UNIT- 4

Chapter -12

I/O SYSTEMS

TOPICS:

- **I/O Hardware**
- **Application I/O Interface**
- **Kernel I/O Subsystem**
- **Transforming I/O to Hardware Operations**
- **STREAMS**
- **Performance**

Overview:

- ➢ Management of I/O devices is a very important part of the operating system - so important and so varied that entire I/O subsystems are devoted to its operation.

- ➢ (Consider the range of devices on a modern computer, from mice, keyboards, disk drives, display adapters, USB devices, network connections, audio I/O, printers, special devices for the handicapped, and many special-purpose peripherals.)
  I/O Subsystems must contend with two trends:
- ➢ (1) The gravitation towards standard interfaces for a wide range of devices, making it easier to add newly developed devices to existing systems, and
- ➢ (2) the development of entirely new types of devices, for which the existing standard interfaces are not always easy to apply.
  Device drivers are modules that can be plugged into an OS to handle a particular device or category of similar devices.

1.) I/O Hardware:

- ➢ Computers operate a great many kinds of devices. Most fit into the general categories of storage devices (disks,tapes), transmission devices (network cards, modems), and human interface devices (screen, keyboard, mouse).
- ➢ A device communicates with a computer system by sending signals over a cable or even through the air.
- ➢ VGThe device communicates with the machine via a connection point(or port), for example, a serial port. If one or more devices use a common set of wires, the connection is called a **bus.**
- ➢ A **bus** is a set of wires and a rigidly defined protocol that specifies a set of messages that can be sent on the wires.

- I/O devices can be roughly categorized as storage, communications, user-interface, and other.
- Devices communicate with the computer via signals sent over wires or through the air.
  Devices connect with the computer via ports, e.g. a serial or parallel port.
  A common set of wires connecting multiple devices is termed a bus.
- Buses include rigid protocols for the types of messages that can be sent across the bus and the procedures for resolving contention issues.

- Figure below illustrates three of the four bus types commonly found in a modern PC:
- The **PCI bus** connects high-speed high-bandwidth devices to the memory subsystem. The **expansion bus** connects slower low-bandwidth devices, which typically deliver data one character at a time (with buffering).
- The **SCSI bus** connects a number of SCSI devices to a common SCSI controller.
- A **daisy-chain bus**, (not shown) is when a string of devices is connected to each other like beads on a chain, and only one of the devices is directly connected to the host.

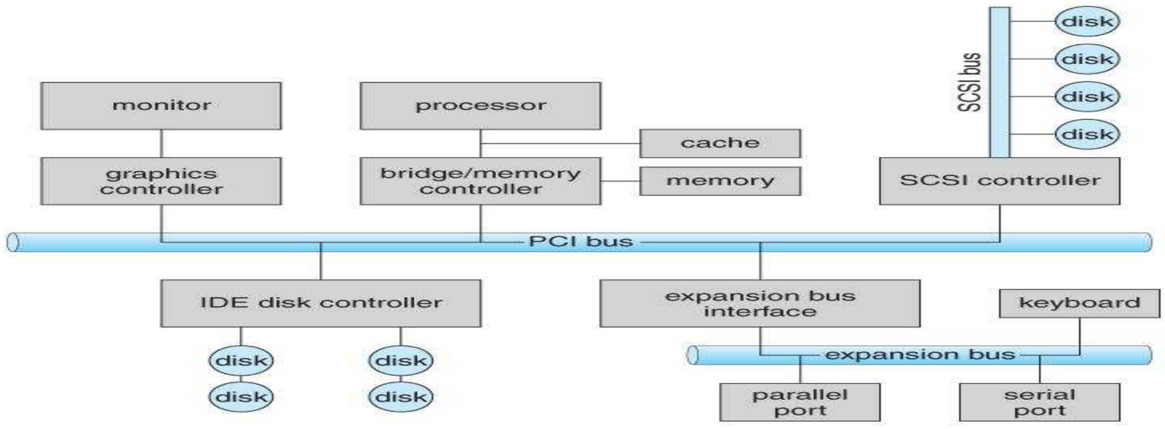
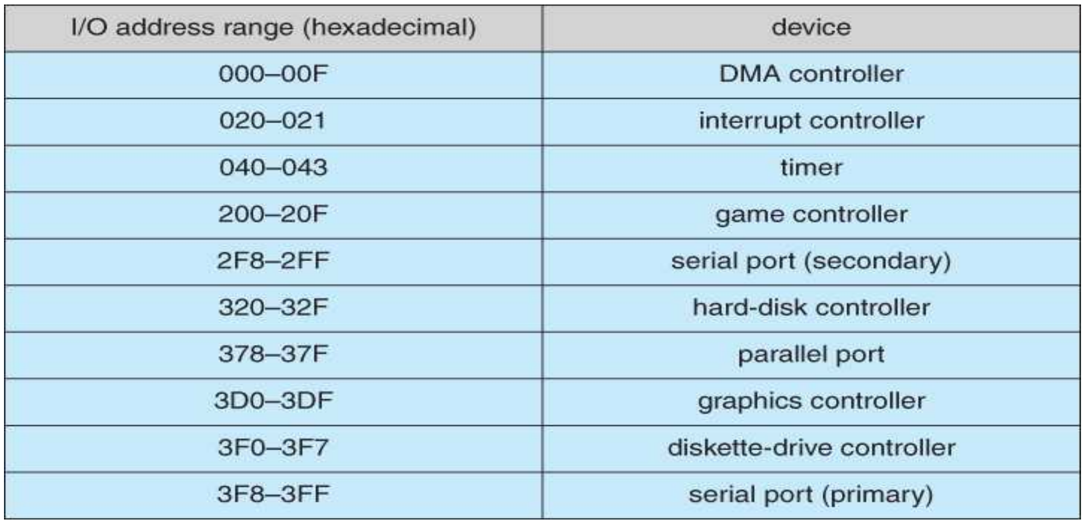


