Slide 1

Part 10: I/O System & Disk

- I/O Hardware
- I/O System Design and Implementation
  - Design Objectives
  - Kernel I/O Structure
  - Kernel I/O Subsystem
  - Performance Consideration
- Disk
  - Disk Structure
  - Disk Scheduling
    - *FCFS, SSTF, SCAN, C-SCAN, C-LOOK*
  - Disk Management
  - Disk Reliability

Perhaps the messiest aspect of operating system design is input/output. Because there is such a wide variety of devices and applications of those devices, it is difficult to develop a general, consistent solution.

We begin this part of the lecture with a brief discussion of I/O devices. This topic, which generally comes within the scope of computer architecture, will set the stage for an examination of I/O from the point of view of the operating system.

The next section examines operating system design issues, including design objectives, and the way in which the I/O function can be structured.

The last section is devoted to magnetic disk I/O. In contemporary systems, this form of I/O is the most important and is key to the performance as perceived by the user.

Slide 2

I/O Hardware

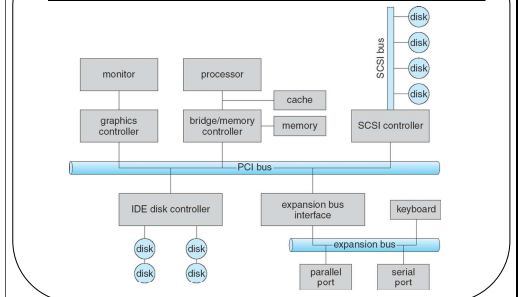


Figure shows a typical PC bus structure.

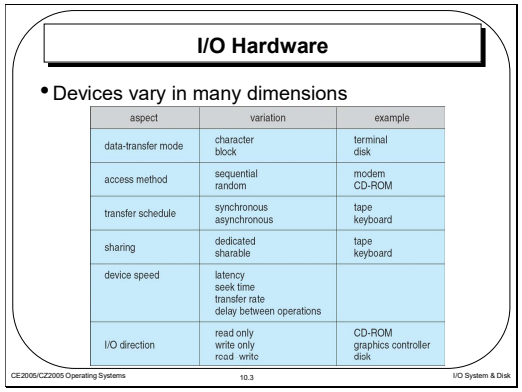
PCI Bus (The common PC system bus) connects processor-memory subsystem to fast devices. Expansion bus connects relatively slow devices. Controller operates device and is often implemented as a host adaptor. Processor communicates with controller by issuing special I/O instructions on controller's I/O port address.

(For memory mapped I/O, controller's registers are mapped into the address space of processor. The processor executes I/O requests by directly read/write the memory-mapped region.)

Although the hardware aspects of I/O are complex when considered at the level of detail of electronics-hardware design, the concepts that we have just described are sufficient to enable us to understand many I/O features of operating systems.

For more information about I/O hardware, you can look at Section 12.2 in the textbook and refer to your lecture notes of CE/CZ1006 Computer Organization and Architecture.

Slide 3



Devices vary on many dimensions, as illustrated in this table:

Character-stream or block. A character-stream device transfers bytes one by one, whereas a block device transfers a block of bytes as a unit.

Sequential or random access. A sequential device transfers data in a fixed order determined by the device, whereas the user of a random-access device can instruct the device to seek to any of the available data storage locations.

Synchronous or asynchronous. A synchronous device performs data transfers with predictable response times, in coordination with other aspects of the system. An asynchronous device exhibits irregular or unpredictable response times not coordinated with other computer events.

Sharable or dedicated. A sharable device can be used concurrently by several processes or threads; a dedicated device cannot.

Speed of operation. Device speeds range from a few bytes per second to gigabytes per second.

Read–write, read only, write once. Some devices perform both input and output, but others support only one data transfer direction. Some allow data to be modified after write, but others can be written only once and are read-only thereafter.

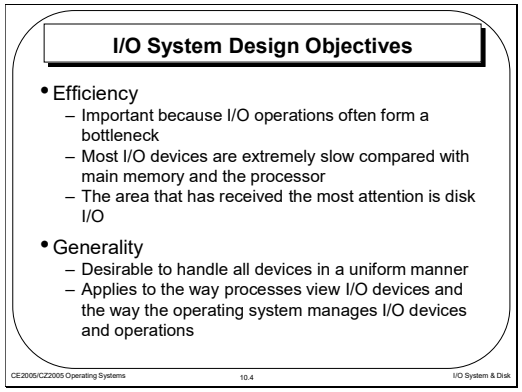
External devices that engage in I/O with computer systems can be roughly grouped into three categories:

• **Human readable: 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: Suitable for communicating with electronic equipment.** Examples are disk drives, USB keys, sensors, controllers, and actuators.

• **Communication: Suitable for communicating with remote devices. Examples** are digital line drivers and modems.

Slide 4



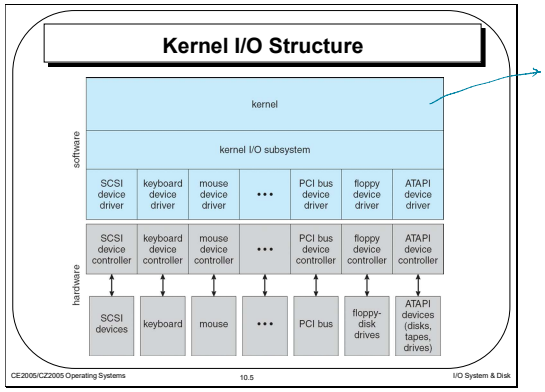
The role of the operating system in computer I/O is to manage and control I/O operations and I/O devices. There are two main objectives: efficiency and generality.

Efficiency is important because I/O operations often form a bottleneck in a computing system. As shown the last slide, we see that most I/O devices are extremely slow compared with main memory and the processor. Furthermore, CPU performance has been improved exponentially over the years; whereas the improvement on disk speed is linear. So, I/O performance is critical.

