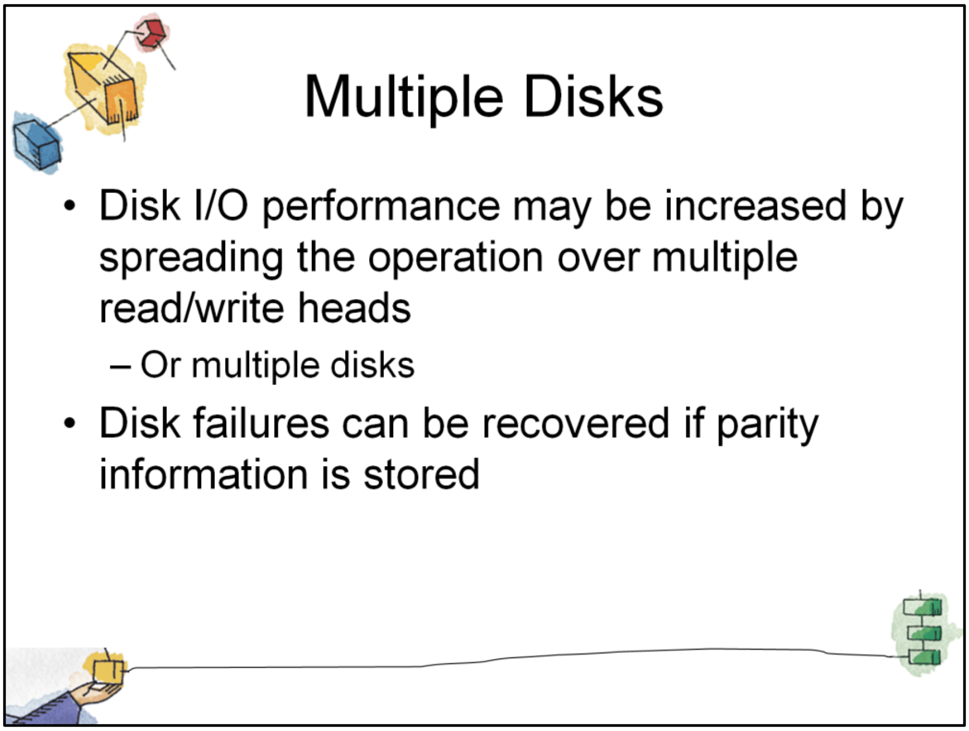
Table 11.3 Disk Scheduling Algorithms

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



Roadmap

- I/O Devices
- Organization of the I/O Function
- Operating System Design Issues
- I/O Buffering
- Disk Scheduling
- → Raid
- Disk Cache
- UNIX SVR4 I/O
- LINUX I/O
- Windows I/O

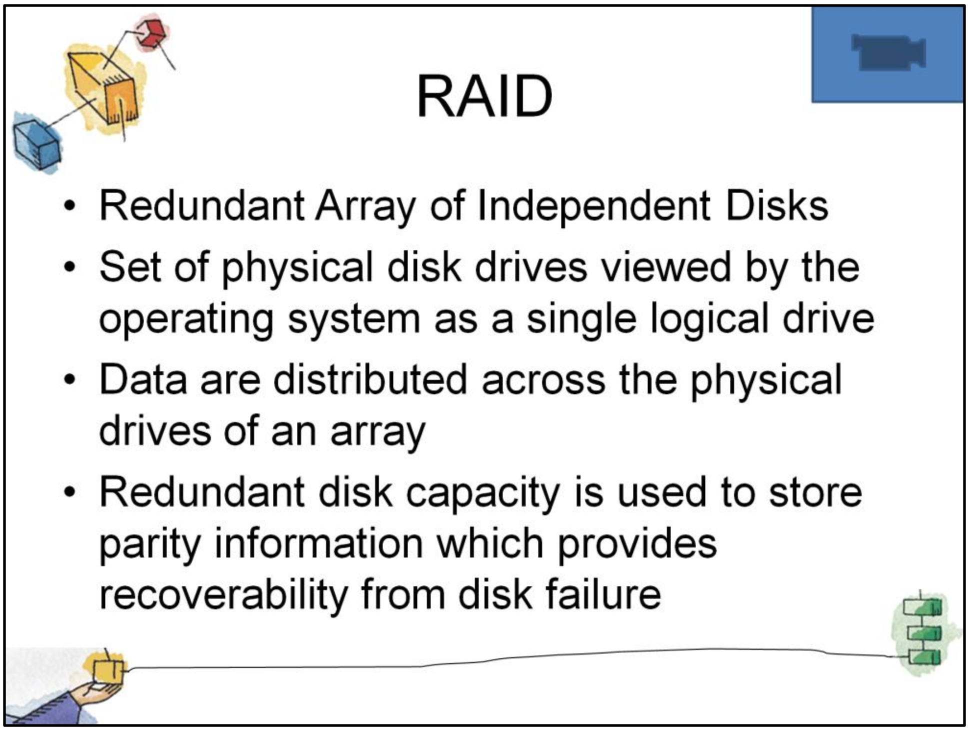


Multiple Disks

- Disk I/O performance may be increased by spreading the operation over multiple read/write heads
  - Or multiple disks
- Disk failures can be recovered if parity information is stored

With multiple disks, separate I/O requests can be handled in parallel, as long as the data required reside on separate disks.

Also, a single I/O request can be executed in parallel if the block of data to be accessed is distributed across multiple disks.



RAID

- Redundant Array of Independent Disks
- 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 which provides recoverability from disk failure

Movie icon links to animation at:
http://gaia.ecs.csus.edu/%7ezhangd/oscal/RAIDFiles/RAID.htm

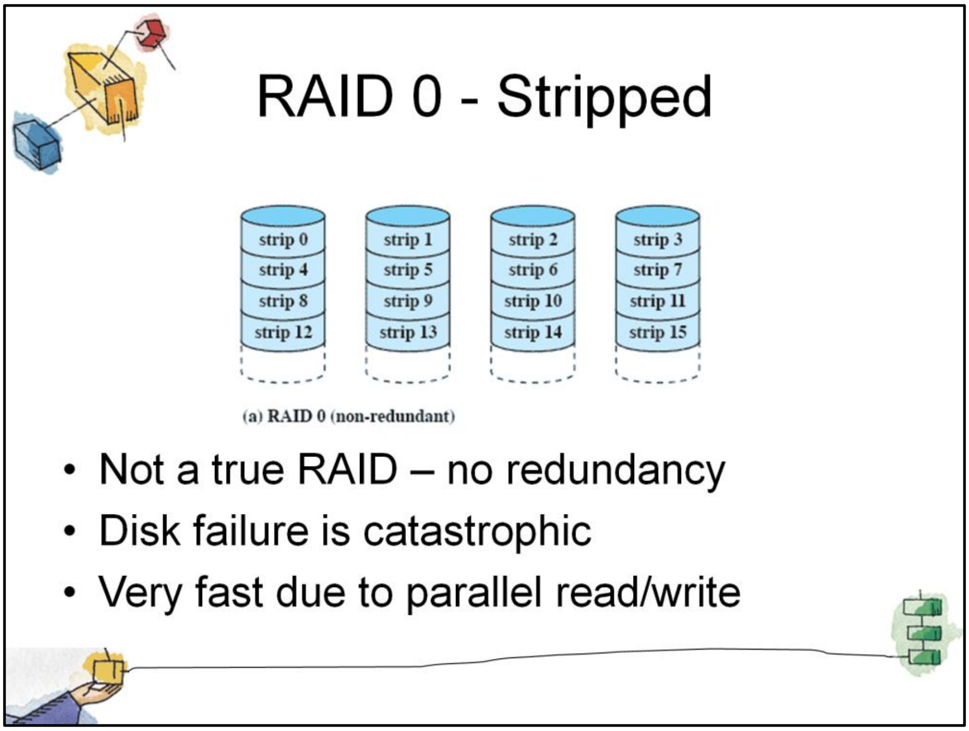
The RAID scheme consists of seven levels, zero through six.

These levels do not imply a hierarchical relationship but designate different design architectures that share three common characteristics:

1. RAID is a set of physical disk drives viewed by the operating system as a single logical drive.

2. Data are distributed across the physical drives of an array in a scheme known as striping, described subsequently.

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



strip 0
strip 4
strip 8
strip 12
strip 1
strip 5
strip 9
strip 13
strip 2
strip 6
strip 10
strip 14
strip 3
strip 7
strip 11
strip 15

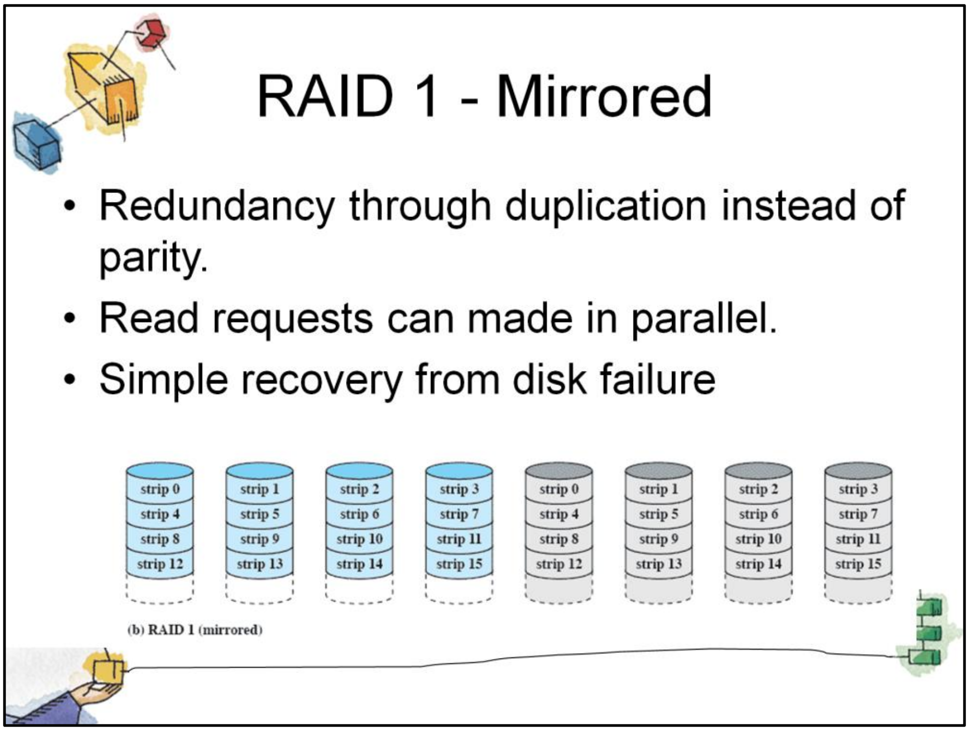
RAID 0 - Stripped

(a) RAID 0 (non-redundant)

- Not a true RAID – no redundancy
- Disk failure is catastrophic
- Very fast due to parallel read/write

RAID level 0 is not a true member of the RAID family, because it does not include redundancy.

The advantage of this layout is that if a single I/O request consists of multiple logically contiguous strips, then up to n strips for that request can be handled in parallel, greatly reducing the I/O transfer time.



RAID 1 - Mirrored

- Redundancy through duplication instead of parity.
- Read requests can made in parallel.
- Simple recovery from disk failure

(b) RAID 1 (mirrored)

each logical strip is mapped to two separate physical disks so that every disk in the array has a mirror disk that contains the same data.