Figure - A typical PC bus structure.

One way of communicating with devices is through **registers** associated with each port. Registers may be one to four bytes in size, and may typically include (a subset of) the following four:

1. **The data-in register** is read by the host to get input from the device.
2. The data-out register is written by the host to send output.
3. The **status register** has bits read by the host to ascertain the status of the device, such as idle, ready for input, busy, error, transaction complete, etc.
4. **The control register** has bits written by the host to issue commands or to change

settings of the device such as parity checking, word length, or full- versus half-duplex operation.
Figure shows some of the most common I/O port address ranges.



I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

Figure - Device I/O port locations on PCs (partial);-

1.1 Polling:

- One simple means of device handshaking involves polling: The host repeatedly checks the busy bit on the device until it becomes clear.
- The host writes a byte of data into the data-out register, and sets the write bit in the command register (in either order).\
- The host sets the command ready bit in the command register to notify the device of the pending command.
- When the device controller sees the command-ready bit set, it first sets the busy bit.
- Then the device controller reads the command register, sees the write bit set, reads the byte of data from the data-out register, and outputs the byte of data.
- The device controller then clears the error bit in the status register, the command-ready bit, and finally clears the busy bit, signaling the completion of the operation.
- Polling can be very fast and efficient, if both the device and the controller are fast and if there is significant data to transfer.
- It becomes inefficient, however, if the host must wait a long time in the busy loop waiting for the device, or if frequent checks need to be made for data that is infrequently there.

1.2.) Interrupts:

- Interrupts allow devices to notify the CPU when they have data to transfer or when an operation is complete, allowing the CPU to perform other duties when no I/O transfers need its immediate attention.
- The CPU has an interrupt-request line that is sensed after every instruction. A device's controller raises an interrupt by asserting a signal on the interrupt request line.
- The CPU then performs a state save, and transfers control to the interrupt handler routine at a fixed address in memory. (The CPU catches the interrupt and dispatches the interrupt handler).
- The interrupt handler determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a return from interrupt instruction to return control to the CPU. (The interrupt handler clears the interrupt by servicing the device).
- (**Note** that the state restored does not need to be the same state as the one that was saved when the interrupt went off. See below for an example involving time-slicing).

  Figure illustrates the interrupt-driven I/O procedure:
- The above description is adequate for simple interrupt-driven I/O, but there are three needs in modern computing which complicate the picture:
- The need to defer interrupt handling during critical processing, the need to determine which interrupt handler to invoke, without having to poll all devices to see which one needs attention, and The need for multi-level interrupts, so the system can differentiate between high- and low-priority interrupts for proper response.
- These issues are handled in modern computer architectures with interrupt-controller hardware.