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, it will still often be the case that I/O is not keeping up with the activities of the processor. 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.

The other major objective is generality . In the interests of simplicity and freedom from error, it is desirable to handle all devices in a uniform manner. This statement 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 system. This approach 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, and unlock.

Slide 5



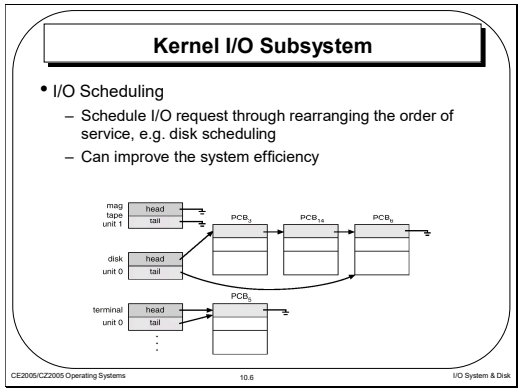
Let's look at first how to achieve generality. Like other complex software-engineering problems, the approach here involves abstraction, encapsulation, and software layering.

To achieve generality, I/O system calls encapsulate device behaviours in generic classes and provide an application I/O interface. Kernel I/O subsystem provides generic services.

Device-driver layer hides differences among I/O controllers from kernel. Add new I/O device doesn't need to change I/O interface and kernel I/O subsystem.

The device drivers present a uniform device-access interface to the kernel I/O subsystem, much as system calls provide a standard interface between the application and the operating system.

Slide 6



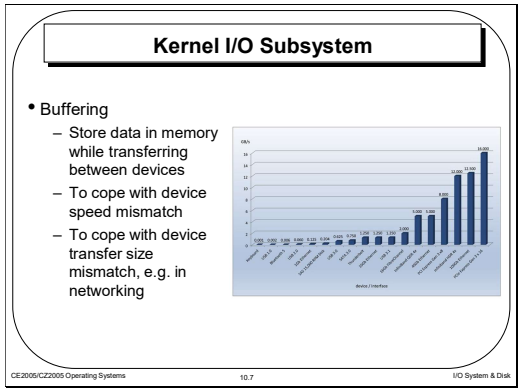
Kernel I/O subsystem provides many generic services related to I/O, e.g., scheduling, buffering, caching, spooling, device reservation, and error handling.

To schedule a set of I/O requests means to determine a good order in which to execute them. The order in which applications issue system calls rarely is the best choice. Scheduling can improve overall system performance, can share device access fairly among processes, and can reduce the average waiting time for I/O to complete.

Operating-system developers implement scheduling by maintaining a wait queue of requests for each device. When an application issues a blocking I/O system call, the request is placed on the queue for that device. The I/O scheduler rearranges the order of the queue to improve the overall system efficiency and the average response time experienced by applications. The operating system may also try to be fair, so that no one application receives especially poor service.

Several scheduling algorithms for disk I/O were detailed later.

Slide 7

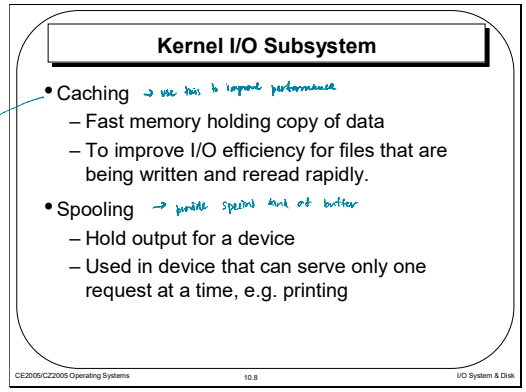


A buffer, of course, is a memory area that stores data being transferred between two devices or between a device and an application.

Buffering is used for two main reasons. One reason is to cope with a speed mismatch between the producer and consumer of a data stream. As illustrated in the figure, there are enormous differences in device speeds for typical computer hardware and interfaces. Suppose, for example, that a file is being received via Internet for storage on an SSD. The network speed may be a thousand times slower than the drive. So a buffer is created in main memory to accumulate the bytes received from the network. When an entire buffer of data has arrived, the buffer can be written to the drive in a single operation.

A second use of buffering is to provide adaptations for devices that have different data-transfer sizes. Such disparities are especially common in computer networking, where buffers are used widely for fragmentation and reassembly of messages. At the sending side, a large message is fragmented into small network packets. The packets are sent over the network, and the receiving side places them in a reassembly buffer to form an image of the source data.

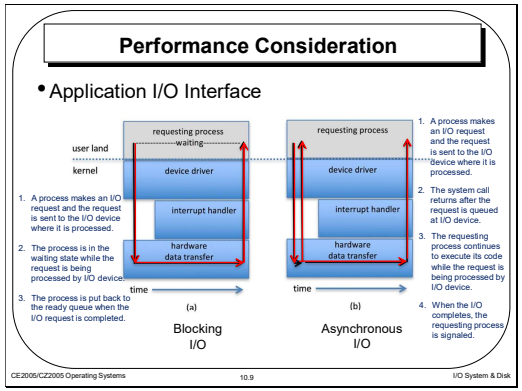
Slide 8



A cache is a region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original. For instance, the instructions of the currently running process are stored on disk, cached in physical memory, and copied again in the CPU's secondary and primary caches. Another example is the use of open file table to cache a copy of FCB in the memory. The difference between a buffer and a cache is that a buffer may hold the only existing copy of a data item; whereas a cache, by definition, holds a copy on faster storage of an item that resides elsewhere.

A spool is a buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams. Although a printer can serve only one job at a time, several applications may wish to print their output concurrently, without having their output mixed together. The operating system solves this problem by intercepting all output to the printer. Each application's output is spooled to a separate secondary storage file. When an application finishes printing, the spooling system queues the corresponding spool file for output to the printer. The spooling system copies the queued spool files to the printer one at a time. The operating system provides a control interface that enables users and system administrators to display the queue, remove unwanted jobs before those jobs print, and so on.