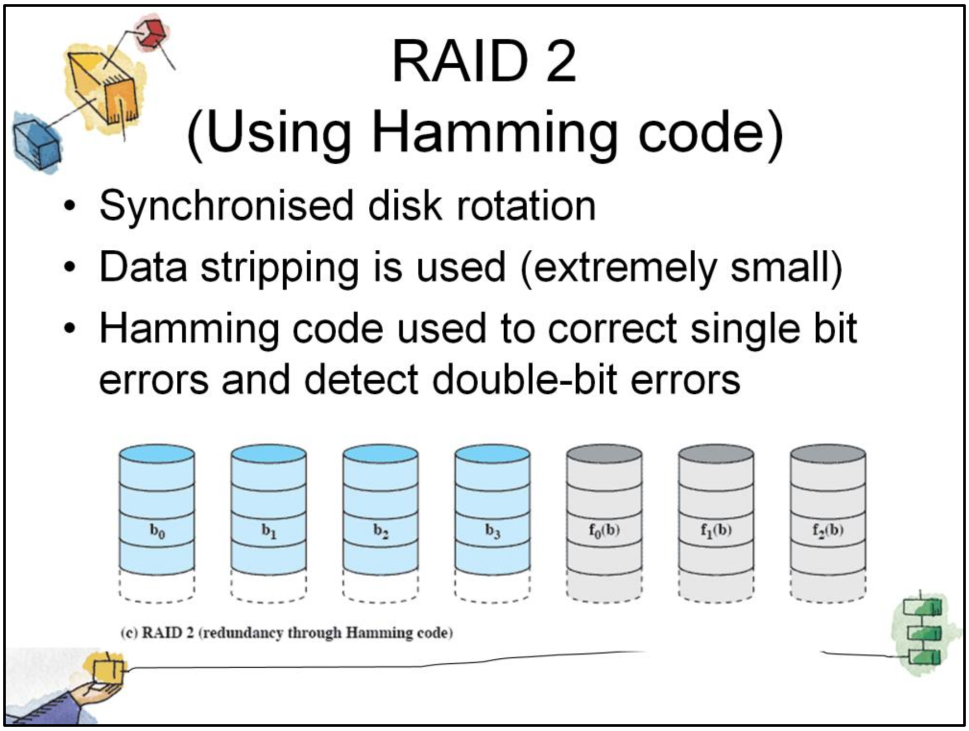
A read request can be serviced by either of the two disks that contains the requested data, whichever one involves the minimum seek time plus rotational latency.

A write request requires that both corresponding strips be updated, but this can be done in parallel.

- Thus, the write performance is dictated by the slower of the two writes

Recovery from a failure is simple.

- When a drive fails, the data may still be accessed from the second drive.



RAID 2
(Using Hamming code)

- Synchronised disk rotation
- Data stripping is used (extremely small)
- Hamming code used to correct single bit errors and detect double-bit errors

(c) RAID 2 (redundancy through Hamming code)

In a parallel access array, all member disks participate in the execution of every I/O request.

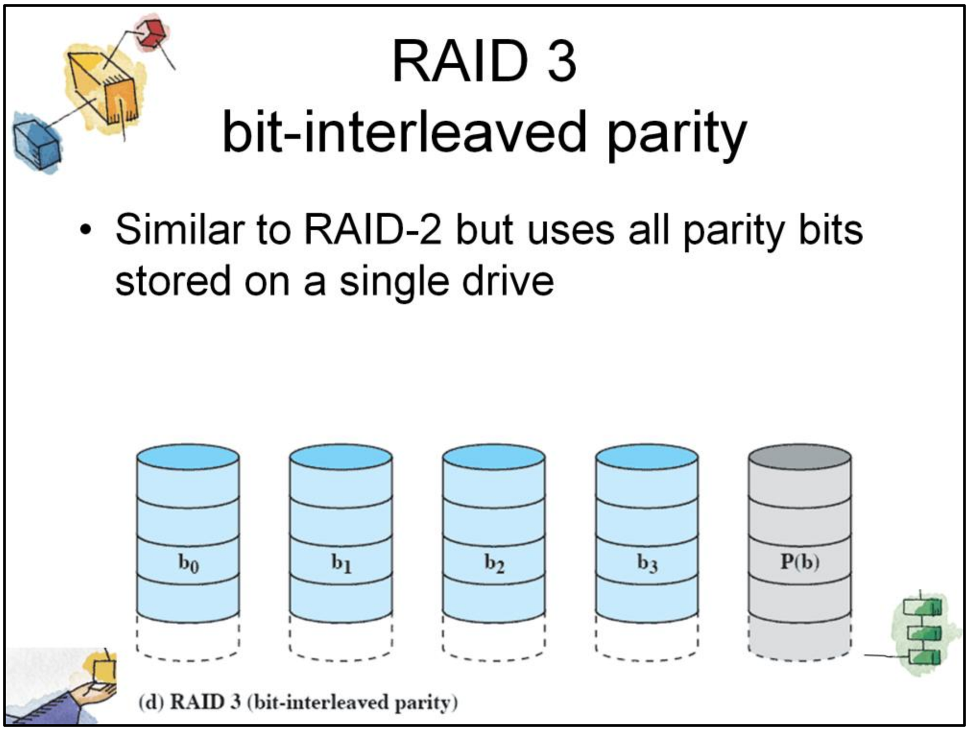
- Typically, the spindles of the individual drives are synchronized so that each disk head is in the same position on each disk at any given time.

As in the other RAID schemes, data striping is used.

- In the case of RAID 2 and 3, the strips are very small, often as small as a single byte or word.

With RAID 2, an error-correcting code is calculated across corresponding bits on each data disk, and the bits of the code are stored in the corresponding bit positions on multiple parity disks.

Typically, a Hamming code is used, which is able to correct single-bit errors and detect double-bit errors.



RAID 3 bit-interleaved parity

- Similar to RAID-2 but uses all parity bits stored on a single drive

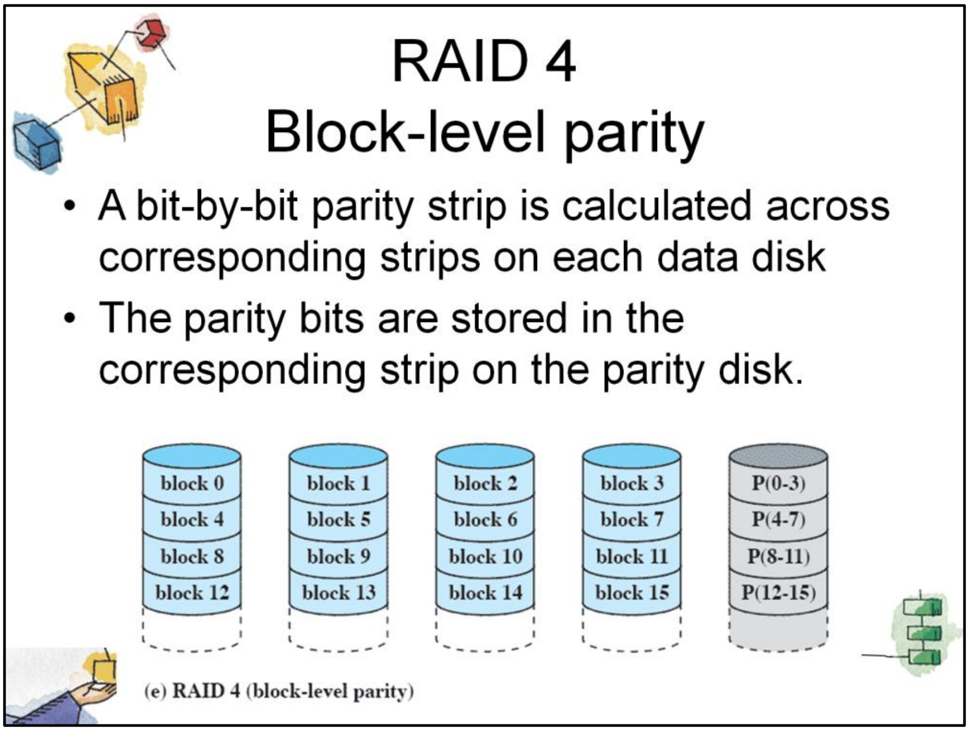
(d) RAID 3 (bit-interleaved parity)

RAID 3 is organized in a similar fashion to RAID 2.

The difference is that RAID 3 requires only a single redundant disk, no matter how large the disk array.

RAID 3 employs parallel access, with data distributed in small strips.

Instead of an error-correcting code, a simple parity bit is computed for the set of individual bits in the same position on all of the data disks.



RAID 4
Block-level parity

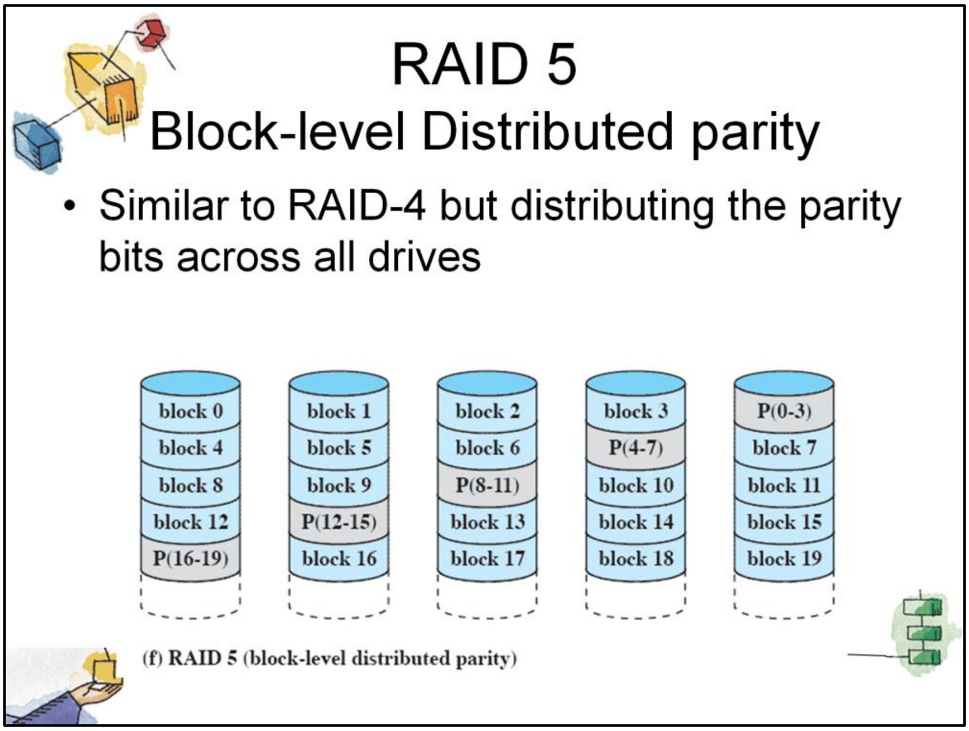
- A bit-by-bit parity strip is calculated across corresponding strips on each data disk
- The parity bits are stored in the corresponding strip on the parity disk.

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

(e) RAID 4 (block-level parity)

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.



RAID 5
Block-level Distributed parity

- Similar to RAID-4 but distributing the parity bits across all drives

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

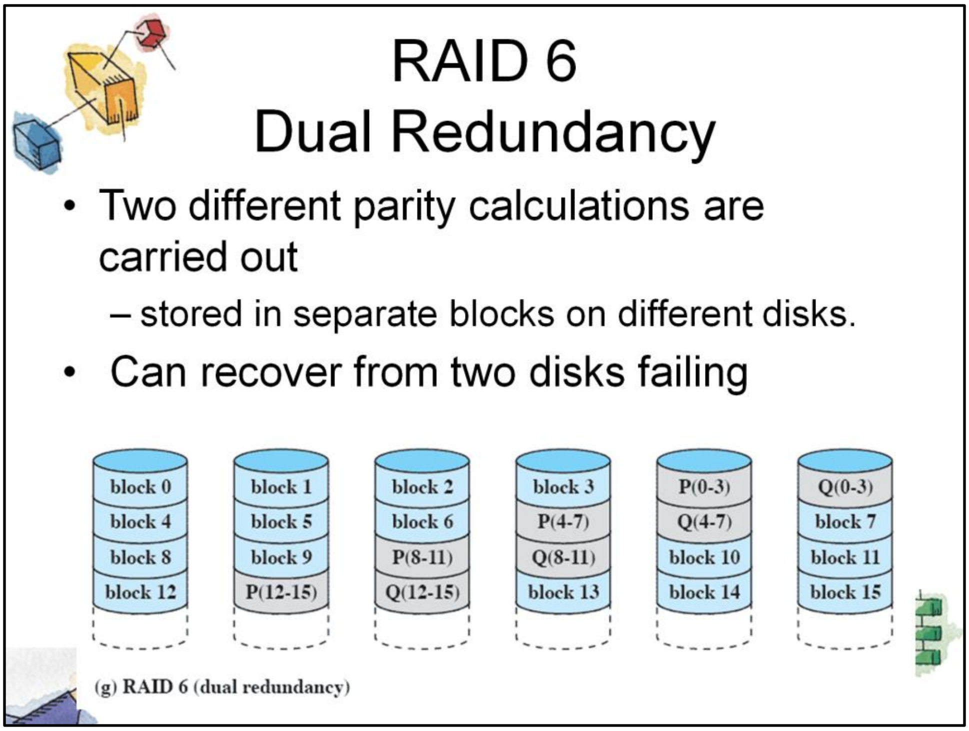
(f) RAID 5 (block-level distributed parity)

RAID 5 is organized in a similar fashion to RAID 4.