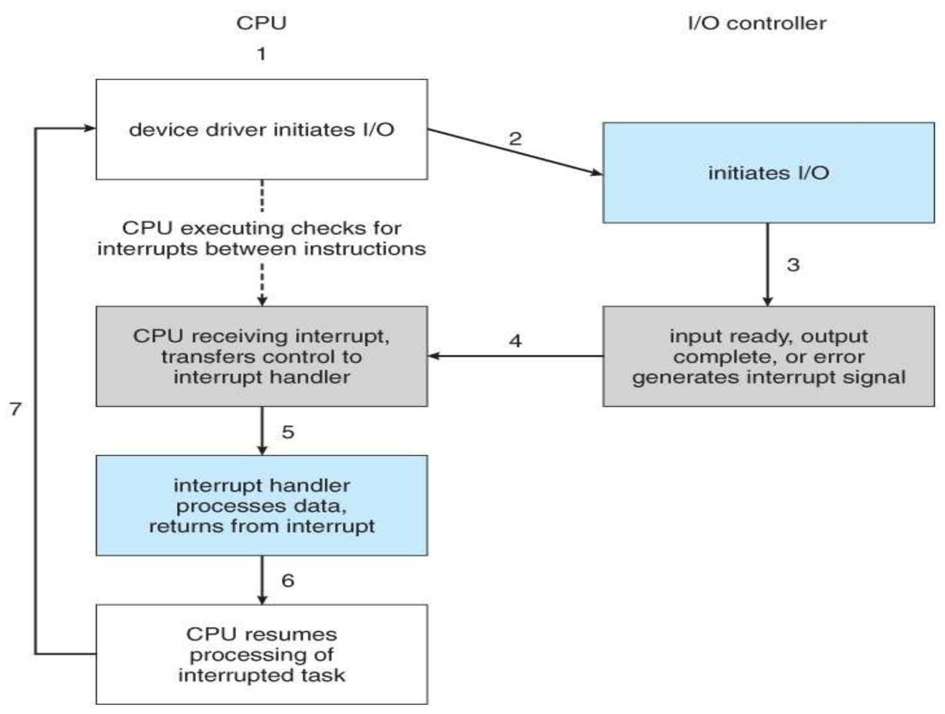
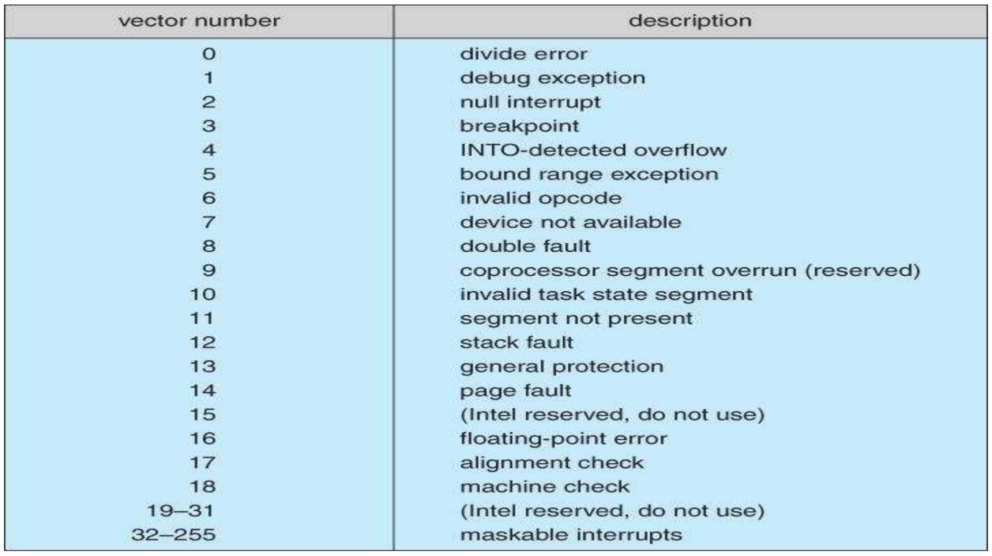

CPU
I/O controller
1
device driver initiates I/O
2
initiates I/O
CPU executing checks for interrupts between instructions
3
CPU receiving interrupt, transfers control to interrupt handler
4
input ready, output complete, or error generates interrupt signal
7
5
interrupt handler processes data, returns from interrupt
6
CPU resumes processing of interrupted task

Figure - Interrupt-driven I/O cycle.