Slide 9

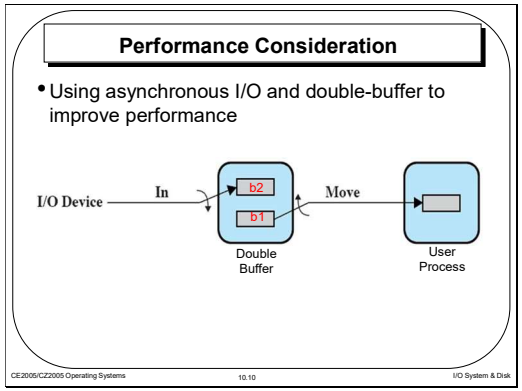


Another aspect of the system-call interface relates to the choice between blocking I/O and asynchronous I/O. When an application issues a blocking system call, the execution of the calling process is suspended. The process changes state from running to waiting and is moved to a device queue. The process is in the waiting state while the request is being processed by I/O device. After the system call completes, the process changes state from waiting to ready and is moved back to the ready queue. When it resumes execution, it will receive the values returned by the system call. The physical actions performed by I/O devices are generally asynchronous—they take a varying or unpredictable amount of time. Operating systems provide blocking system calls for the application interface, because blocking application code is easier to write.

An asynchronous call returns immediately, without waiting for the I/O to complete. The process continues to execute its code. The completion of the I/O at some future time is communicated to the process, either through the setting of some variable in the address space of the process or through the triggering of a signal or a call-back routine.

Generally, blocking I/O is appropriate when the process will only be waiting for one specific event. Examples include a disk, tape, or keyboard read by an application program. Asynchronous I/O is useful when I/O may come from more than one source and the order of the I/O arrival is not predetermined. Examples include network daemons listening to more than one network sockets, window manager that accepts mouse movement as well as keyboard input, and I/O-management programs, such as a copy command that copies data between I/O devices. In the last case, the program could optimize its performance by buffering the input and output and using asynchronous I/O to keep both devices fully occupied.

Slide 10



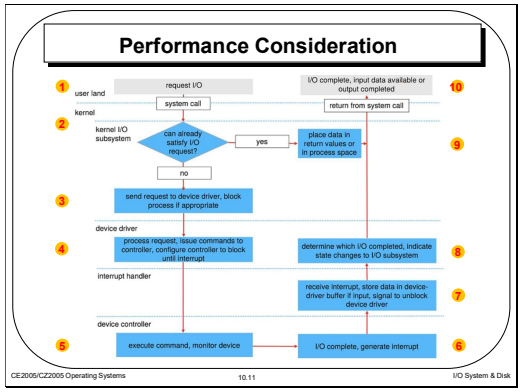
Asynchronous I/O and double-buffer can be used together to overlap I/O operation with process processing to improve performance.

Assuming a user process reads data from an I/O device for processing using asynchronous I/O. After the I/O device fills buffer 1, the process is signaled. The I/O device then starts to fill buffer 2 while data in buffer 1 is being processed. By the time the I/O device has filled buffer 2, the process should have completed processing data in buffer 1, so the I/O device can switch back to buffer 1 while the process gets data from buffer 2. ∴ can overlap

Another example is a video application that reads frames from a file on disk to one buffer while simultaneously decompressing and displaying the frames in another buffer.

The idea can be further extended to multiple (circular) buffers.

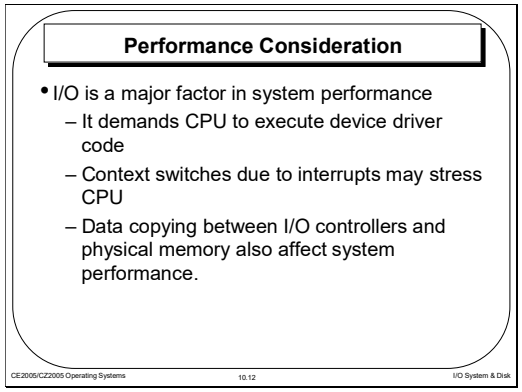
Slide 11

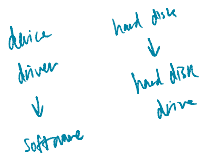


Now, let's look at how an blocking I/O request is transformed to hardware operation by using an example:

- A process issues a blocking read() system call to a file descriptor of a file that has been opened previously.
- The system-call code in the kernel checks the parameters for correctness. If the data are already available in the buffer cache, the kernel transfers the data to the user process, and the I/O request is completed.
- Otherwise, a physical I/O must be performed. The process changes to waiting state and is placed on the waiting queue for the device, and the I/O request is scheduled. The I/O subsystem sends the request to the device driver.
- The device driver allocates kernel buffer space to receive the data and schedules the I/O. It sends commands to the device controller by writing into the device-control registers.
- The device controller operates the device hardware to perform the data transfer.
- Assuming the transfer is done by DMA, the device controller generates an interrupt when the transfer completes.
- The correct interrupt handler receives the interrupt via the interrupt-vector table, stores any necessary data, signals the device driver, and returns from the interrupt.
- The device driver receives the signal, determines which I/O request has completed and the request's status, and signals the kernel I/O subsystem that the request has been completed.
- The kernel transfers data to the user process and moves the process from the wait queue back to the ready queue.
- Moving the process to the ready queue unblocks the process. When the scheduler assigns the process to the CPU, the process resumes execution at the completion of the system call.