RAID 5 distributes the parity strips across all disks.

- A typical allocation is a round-robin scheme
- For an n-disk array, the parity strip is on a different disk for the first n stripes, and the pattern then repeats.

The distribution of parity strips across all drives avoids the potential I/O bottleneck of the single parity disk found in RAID 4.



RAID 6
Dual Redundancy

- Two different parity calculations are carried out
  - stored in separate blocks on different disks.
- Can recover from two disks failing

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)

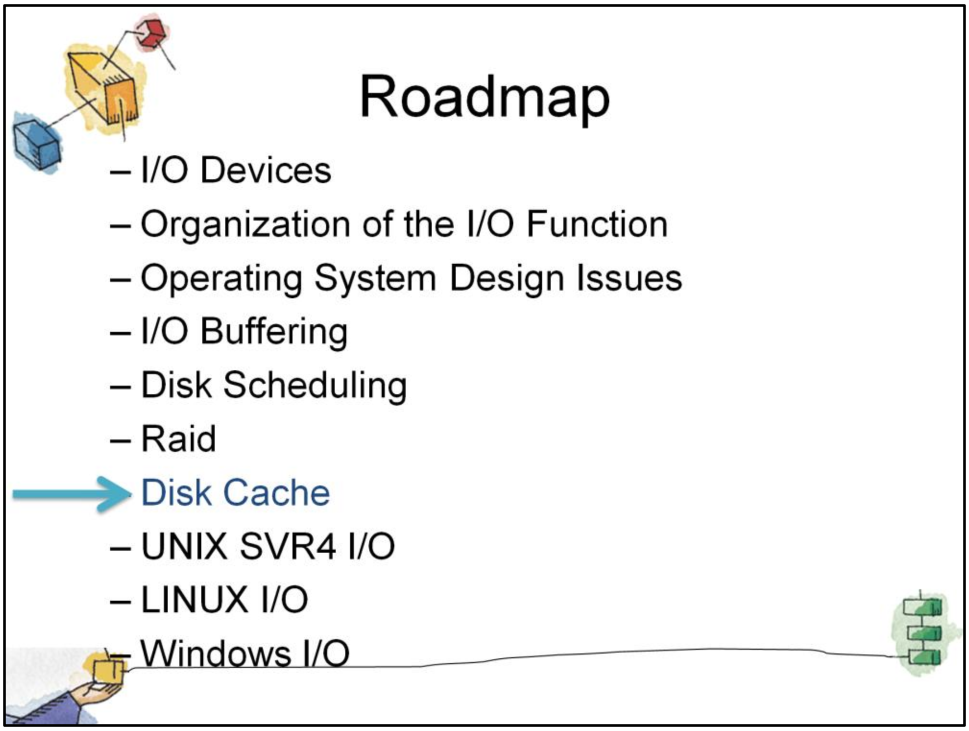
Two different parity calculations are carried out and stored in separate blocks on different disks.

- Thus, a RAID 6 array whose user data require N disks consists of N+2 disks.

P and Q are two different data check algorithms.

- One of the two is the exclusive-OR calculation used in RAID 4 and 5.
- The other is an independent data check algorithm.

This makes it possible to regenerate data even if two disks containing user data fail.

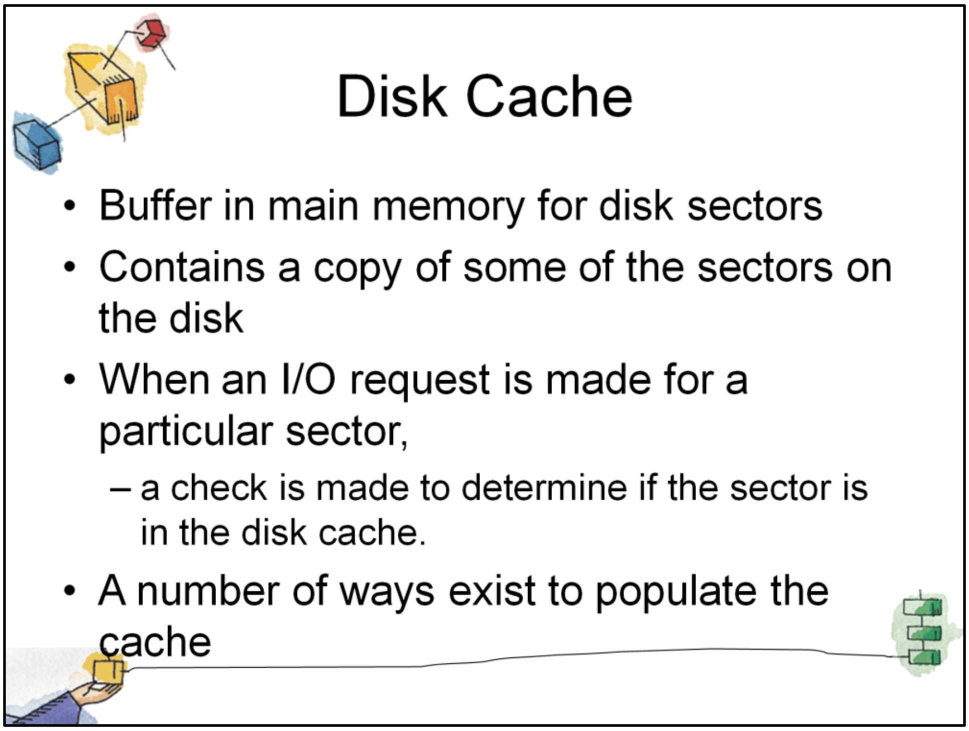


Roadmap

- I/O Devices
- Organization of the I/O Function
- Operating System Design Issues
- I/O Buffering
- Disk Scheduling
- Raid

→ Disk Cache

- UNIX SVR4 I/O
- LINUX I/O
- Windows I/O



Disk Cache

- 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.
- A number of ways exist to populate the cache

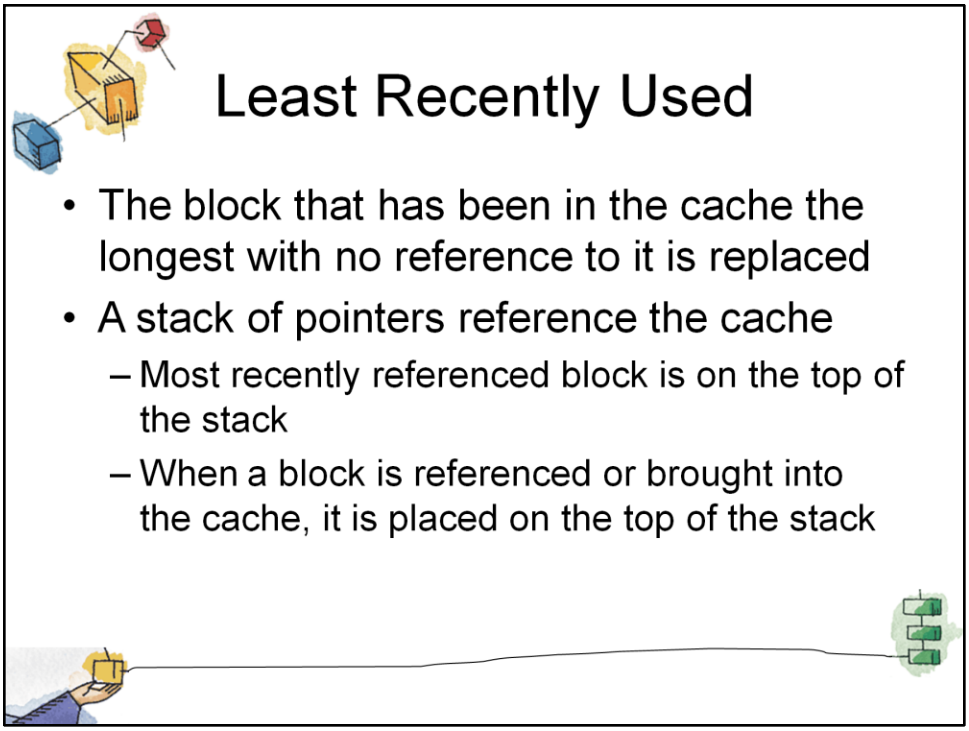
A disk cache is a buffer in main memory for disk sectors.

The cache 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 so, the request is satisfied via the cache.
- If not, the requested sector is read into the disk cache from the disk.

Because of the phenomenon of locality of reference, when a block of data is fetched into the cache to satisfy a single I/O request, it is likely that there will be future references to that same block.



Least Recently Used

- 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

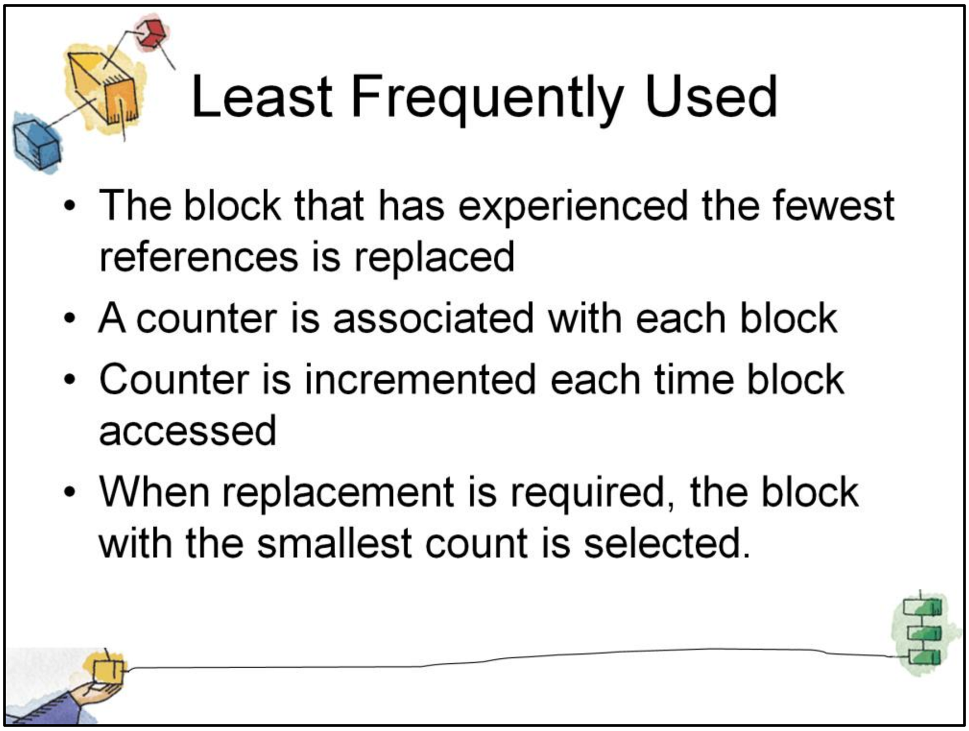
The most commonly used algorithm is **least recently used (LRU)**

Replace that block that has been in the cache longest with no reference to it.

The cache consists of a stack of blocks, with the most recently referenced block on the top of the stack.

- When a block in the cache is referenced, it is moved from its existing position on the stack to the top of the stack.
- When a block is brought in from secondary memory, remove the block that is on the bottom of the stack and push the incoming block onto the top of the stack.