- Most CPUs now have two interrupt-request lines: One that is non-maskable for critical error conditions and one that is maskable, that the CPU can temporarily ignore during critical processing.
- The interrupt mechanism accepts an address, which is usually one of a small set of numbers for an offset into a table called the interrupt vector.
- This table (usually located at physical address zero ?) holds the addresses of routines prepared to process specific interrupts.
- The number of possible interrupt handlers still exceeds the range of defined interrupt numbers, so multiple handlers can be interrupt chained.
- Effectively the addresses held in the interrupt vectors are the head pointers for linked-lists of interrupt handlers.
- Figure shows the Intel Pentium interrupt vector. Interrupts 0 to 31 are **nonmaskable** and reserved for serious hardware and other errors.
- **Maskable** interrupts, including normal device I/O interrupts begin at interrupt 32.

- Modern interrupt hardware also supports interrupt priority levels, allowing systems to mask off only lower-priority interrupts while servicing a high-priority interrupt, or conversely to allow a high-priority signal to interrupt the processing of a low-priority one.



vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19–31	(Intel reserved, do not use)
32–255	maskable interrupts

Fig: Intel Pentium processor event-vector table.

- At boot time the system determines which devices are present, and loads the appropriate handler addresses into the interrupt table.
- During operation, devices signal errors or the completion of commands via interrupts.
- Exceptions, such as dividing by zero, invalid memory accesses, or attempts to access kernel mode instructions can be signaled via interrupts.

- Time slicing and context switches can also be implemented using the interrupt mechanism.
- The scheduler sets a hardware timer before transferring control over to a user process.
  When the timer raises the interrupt request line, the CPU performs a state-save, and transfers control over to the proper interrupt handler, which in turn runs the scheduler.
- The scheduler does a state-restore of a different process before resetting the timer and issuing the return-from-interrupt instruction.
- A similar example involves the paging system for virtual memory - A page fault causes an interrupt, which in turn issues an I/O request and a context switch as described above, moving the interrupted process into the wait queue and selecting a different process to run.
- When the I/O request has completed (i.e. when the requested page has been loaded up into physical memory), then the device interrupts, and the interrupt handler moves the process from the wait queue into the ready queue, (or depending on scheduling algorithms and policies, may go ahead and context

switch it back onto the CPU.)
System calls are implemented via software interrupts, a.k.a. traps.

- When a (library) program needs work performed in kernel mode, it sets command information and possibly data addresses in certain registers, and then raises a software interrupt.(E.g. 21 hex in DOS.)

- The system does a state save and then calls on the proper interrupt handler to process the request in kernel mode. Software interrupts generally have low priority, as they are not as urgent as devices with limited buffering space.

- Interrupts are also used to control kernel operations, and to schedule activities for optimal performance. For example, the completion of a disk read operation involves two interrupts:
  A high-priority interrupt acknowledges the device completion, and issues the next disk request so that the hardware does not sit idle.

- A lower-priority interrupt transfers the data from the kernel memory space to the user space, and then transfers the process from the waiting queue to the ready queue.

The Solaris OS uses a multi-threaded kernel and priority threads to assign different threads to different interrupt handlers.
This allows for the "simultaneous" handling of multiple interrupts, and the assurance that high-priority interrupts will take precedence over low-priority ones and over user processes.

1.3.) Direct Memory Access:

- For devices that transfer large quantities of data (such as disk controllers), it is wasteful to tie up the CPU transferring data in and out of registers one byte at a time.
- Instead this work can be off-loaded to a special processor, known as the Direct Memory Access**, DMA, Controller.**

- The host issues a command to the DMA controller, indicating the location where the data is located, the location where the data is to be transferred to, and the number of bytes of data to transfer. The DMA controller handles the data transfer, and then interrupts the CPU when the transfer is complete.

- A simple DMA controller is a standard component in modern PCs, and many bus-mastering I/O cards contain their own DMA hardware.
- Handshaking between DMA controllers and their devices is accomplished through two wires called the DMA-request and DMA-acknowledge wires.

- While the DMA transfer is going on the CPU does not have access to the PCI bus (including main memory), but it does have access to its internal registers and primary and secondary caches.

DMA can be done in terms of either physical addresses or virtual addresses that are mapped to physical addresses. The latter approach is known as Direct Virtual Memory Access, DVMA, and allows direct data transfer from one memory-mapped device to another without using the main memory chips.
Direct DMA access by user processes can speed up operations, but is generally forbidden by modern systems for security and protection reasons. (I.e. DMA is a kernel-mode operation figure showing below illustrates the DMA process.