Slide 12



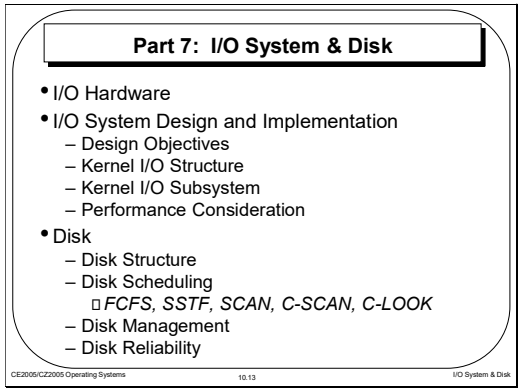


As you can seen from the last slide, I/O incurs a lot of system overheads. I/O is a major factor in system performance. It places heavy demands on the CPU to execute device-driver code and to schedule processes fairly and efficiently as they block and unblock. The resulting context switches stress the CPU and its hardware caches. I/O requires interrupt handling. Although modern computers can handle many thousands of interrupts per second, interrupt handling is still a relatively expensive task. In addition, data copying between controllers and physical memory and between kernel buffers and application data space also affect system performance.

We can employ several principles to improve the efficiency of I/O:
- Reduce the number of context switches.
- Reduce the number of times that data must be copied in memory while passing between device and application.
- Reduce the frequency of interrupts by using large transfers, smart controllers, and polling (if busy waiting can be minimized).
- Increase concurrency by using DMA-knowledgeable controllers or channels to offload simple data copying from the CPU.
- Move processing primitives into hardware, to allow their operation in device controllers to be concurrent with CPU and bus operation.
- Balance CPU, memory subsystem, bus, and I/O performance, because an overload in any one area will cause idleness in others.

Given the performance consideration of I/O, hence we should reduce number of I/Os if possible and also manage I/O operations carefully.

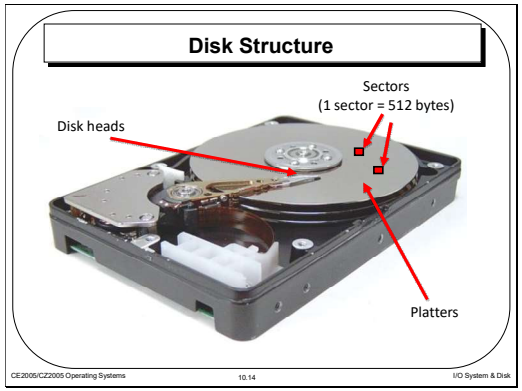
Slide 13



In this part of lecture, we discuss how mass storage—the nonvolatile storage system of a computer—is structured. The main mass-storage system in modern computers is secondary storage, which is usually provided by hard disk drives (HDD).

We first describe their physical structure. We then consider scheduling algorithms, which schedule the order of I/O requests to maximize performance. Next, we discuss disk management. Finally, we examine the structure of RAID systems.

Slide 14

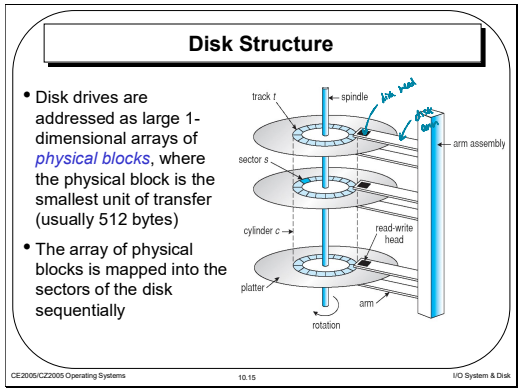


A hard disk consists of multiple platters and disk heads. Each disk platter has a flat circular shape, like a CD. Common platter diameters range from 1.8 to 3.5 inches. The two surfaces of a platter are covered with a magnetic material. We store information by recording it magnetically on the platters, and we read information by detecting the magnetic pattern on the platters.

A read–write head “flies” just above each surface of every platter. The heads are attached to a disk arm that moves all the heads as a unit.

There are two classes of disk (IDE/SATA and SCSI).

Slide 15



The diagram shows conceptually how a hard disk looks like.

A **disk** is a stack of 1 or more **platters**
A **platter** can have two active **surfaces**
Each **surface** is divided into concentric **tracks** (imagine rings)

A **track** is divided into **sectors**
A **sector** (512 bytes) is a unit of disk read/write

A **cylinder** is a combination of tracks that are lined-up together

A **disk head** (or disk arm) is used to read sectors

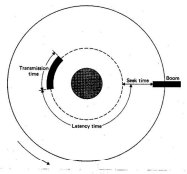
Storage devices are typically addressed as large one-dimensional arrays of physical blocks, where the physical block is the smallest unit of transfer. (Recall what we covered under file allocation methods, where we view a hard disk as containing an array of physical blocks.) The array of physical blocks is mapped into the sectors of the disk sequentially. Each physical block may map to one or multiple sectors. For example, sector 0 could be the first sector of the first track on the outermost cylinder on an HDD. The mapping proceeds in order through that track, then through the rest of the tracks on that cylinder, and then through the rest of the cylinders, from outermost to innermost.

By using this mapping on an HDD, we can—at least in theory—convert a physical block number into an old-style disk address that consists of a cylinder number, a track number within that cylinder, and a sector number within that track.

Slide 16

Disk I/O

- To read a requested sector, disk drive needs to position the disk head and perform data transfer.
- Position time has two major components
  - *Seek time* for the disk to move the heads to the track containing the desired sector (5 -25 milliseconds)
  - *Rotational latency* for the disk to rotate the desired sector to the disk head (8 milliseconds)



To read a requested sector, disk drive needs to position the disk head and perform data transfer.

The positioning time consists of two parts: the time necessary to move the disk heads to the desired cylinder, called the seek time, and the time necessary for the desired sector to rotate to the disk head, called the rotational latency. Typically, disks have seek time and rotational latency of several milliseconds.

Platters spin constantly at high speed. Most drives rotate 60 to 250 times per second (i.e., 17 to 4 milliseconds)

Rotation speed relates to transfer rates. Typical disks can transfer tens to hundreds of megabytes of data per second.

Slide 17

Disk I/O (Cont.)