It is not necessary actually to move these blocks around in main memory; a stack of pointers can be associated with the cache.



Least Frequently Used

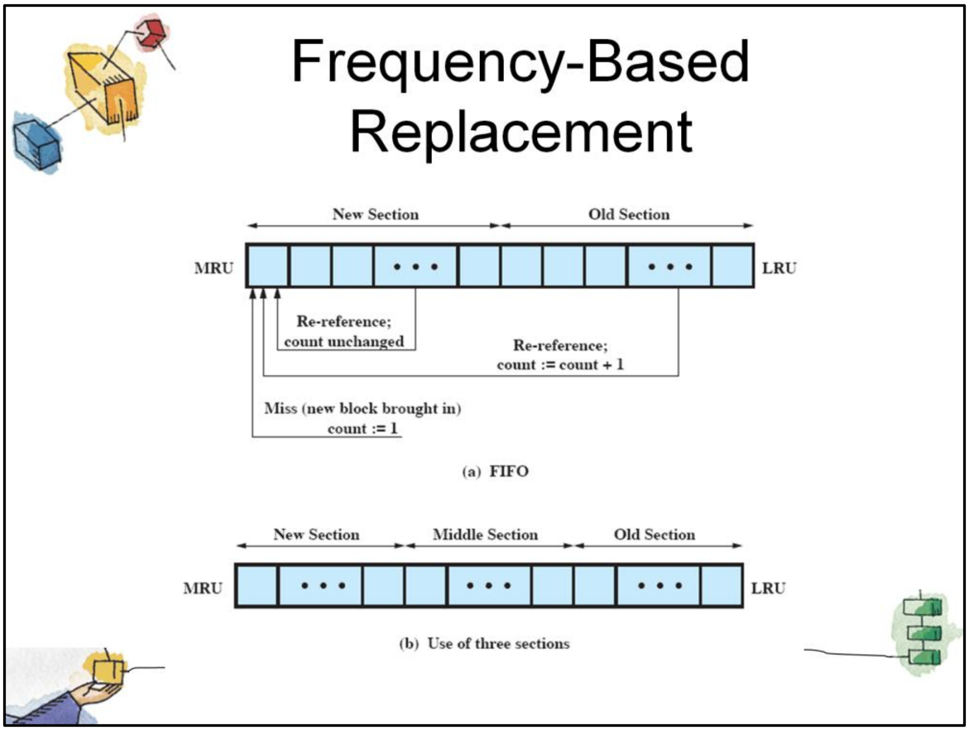
- The block that has experienced the fewest references is replaced
- A counter is associated with each block
- Counter is incremented each time block accessed
- When replacement is required, the block with the smallest count is selected.

Replace that block in the set that has experienced the fewest references.

LFU could be implemented by associating a counter with each block.

- When a block is brought in, it is assigned a count of 1;
- with each reference to the block, its count is incremented by 1.
- When replacement is required, the block with the smallest count is selected.

Intuitively, it might seem that LFU is more appropriate than LRU because LFU makes use of more pertinent information about each block in the selection process.



Frequency-Based Replacement

(a) FIFO

(b) Use of three sections

The blocks are logically organized in a stack, as with the LRU algorithm.

A certain portion of the top part of the stack is designated the new section.

When there is a cache hit, the referenced block is moved to the top of the stack.

- If the block was already in the new section, its reference count is not incremented;
- otherwise it is incremented by 1.

Given a sufficiently large new section, this results in the reference counts for blocks that are repeatedly re-referenced within a short interval remaining unchanged.

On a miss, the block with the smallest reference count that is not in the new section is chosen for replacement;

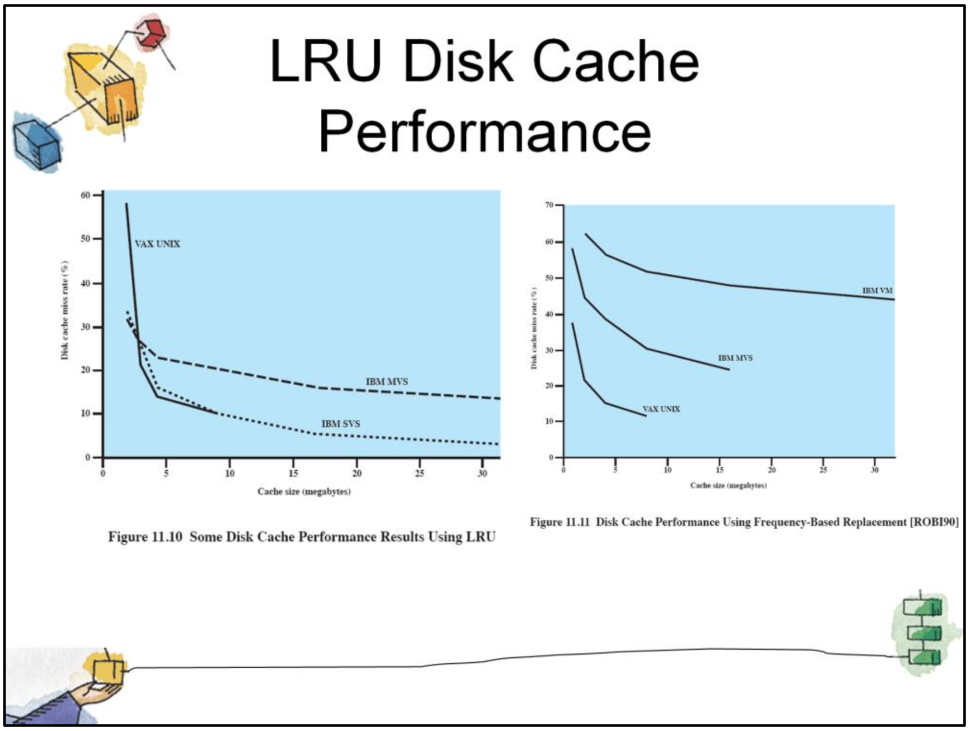
- the least recently used such block is chosen in the event of a tie.

A further refinement (Figure 11.9b):

Divide the stack into three sections: new, middle, and old.

As before, reference counts are not incremented on blocks in the new section.

- However, only blocks in the old section are eligible for replacement.
- Assuming a sufficiently large middle section, this allows relatively frequently referenced blocks a chance to build up their reference counts before becoming eligible for replacement.



LRU Disk Cache Performance

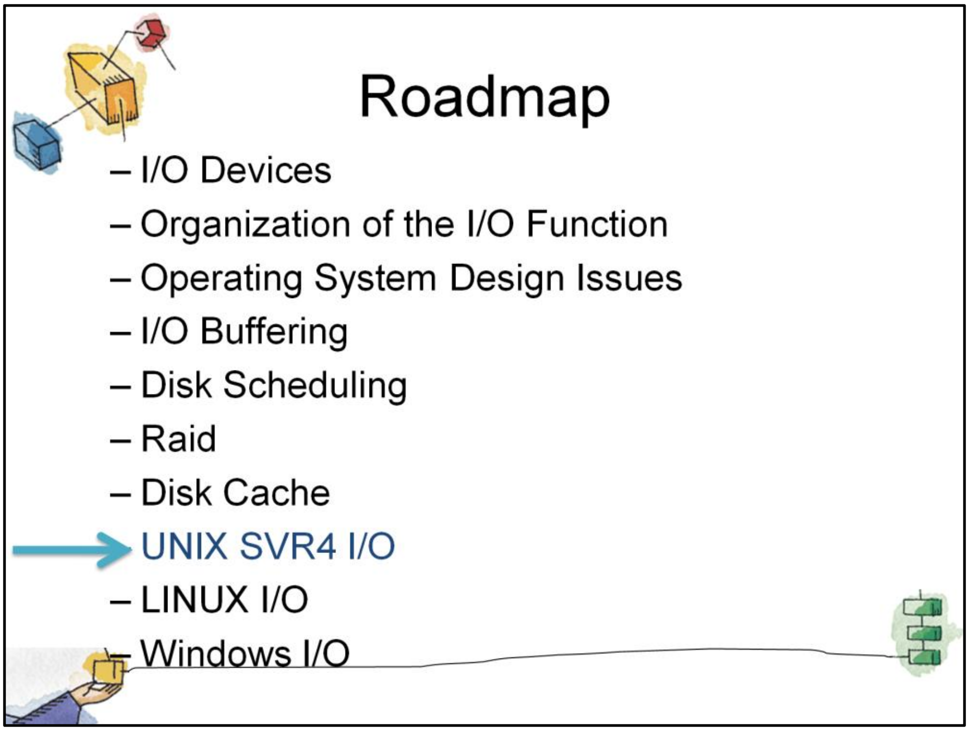
Figure 11.10 Some Disk Cache Performance Results Using LRU

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

- Figure 11.10 summarizes results from several studies using LRU, one for a UNIX system running on a VAX
- Figure 11.11 shows results for simulation studies of the frequency-based replacement algorithm.

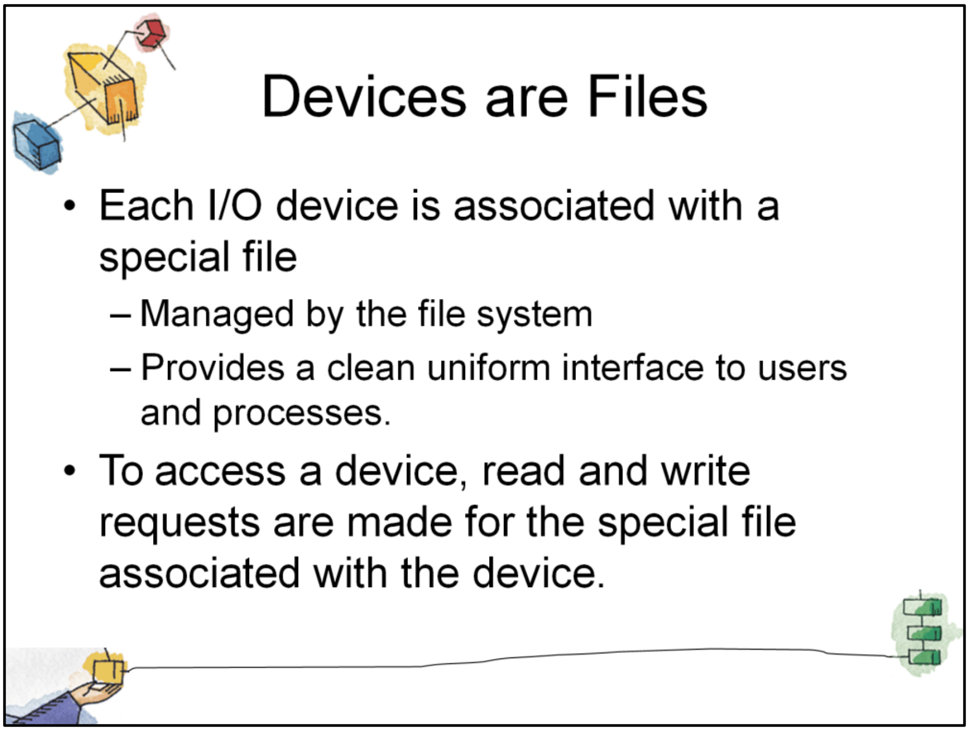
A comparison of the two figures points out one of the risks of this sort of performance assessment.

- The figures appear to show that LRU outperforms the frequency-based replacement algorithm.
- However, when identical reference patterns using the same cache structure are compared, the frequency-based replacement algorithm is superior.
- Thus, the exact sequence of reference patterns, plus related design issues such as block size, will have a profound influence on the performance achieved.