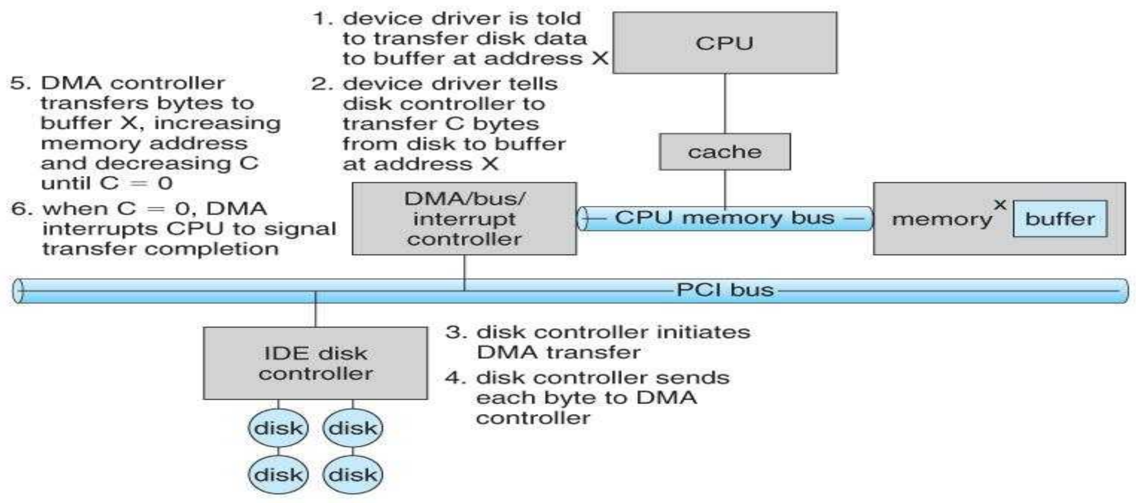

1. device driver is told to transfer disk data to buffer at address X
2. device driver tells disk controller to transfer C bytes from disk to buffer at address X
3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller
5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
6. when C = 0, DMA interrupts CPU to signal transfer completion
CPU
cache
DMA/bus/ interrupt controller
CPU memory bus
memory
x
buffer
PCI bus
IDE disk controller
disk
disk
disk
disk

Figure 13.5 - Steps in a DMA transfer.

The main concepts:
- A bus
- A controller
- An i/o port and its registers
- The handshaking relationship between the host and a device controller

- The offloading of the work to a DMA controller for large transfer.

2.) Application I/O Interface:--

User application access to a wide variety of different devices is accomplished through layering, and through encapsulating all of the device-specific code into **device drivers**, while application layers are presented with a common interface for all (or at least large general categories of) devices.