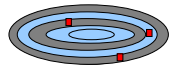
- How long to read or write N sectors?
  - Positioning Time + Data Transfer Time (N)
  - Positioning time: *Seek Time + Rotational Latency*
  - Transfer time: (*N / Number of Sectors per Track*) × *Full Rotation Time*
- Some parameters:
  - *Seek time*: 10 ms
  - *Full rotational time*: 8 ms (*Rotational latency*: 4 ms)
  - *Sector size*: 512 Bytes
  - *Track size*: 1 K sectors per track

Now let's look at an example to compare performance of random access vs sequential access on a hard disk.

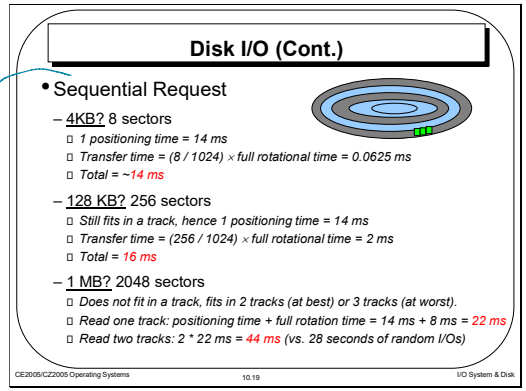
Slide 18

Disk I/O (Cont.)

- Random Request
  - Seek Time + Rotational Latency + Transfer Time
    10 ms + 4 ms + (1/1024) × 8ms = ~14 ms
- Time and bandwidth for random request of size:
  - 4 KB = *8 sectors = 8 * 14 ms = 112 ms*
  - 128 KB = *256 sectors = 256 * 14 ms = 3.5 secs*
  - 1 MB = *2048 sectors = 2048 * 14 ms = 28 seconds !!*



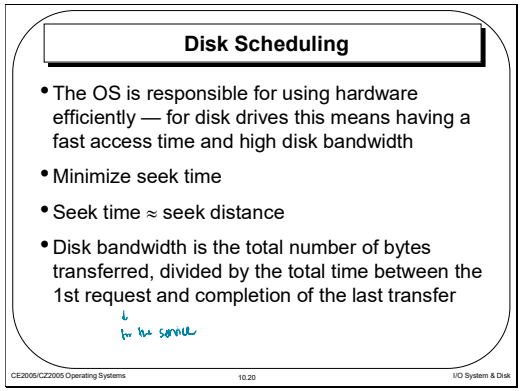
Slide 19



consecutive in hard disk

↑ for faster than random access

Slide 20



↓ for the service

As you can see from the example, servicing random requests takes much longer time than servicing sequential requests.

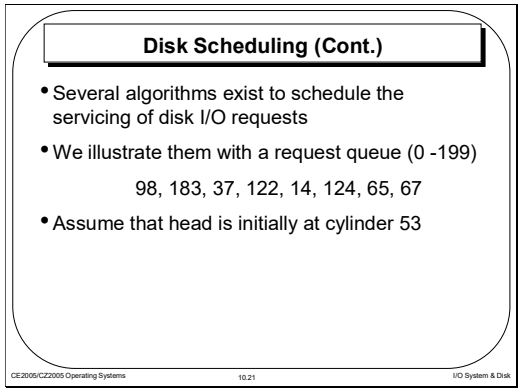
One of the responsibilities of the operating system is to use the hardware efficiently. For HDDs, meeting this responsibility means having a fast access time and high disk bandwidth.

The disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.

The goal of disk scheduling algorithm is to minimize the seek distance (i.e., the distance traveled by disk heads) when servicing multiple random requests by managing the order in which these requests are serviced. By doing so, the disk bandwidth will be improved.

minimized traveling distance of disk head

Slide 21

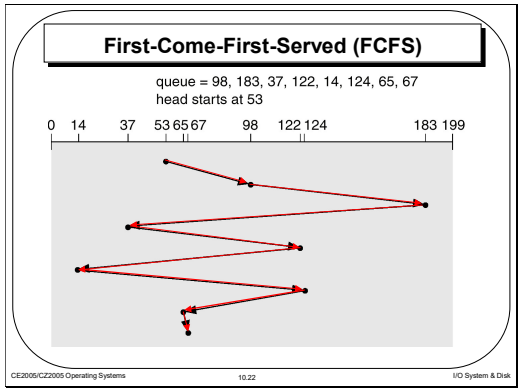


When a process issues an I/O request, if the disk drive is available, the request can be serviced immediately. If the drive is busy, any new requests for service will be placed in the queue of pending requests for that drive. For a multiprogramming system with many processes, the device queue may often have several pending requests.

The existence of a queue of requests to a device provides the opportunity for OS to optimize performance using disk scheduling algorithms.

To service a request, disk head needs to move the cylinder where the track is. Read/write starts when the required sector is rotated under the head. So, we use cylinder number to represent a request. For illustration, we assume we have the requests at the following cylinders: ….

Slide 22



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. Figure illustrates the disk arm movement with FIFO. 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. However, this technique will often approximate random scheduling in performance, if there are many processes competing for the disk. Thus, it may be profitable to consider a more sophisticated scheduling policy.

Slide 23

FCFS (Cont.)

Total head movement =

|98 - 53| + |183 - 98| + |37 - 183| + |122 - 37| +

|14 -122| + |124 -14| + |65 -124| + |67 - 65|

= 640

Problem: *FCFS may have wild swings back and forth!*

Slide 24

Shortest-Seek-Time-First (SSTF)

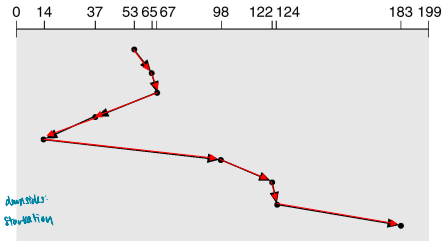
- Selects the request with the minimum seek time from the current head position
- SSTF scheduling is a form of SJF scheduling; may have starvation under heavy load, since distant requests may never be serviced
- Note: requests are from processes executing concurrently

The shortest-service-time-first (SSTF) policy is to 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. Of course, 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. A random tie-breaking algorithm may be used to resolve cases of equal distances.