Roadmap

- – I/O Devices
- – Organization of the I/O Function
- – Operating System Design Issues
- – I/O Buffering
- – Disk Scheduling
- – Raid
- – Disk Cache
- UNIX SVR4 I/O
- – LINUX I/O
- – Windows I/O



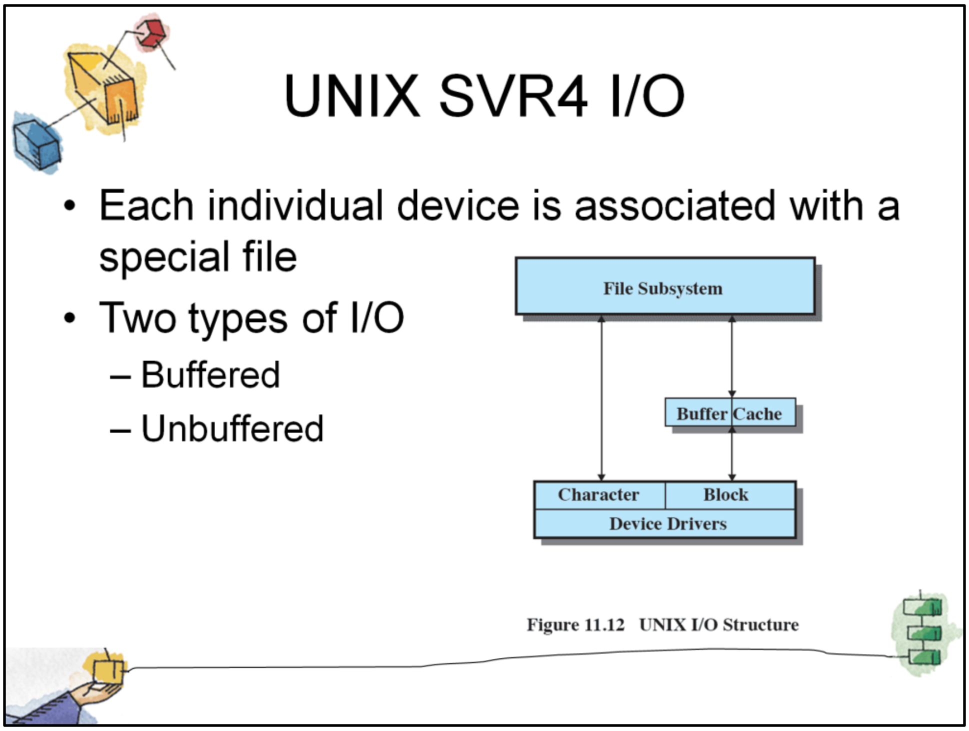
Devices are Files

- Each I/O device is associated with a special file
  - Managed by the file system
  - Provides a clean uniform interface to users and processes.
- To access a device, read and write requests are made for the special file associated with the device.

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.



UNIX SVR4 I/O

- Each individual device is associated with a special file
- Two types of I/O
  - Buffered
  - Unbuffered

Figure 11.12 UNIX I/O Structure

There are two types of I/O in UNIX:

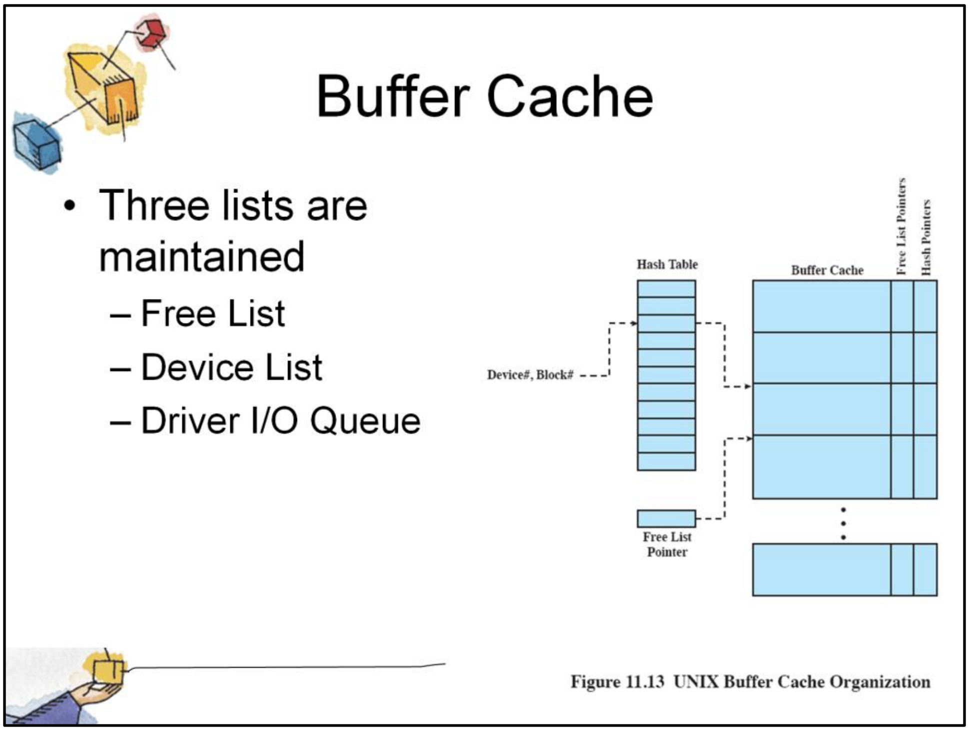
- buffered and
- unbuffered.

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.



Buffer Cache

- Three lists are maintained
  - Free List
  - Device List
  - Driver I/O Queue

Figure 11.13 UNIX Buffer Cache Organization

The buffer cache in UNIX is essentially a disk cache.

I/O operations with disk are handled through the buffer cache.

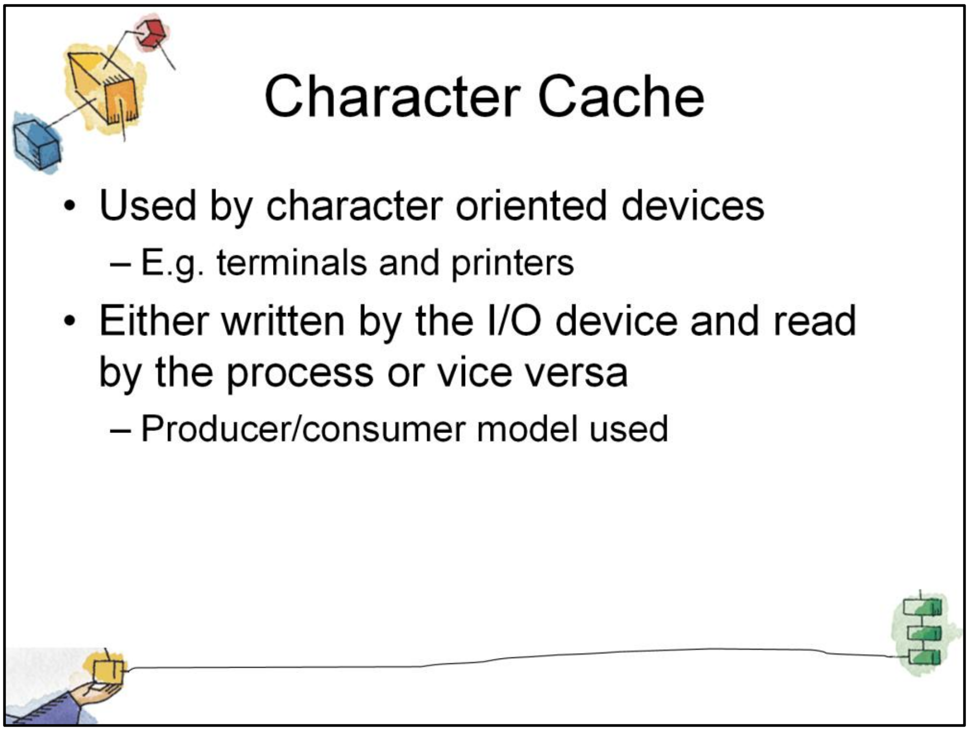
The data transfer between the buffer cache and the user process space always occurs using DMA.

Because both the buffer cache and the process I/O area are in main memory, the DMA facility is used to perform a memory-to-memory copy.

- This does not use up any processor cycles, but it does consume bus cycles.

To manage the buffer cache, three lists are maintained:

- **Free list:** List of all slots in the cache (a slot is referred to as a buffer in UNIX; each slot holds one disk sector) 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



Character Cache

- Used by character oriented devices
  - E.g. terminals and printers
- Either written by the I/O device and read by the process or vice versa
  - Producer/consumer model used

Block-oriented devices, such as disk and USB keys, can be effectively served by the buffer cache.

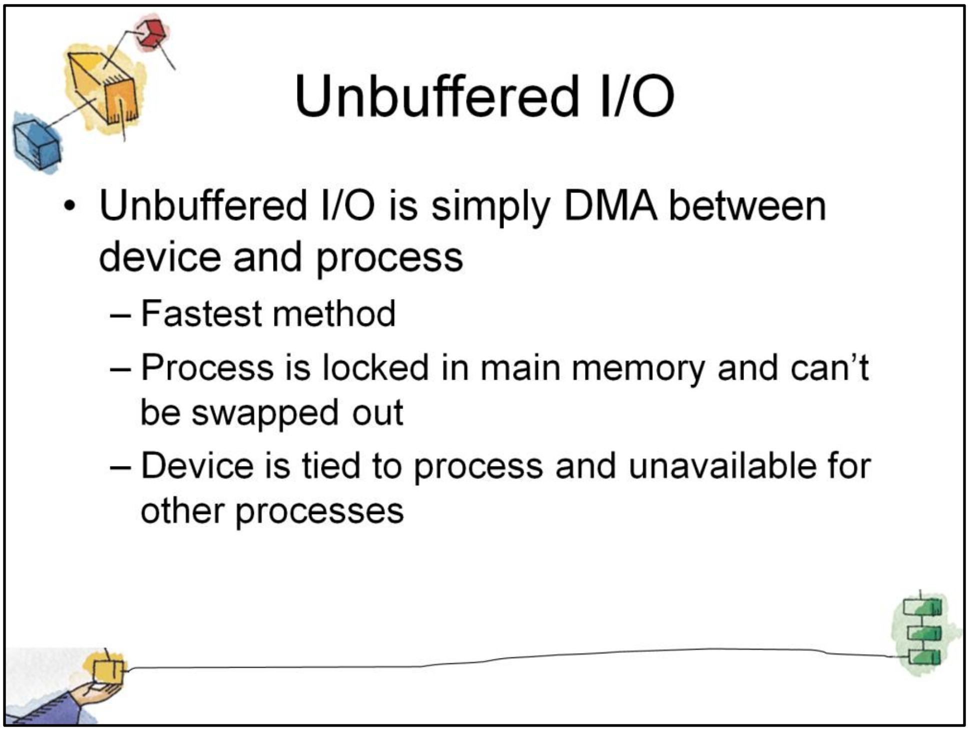
A different form of buffering is appropriate for character-oriented devices, such as terminals and printers.

A character queue is either written by the I/O device and read by the process or written by the process and read by the device.

In both cases, the producer/consumer model introduced in Chapter 5 is used.

- Thus, character queues may only be read once; as each character is read, it is effectively destroyed.

This is in contrast to the buffer cache, which may be read multiple times and hence follows the readers/writers model



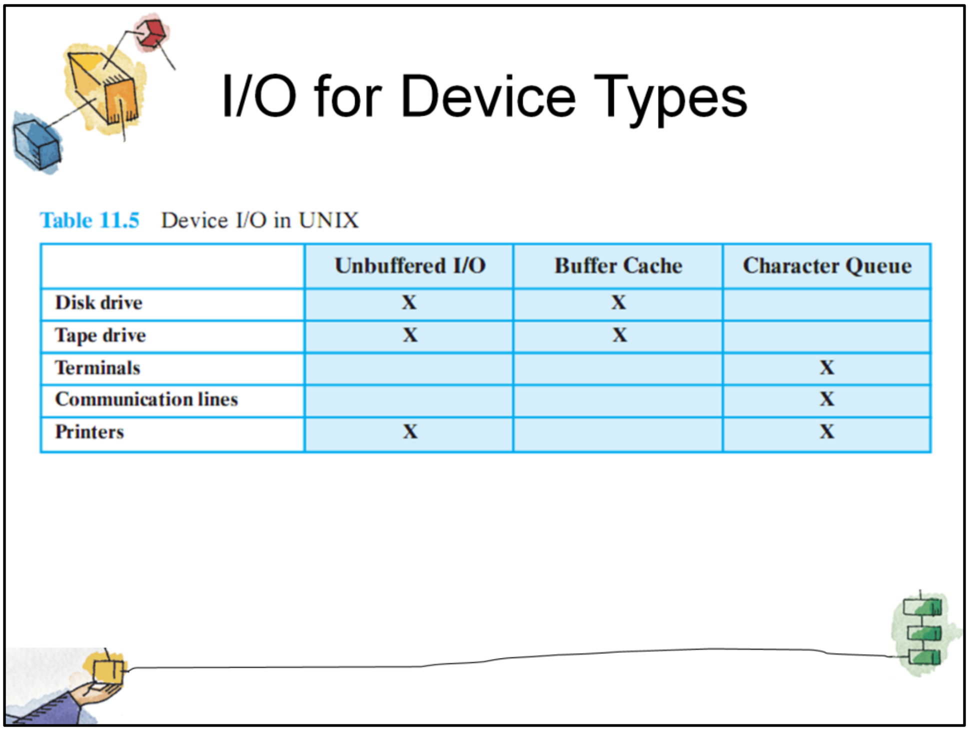
Unbuffered I/O

- Unbuffered I/O is simply DMA between device and process
  - Fastest method
  - Process is locked in main memory and can't be swapped out
  - Device is tied to process and unavailable for other processes

Unbuffered I/O, which is simply DMA between device and process space, is always the fastest method for a process to perform I/O.

A process that is performing unbuffered I/O is locked in main memory and cannot be swapped out.

- This reduces the opportunities for swapping by tying up part of main memory, thus reducing the overall system performance.
- Also, the I/O device is tied up with the process for the duration of the transfer, making it unavailable for other processes.



I/O for Device Types

Table 11.5 Device I/O in UNIX

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

This figure shows the types of I/O suited to each type of device.

Disk drives are block oriented, and have the potential for reasonable high throughput.

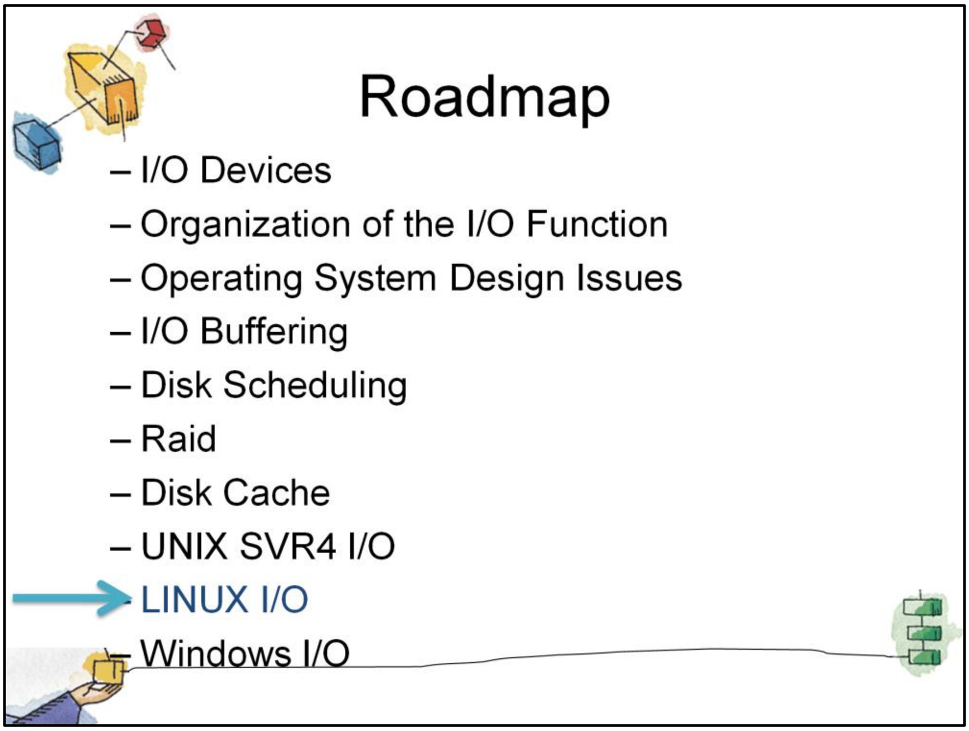
- Thus, I/O for these devices tends to be unbuffered or via buffer cache.

Tape drives are functionally similar to disk drives and use similar I/O schemes.

Because terminals involve relatively slow exchange of characters, terminal I/O typically makes use of the character queue.

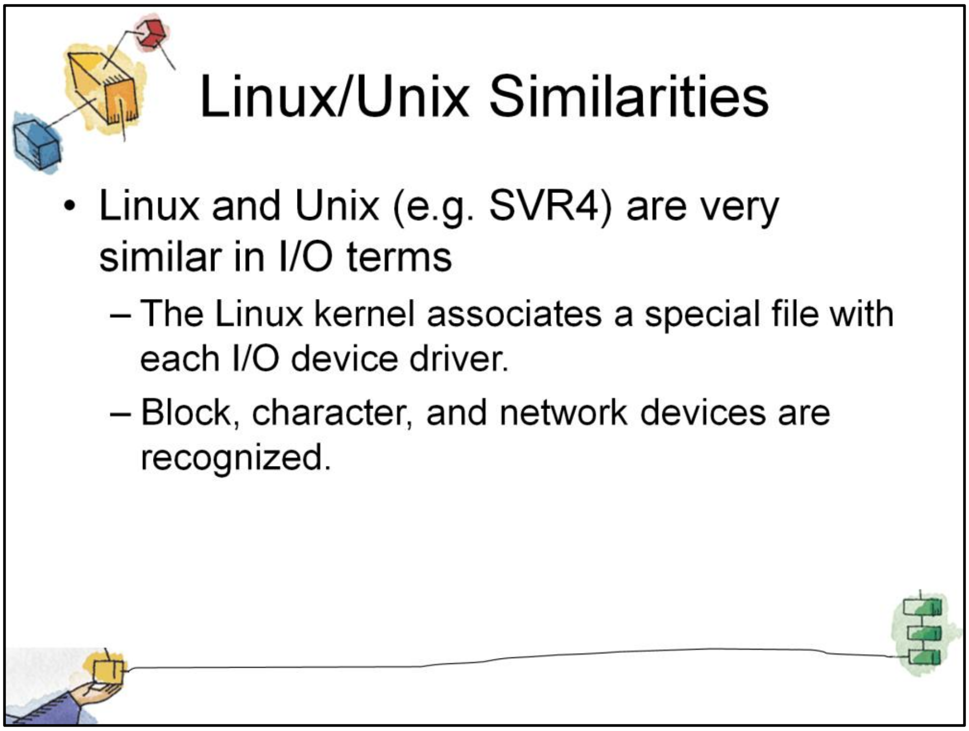
Similarly, communication lines require serial processing of bytes of data for input or output and are best handled by character queues.

The type of I/O used for a printer will generally depend on its speed.



Roadmap

- I/O Devices
- Organization of the I/O Function
- Operating System Design Issues
- I/O Buffering
- Disk Scheduling
- Raid
- Disk Cache
- UNIX SVR4 I/O
- LINUX I/O
- Windows I/O



Linux/Unix Similarities

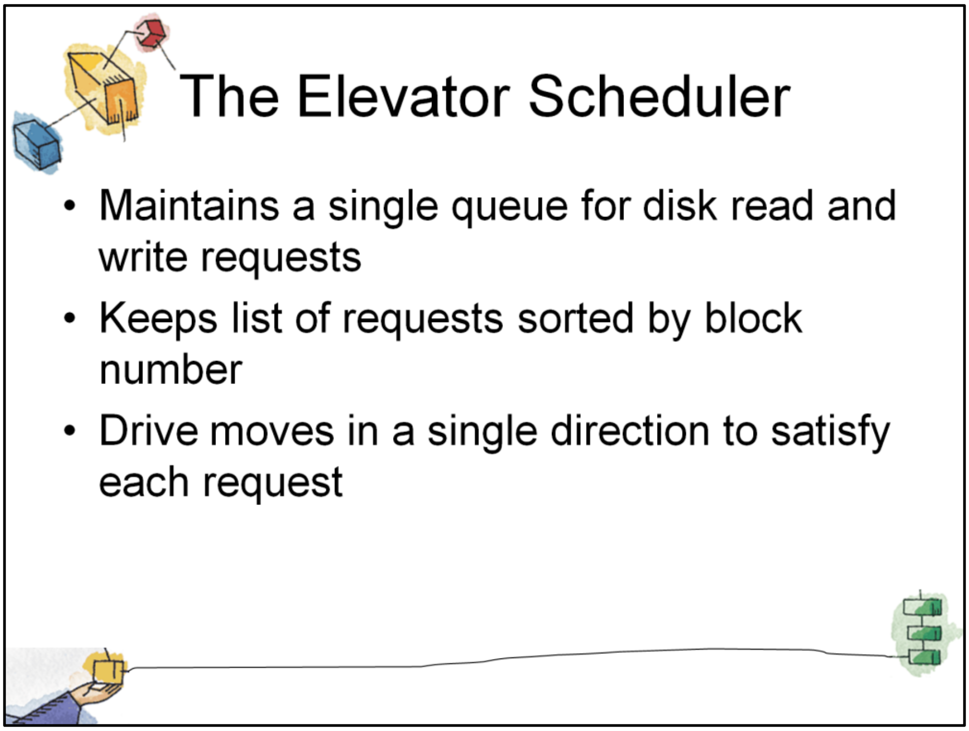
- Linux and Unix (e.g. SVR4) are very similar in I/O terms
  - – The Linux kernel associates a special file with each I/O device driver.
  - – Block, character, and network devices are recognized.

In general terms, the Linux I/O kernel facility is very similar to that of other UNIX implementation, such as SVR4.

•The Linux kernel associates a special file with each I/O device driver.

•Block, character, and network devices are recognized.

In this section, we look at several features of the Linux I/O facility.



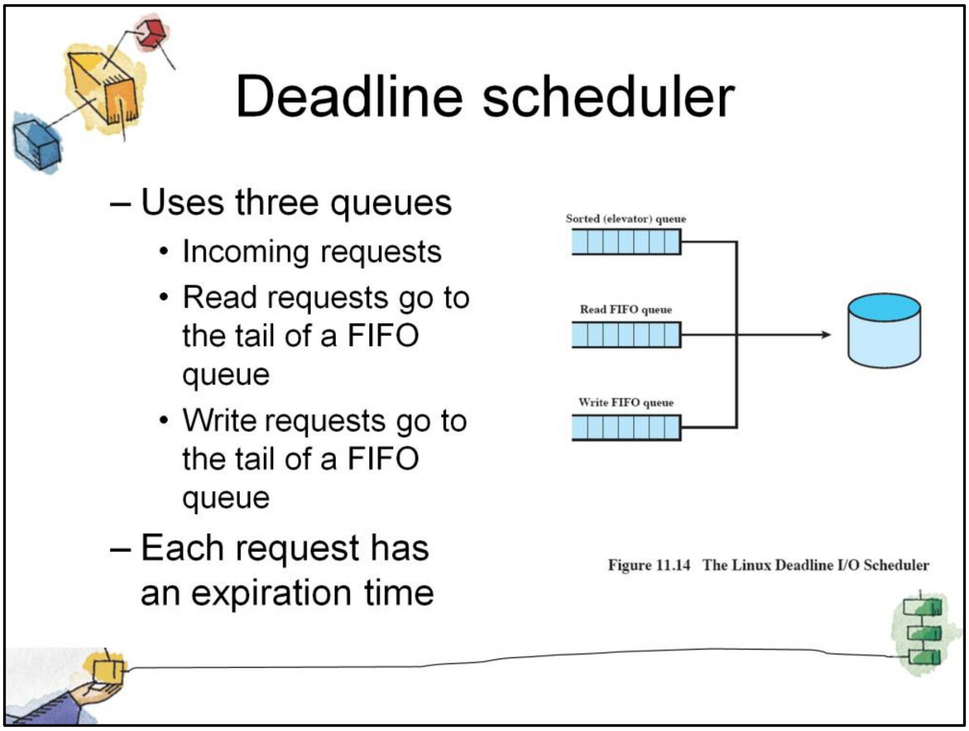
The Elevator Scheduler

- Maintains a single queue for disk read and write requests
- Keeps list of requests sorted by block number
- Drive moves in a single direction to satisfy each request

The elevator scheduler maintains a single queue for disk read and write requests and performs both sorting and merging functions on the queue.