Slide 25

SSTF (Cont.)

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53



Slide 26

SSTF (Cont.)

Total head movement =

|65 - 53| + |67 - 65| + |37 - 67| + |14 - 37| +

|98 -14| + |122 - 98| + |124 - 122| + |183 - 124|

= 236

Problem: *SSTF may cause starvation!*

Slide 27

SCAN

- Disk arm starts at one end of the disk, and moves toward the other end, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues
- Also called the *elevator algorithm*
- Assuming here the head is moving toward 0
- Starvation unlikely but servicing requests may be delayed

Different from FIFO, in SSTF there may always be new requests arriving that will be chosen before an existing request. A simple alternative that prevents this sort of starvation is the SCAN algorithm, also known as the elevator algorithm because it operates much the way an elevator does.

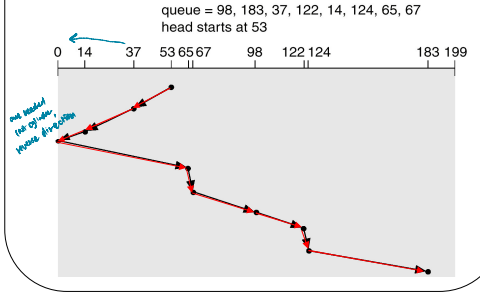
With SCAN, the arm moves in one direction, satisfying all outstanding requests en route, until it reaches the last track in that direction or until there are no more requests in that direction. This latter refinement is sometimes referred to as the LOOK policy. The service direction is then reversed and the scan proceeds in the opposite direction, again picking up all requests in order.

Note that the SCAN policy is biased against the area most recently traversed. Thus it does not exploit locality as well as SSTF.

Slide 28

SCAN (Cont.)

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53



Slide 29

SCAN (Cont.)

Total head movement =

|37 - 53| + |14 - 37| + |0 - 14| + |65 - 0| + |67 - 65|

+ |98 - 67| + |122 - 98| + |124 - 122| + |183 - 124|

= 236

After the head reverses direction, does it needs to serve requests in the reversed direction immediately?

Slide 30

Circular-Scan (C-SCAN)

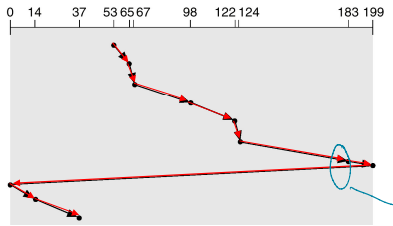
- Head moves from low end to high end of the disk, servicing requests along
- When reaching the high end, head immediately returns to the beginning of the disk, without servicing any requests

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. With SCAN, if the expected time for a scan from inner track to outer track is *t* , then the expected service interval for sectors at the periphery is 2 *t*. With C-SCAN, the interval is on the order of *t* + *max_s*, where *max_s* is the maximum seek time.

Slide 31

C-SCAN (Cont.)

queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53



moves to last cylinder instead

changes here instead

Slide 32

C-SCAN (Cont.)

Total head movement =

|65 - 53| + |67 - 65| + |98 - 67| + |122 - 98| +

|124 - 122| + |183 -124| + |199 - 183| +

|0 -199| + |14 - 0| + |37 -14|

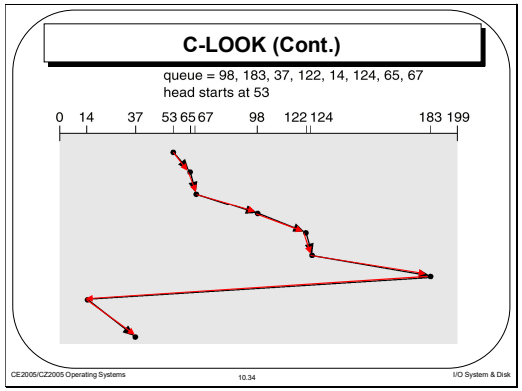
= 382

Slide 33

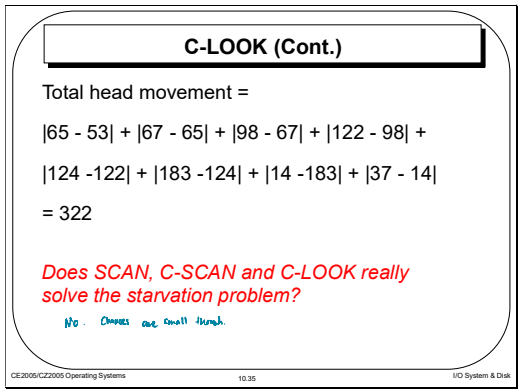
C-LOOK

- Version of C-SCAN
- Head moves from low end to high end of the disk, servicing requests along
- When reaching the last request, head returns immediately to the lowest cylinder # request
- From there it services the requests towards the high end of the disk

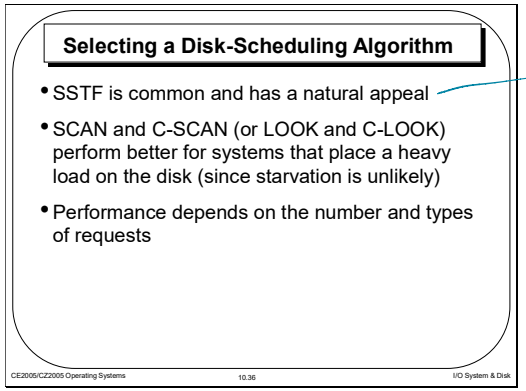
Slide 34



Slide 35



Slide 36



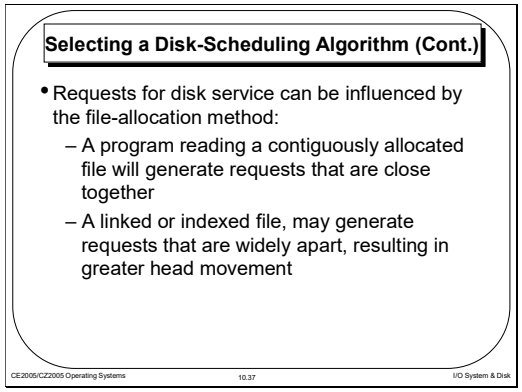
There are many disk-scheduling algorithms not included in this coverage, because they are rarely used. But how do operating system designers decide which to implement? For any particular list of requests, we can define an optimal order to service the requests, but the computation needed to find an optimal schedule may not justify the saving of disk head movement.

With any scheduling algorithm, however, performance depends heavily on the number and types of requests. For instance, suppose that the queue usually has just one outstanding request. Then, all scheduling algorithms behave the same, because they have only one choice of where to move the disk head: they all behave like FCFS scheduling.

SCAN and C-SCAN perform better for systems that place a heavy load on the disk, because they are less likely to cause a starvation problem. There can still be starvation though, which drove Linux to create the deadline scheduler. If you are interested, you can look for more information about the deadline scheduler in the textbook.

Slide 37

random access take longer than sequential

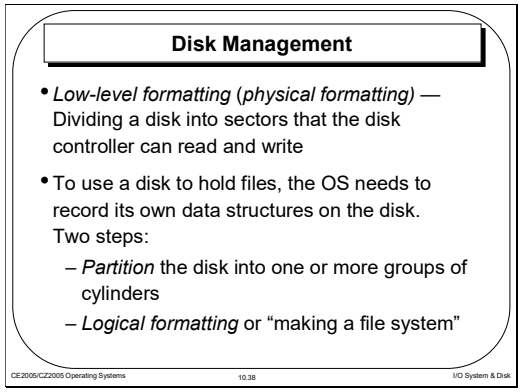


Requests for disk service can be influenced by the file allocation method too.

Contiguous allocation will allocate contiguous physical blocks to a file; whereas physical blocks allocated to a file can be anywhere on the hard disk if linked or indexed allocation is used.

Hence, to read a file sequentially, linked or indexed allocation will result in greater disk head movement than contiguous allocation.

Slide 38



The operating system is responsible for several other aspects of storage device management, too. Here, we discuss drive initialization,

A new storage device is a blank slate: it is just a platter of a magnetic recording material or a set of uninitialized semiconductor storage cells. Before a storage device can store data, it must be divided into sectors that the controller can read and write. This process is called low-level formatting, or physical formatting. Low-level formatting fills the device with a special data structure for each storage location. The data structure for a sector typically consists of a header, a data area, and a trailer. The header and trailer contain information used by the controller, such as a sector number and an error detection or correction code.

Formatting a disk with a larger sector size means that fewer sectors can fit on each track, but it also means that fewer headers and trailers are written on each track and more space is available for user data. Some operating systems can handle only one specific sector size.

Most drives are low-level-formatted at the factory as a part of the manufacturing process.

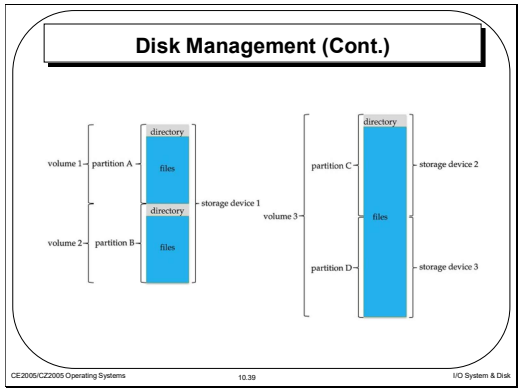
Before it can use a drive to hold files, the operating system still needs to record its own data structures on the device. It does so in the following steps.

The first step is to partition the device into one or more groups of cylinders. The operating system can treat each partition as though it were a separate device. For instance, one partition can hold a file system containing a copy of the operating system's executable code, another the swap space, and another a file system containing the user files. The partition information is written in a fixed format at a fixed location on the storage device.

A partition is a "raw" if it does not contain a file system. For example, the swap space can be a raw partition.

The next step is logical formatting, or creation of a file system. In this step, the operating system stores the initial file-system data structures onto the device. These data structures may include maps of free and allocated space and an initial empty directory.

Slide 39



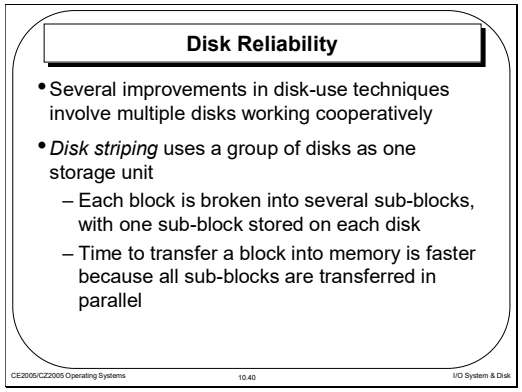
This is how logically a storage device looks like after partitioning and logical formatting.

A volume is a logical concept and is used to refer to a storage area associated with a single file system, which typically residents on a single partition of a hard disk. But, as we see next a volume may also contain multiple partitions cross different storage devices.

Each volume has a volume control block. In UFS (Unix File System), this is called a superblock, which contains volume details, such as the number of blocks in the volume, the size of the blocks, a free-block count and free-block pointers, and a free-FCB count and FCB pointers.

A directory structure is used to organize the files. In UFS, as we have already introduced, each entry in the directory structure contains file name and associated inode number.