The elevator scheduler keeps the list of requests sorted by block number.

- As the disk requests are handled, the drive moves in a single direction, satisfying each request as it is encountered.



Deadline scheduler

- Uses three queues
  - Incoming requests
  - Read requests go to the tail of a FIFO queue
  - Write requests go to the tail of a FIFO queue
- Each request has an expiration time

Figure 11.14 The Linux Deadline I/O Scheduler

Each incoming request is placed in the sorted elevator queue.

- In addition, 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.

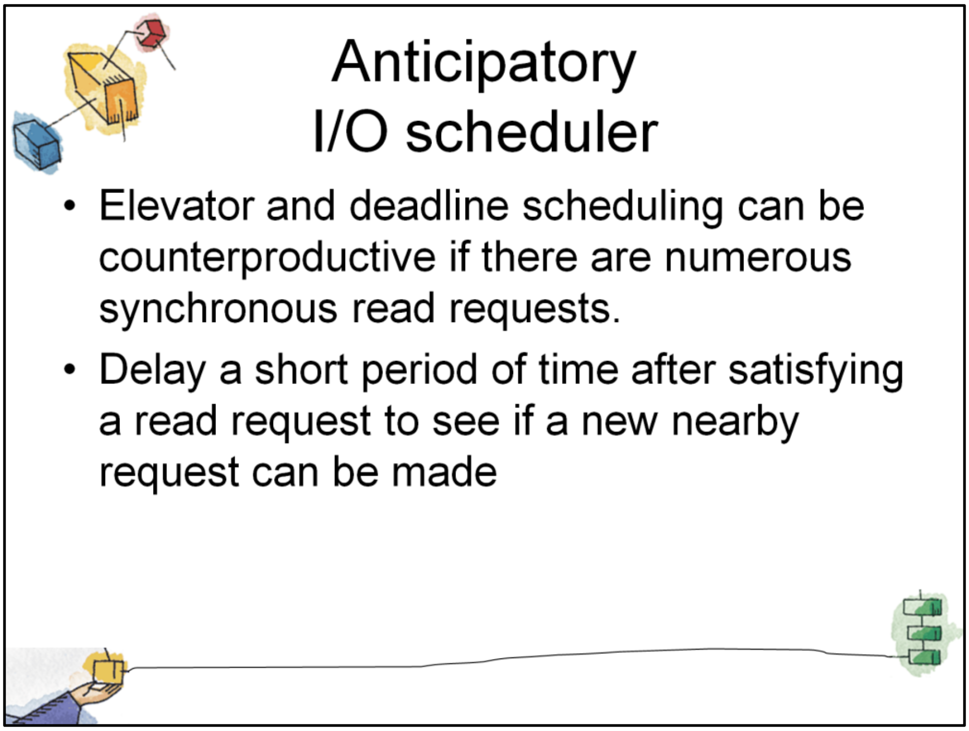
The read and write queues maintain a list of requests in the sequence in which the requests were made.

- Associated with each request is an expiration time, with a default value of 0.5 seconds for a read request and 5 seconds for a write request.

Ordinarily, the scheduler dispatches from the sorted queue.

- 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.



Anticipatory I/O scheduler

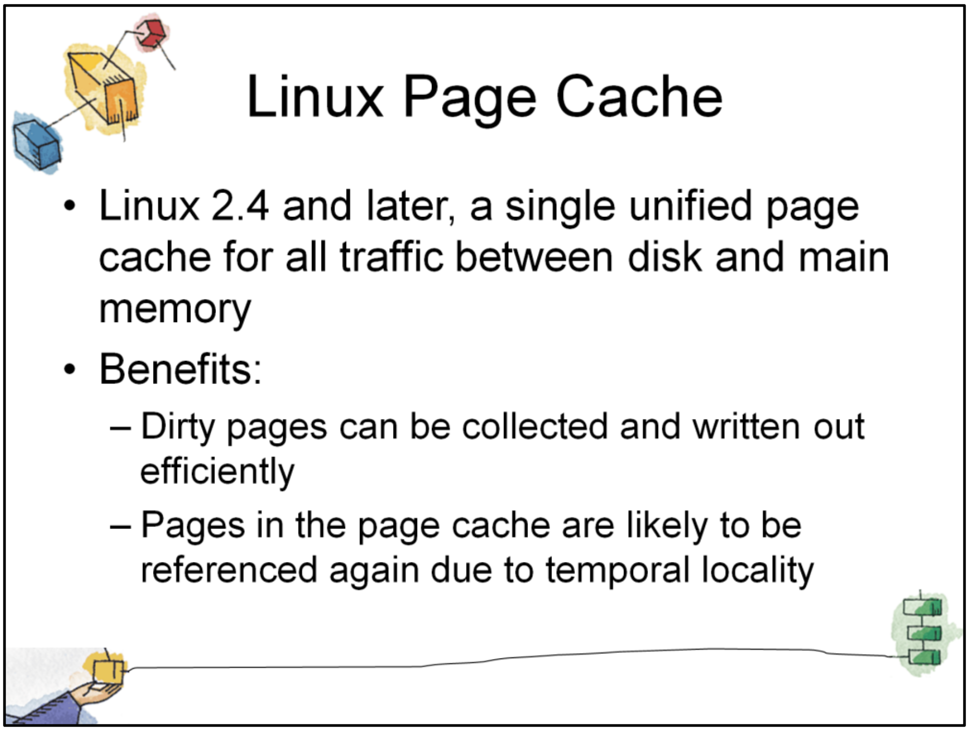
- Elevator and deadline scheduling can be counterproductive if there are numerous synchronous read requests.
- Delay a short period of time after satisfying a read request to see if a new nearby request can be made

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 for up to 6 milliseconds, depending on the configuration.

During this small delay, there is a good chance (principal of locality) that the application that issued the last read request will issue another read request to the same region of the disk.

- If so, that request will be serviced immediately.
- If no such read request occurs, the scheduler resumes using the deadline scheduling algorithm.



Linux Page Cache

- Linux 2.4 and later, a single unified page cache for all traffic between disk and main memory
- Benefits:
  - Dirty pages can be collected and written out efficiently
  - Pages in the page cache are likely to be referenced again due to temporal locality

In Linux 2.2 and earlier releases, the kernel maintained a page cache for reads and writes from regular file system files and for virtual memory pages, and a separate buffer cache for block I/O.

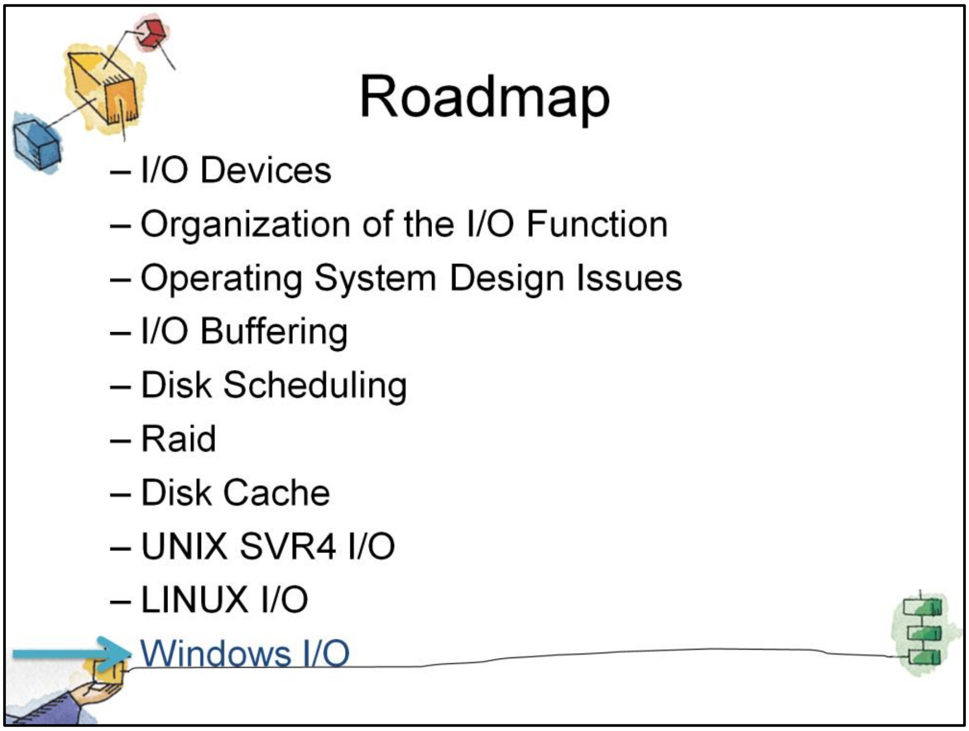
For Linux 2.4 and later, there is a single unified page cache that is involved in all traffic between disk and main memory.

The page cache confers two benefits.

- When it is time to write back dirty pages to disk, a collection of them can be ordered properly and written out efficiently.
- Second, 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.

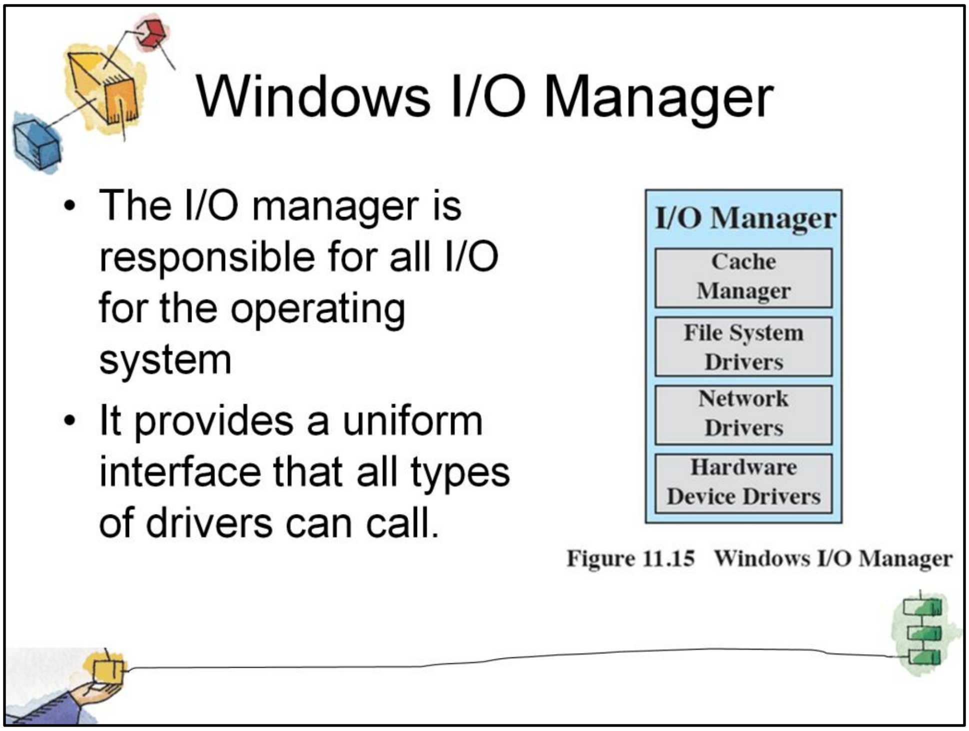
Dirty pages are written back to disk in two situations:

- When free memory falls below a specified threshold, the kernel reduces the size of the page cache to release memory to be added to the free memory pool.
- When dirty pages grow older than a specified threshold, a number of dirty pages are written back to disk.