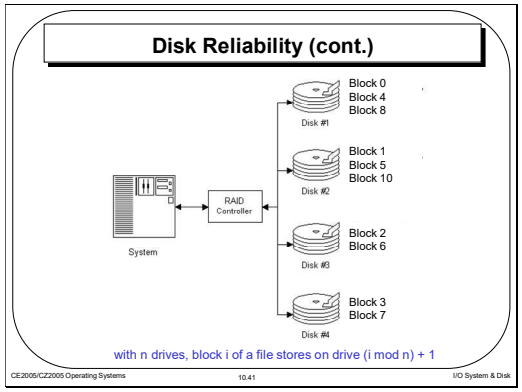
Slide 40



Storage devices have continued to get smaller and cheaper, so it is now economically feasible to attach many drives to a computer system. Having a large number of drives in a system presents opportunities for improving the rate at which data can be read or written, if the drives are operated in parallel. Furthermore, this setup offers the potential for improving the reliability of data storage, because redundant information can be stored on multiple drives. Thus, failure of one drive does not lead to loss of data. A variety of disk-organization techniques, collectively called redundant arrays of independent disks (RAIDs), are commonly used to address the performance and reliability issues.

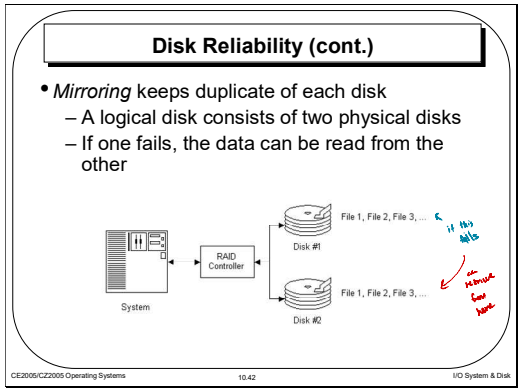
Now let's consider how parallel access to multiple drives improves performance. With multiple drives, we can improve the transfer rate by striping data across the drives. Each block is broken into several sub-blocks, with one sub-block stored on each disk. In this case, time to transfer a block into memory will be much faster since all sub-blocks can be transferred in parallel from different disks.

Slide 41



This slides shows an example of block-level striping. Blocks of a file are stored across multiple drives in round-robin manner; with n drives, block i of a file stores on drive (i mod n) + 1.

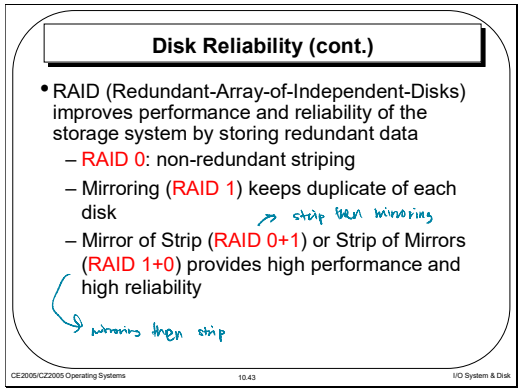
Slide 42



The solution to the problem of reliability is to introduce redundancy; we store extra information that is not normally needed but can be used in the event of disk failure to rebuild the lost information. Thus, even if a disk fails, data are not lost.

The simplest (but most expensive) approach to introducing redundancy is to duplicate every drive. This technique is called mirroring. With mirroring, a logical disk consists of two physical drives, and every write is carried out on both drives. The result is called a mirrored volume. If one of the drives in the volume fails, the data can be read from the other. Data will be lost only if the second drive fails before the first failed drive is replaced.

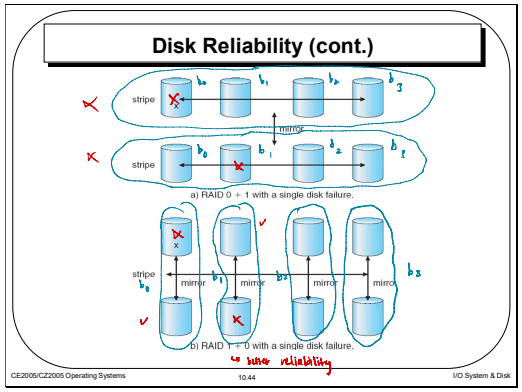
Slide 43



Mirroring provides high reliability, but it is expensive. Striping provides high data-transfer rates, but it does not improve reliability. Numerous schemes to provide redundancy at lower cost have been proposed. These schemes have different cost–performance trade-offs and are classified according to levels called RAID levels. The RAID scheme consists of seven levels, zero through six. Non-redundant striping and mirroring are RAID level 0 and 1 respectively. You can look at the textbook for more details about other RAID levels.

It is common in environments where both performance and reliability are important. So, RAID levels 0 and 1 can be used in combination. RAID 0 provides the performance, while RAID 1 provides the reliability. RAID 0+1 is called mirror of strip and RAID 1+0 is called strip of mirrors.

Slide 44



In RAID 0 + 1, a file is striped in one set of drives first, and then the stripe is mirrored to another, equivalent stripe on a mirrored set of drives.

In RAID level 1 + 0, drives are mirrored in pairs first and then a file is striped on the the resulting mirrored pairs of drives.

RAID 1 + 0 has some theoretical advantages over RAID 0 + 1. For example, if a single drive fails in RAID 0 + 1, an entire stripe is inaccessible, leaving only the other stripe. With a failure in RAID 1 + 0, a single drive is unavailable, but the drive that mirrors it is still available, as are all the rest of the drives.

Slide 45

Summary

- Device-driver layer hides the differences among device controllers from the I/O subsystem of the kernel.
- The I/O subsystem provides functions for: I/O scheduling, buffering, spooling, error handling, device reservation, and device name translation.
- I/O system calls are costly in terms of CPU consumption, because of the many layers of software between a physical device and the application.

Any methods out to improve performance ??

Slide 46

Summary

- Disk scheduling algorithms optimize seek time:
  - FCFS, SSTF, SCAN, C-SCAN, C-LOOK
- Disk management requires disk partitioning and formatting
- Disk striping improve performance
- Mirroring improves reliability

True or False

A process will be in waiting state after performing an I/O system call if non-blocking I/O is used. F

If most of the files are accessed in a sequential manner, index file allocation method would be most appropriate for allocating storage space for files in such a system. F

→ contiguous better

index for random access like that

Buffer holds a copy of the data that resides elsewhere. F

is cache, not buffer