Roadmap

- I/O Devices
- Organization of the I/O Function
- Operating System Design Issues
- I/O Buffering
- Disk Scheduling
- Raid
- Disk Cache
- UNIX SVR4 I/O
- LINUX I/O
- Windows I/O



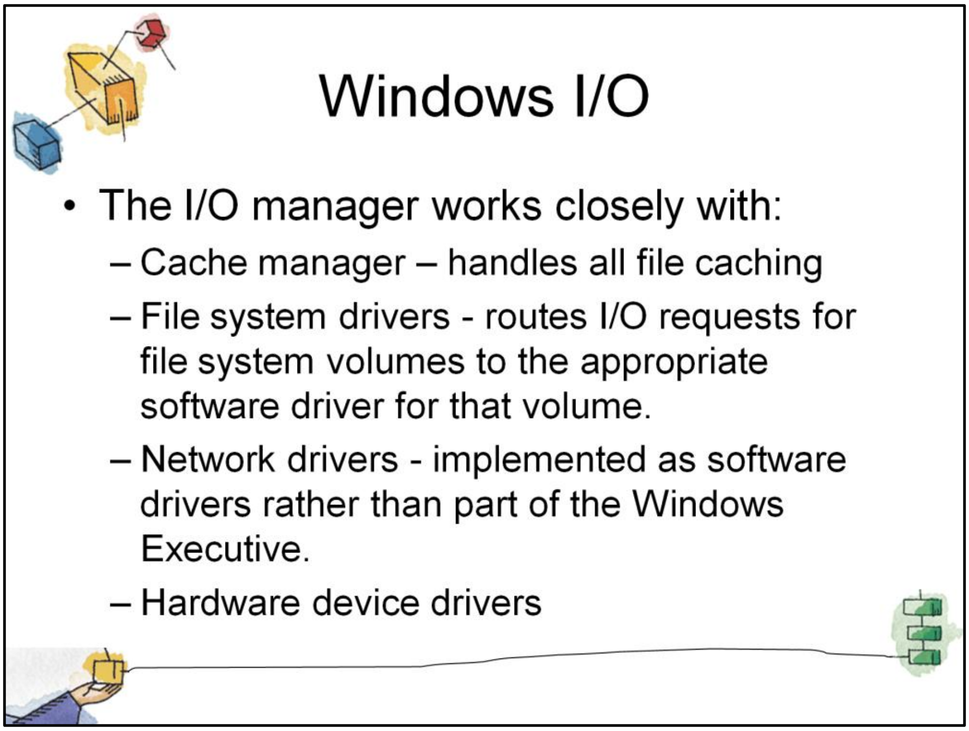
Windows I/O Manager

- The I/O manager is responsible for all I/O for the operating system
- It provides a uniform interface that all types of drivers can call.

I/O Manager
Cache Manager
File System Drivers
Network Drivers
Hardware Device Drivers

Figure 11.15 Windows I/O Manager

The I/O manager is responsible for all I/O for the operating system and provides a uniform interface that all types of drivers can call.



Windows I/O

- The I/O manager works closely with:
  - Cache manager – handles all file caching
  - File system drivers - routes I/O requests for file system volumes to the appropriate software driver for that volume.
  - Network drivers - implemented as software drivers rather than part of the Windows Executive.
  - Hardware device drivers

The I/O manager works closely with four types of kernel components:

Cache manager:

- The cache manager handles file caching for all file systems.
- A kernel thread, the lazy writer, periodically batches the updates together to write to disk which allows the I/O to be more efficient.
- The cache manager works by mapping regions of files into kernel virtual memory and then relying on the virtual memory manager to do most of the work to copy pages to and from the files on disk.

File system drivers:

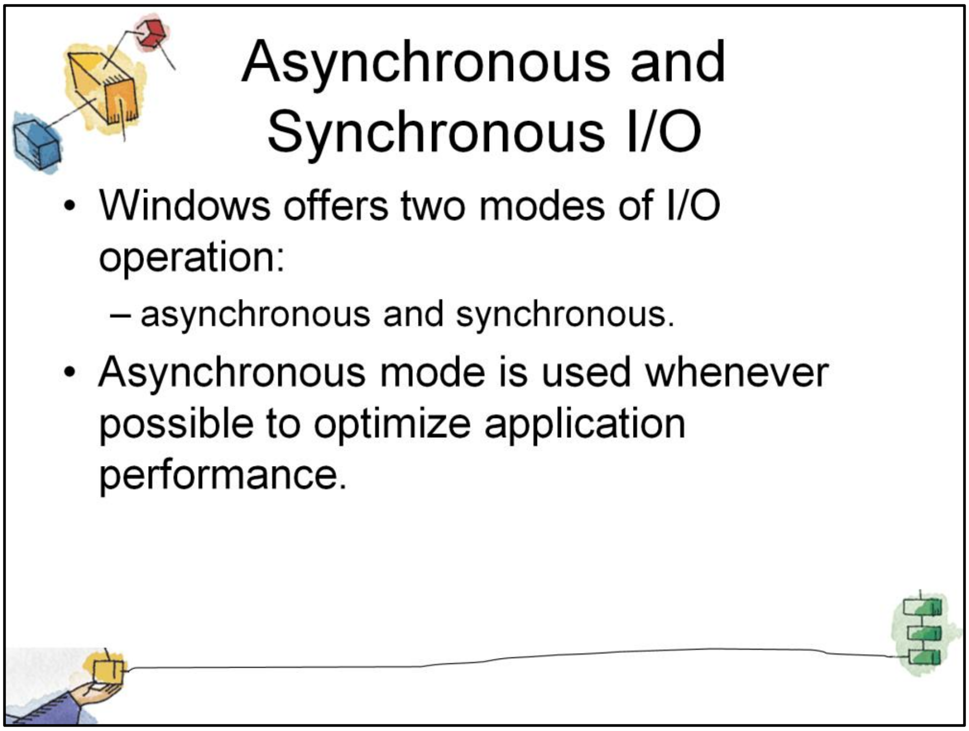
- The I/O manager treats a file system driver as just another device driver and routes I/O requests for file system volumes to the appropriate software driver for that volume.
- The file system, in turn, sends I/O requests to the software drivers that manage the hardware device adapter.

Network drivers:

- Windows includes integrated networking capabilities and support for remote file systems.
- The facilities are implemented as software drivers rather than part of the Windows Executive.

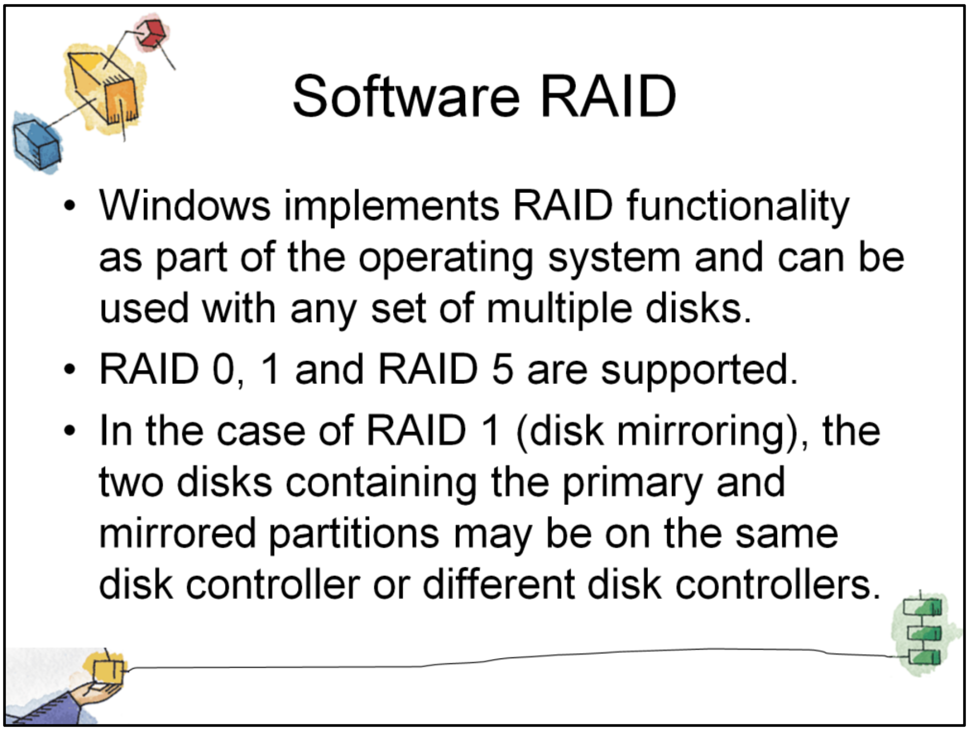
Hardware device drivers:

- These software drivers access the hardware registers of the peripheral devices using entry points in the kernel’s Hardware Abstraction Layer.
- A set of these routines exists for every platform that Windows supports; because the routine names are the same for all platforms, the source code of Windows device drivers is portable across different processor types.



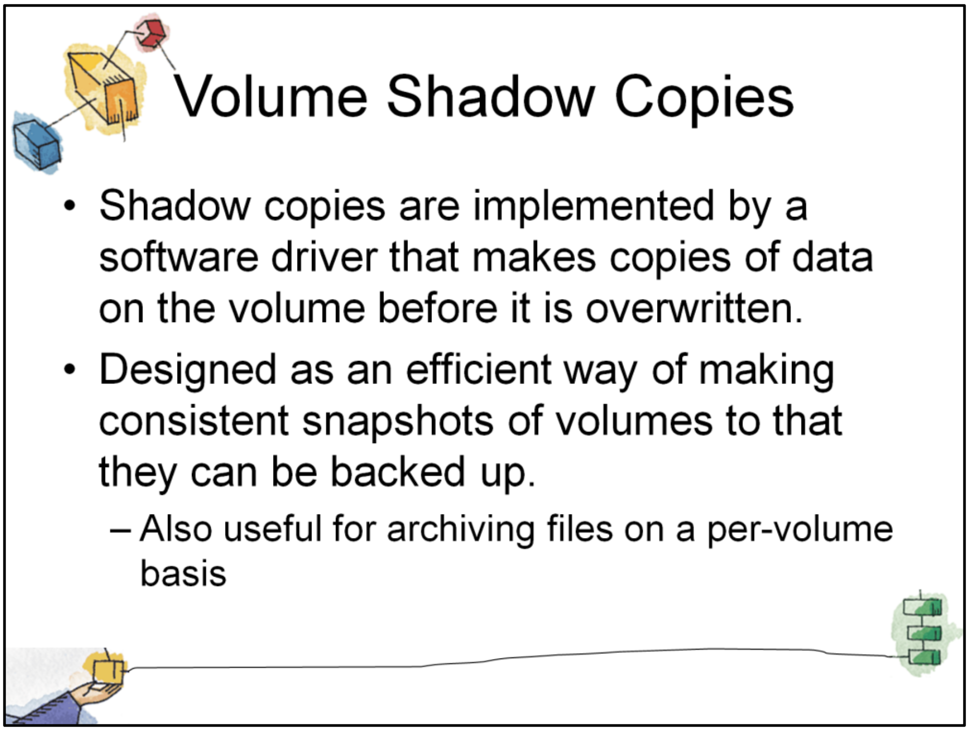
Asynchronous and Synchronous I/O

- Windows offers two modes of I/O operation:
  - asynchronous and synchronous.
- Asynchronous mode is used whenever possible to optimize application performance.



Software RAID

- Windows implements RAID functionality as part of the operating system and can be used with any set of multiple disks.
- RAID 0, 1 and RAID 5 are supported.
- In the case of RAID 1 (disk mirroring), the two disks containing the primary and mirrored partitions may be on the same disk controller or different disk controllers.



Volume Shadow Copies

- Shadow copies are implemented by a software driver that makes copies of data on the volume before it is overwritten.
- Designed as an efficient way of making consistent snapshots of volumes to that they can be backed up.
  - Also useful for archiving files on a per-volume basis

Shadow copies are an efficient way of making consistent snapshots of volumes to that they can be backed up.

They are also useful for archiving files on a per-volume basis.

- If a user deletes a file, he or she can retrieve an earlier copy from any available shadow copy made by the system administrator.

Shadow copies are implemented by a software driver that makes copies of data on the volume before it is overwritten.