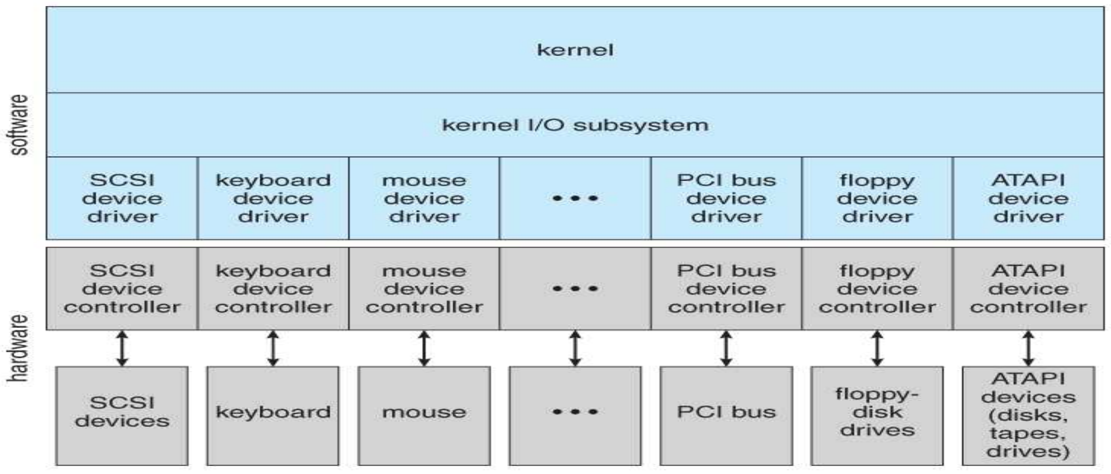
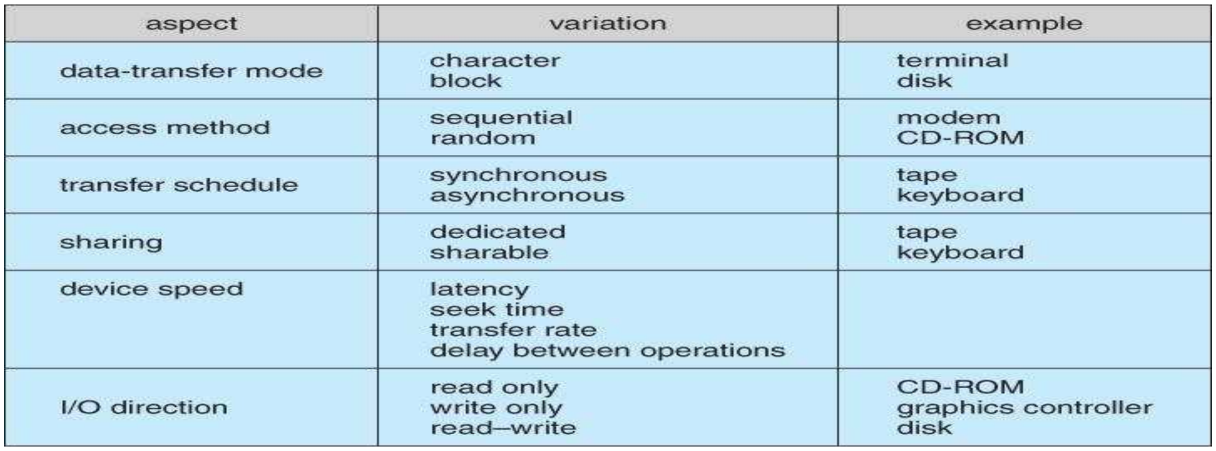


Fig: **A kernel I/O structure**.

Most devices can be characterized as either block I/O, character I/O, memory mapped file access, or network sockets. A few devices are special, such as time-of-day clock and the system timer.
Most of also have an **escape, or back door**, which allows applications to send commands directly to device drivers if needed. In UNIX this is the **ioctl( )** system call (I/O Control).

Ioctl( ) takes three arguments - The file descriptor for the device driver being accessed, an integer indicating the desired function to be performed, and an address used for communicating or transferring additional information.



aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only read–write	CD-ROM graphics controller disk

Figure 13.7 - Characteristics of I/O devices.

- **Character-stream or block:** A Character-stream device transfers bytes one by one, where as block device transfers a block of bytes as a unit.
- **Sequential or random-access:** A sequential device transfers data in affixed order determined by the device, where as the user of random-access device can instruct the device to seek to any of the available data storage locations.
- **Synchronous or asynchronous:** A Synchronous Device Is One That Performs Data Transfers With Predictable Response Times. An asynchronous Device Exhibits Irregular Or Unpredictable Response Times.
- **Sharable or dedicated:** A sharable device can be used concurrently by several process or threads; a dedicated device cannot.
- **Speed of operation:** Device Speeds Range From A Few Bytes Per Second To A Few Gigabytes Per Second,
- **Read-write, read only, or write only:** some devices perform both input and output, but others support only one data direction.

2.1 Block And Character Devices:

- One of the important jobs of an Operating System is to manage various I/O devices including mouse, keyboards, touch pad, disk drives, display adapters, USB devices, Bit-mapped screen, LED, Analog-to-digital converter, On/off switch, network connections, audio I/O, printers etc.
- An I/O system is required to take an application I/O request and send it to the physical device, then take whatever response comes back from the device and send it to the application. I/O devices can be divided into two categories –

Block devices

− A block device is one with which the driver communicates by sending entire blocks of data. For example, Hard disks, USB cameras, Disk-On-Key etc.
Character devices
− A character device is one with which the driver communicates by sending and receiving single characters (bytes, octets). For example, serial ports, parallel ports, sounds cards etc.

2.2 Network Devices.

- Because network access is inherently different from local disk access, most systems provide a separate interface for network devices.
- One common and popular interface is the socket interface, which acts like a cable or pipeline connecting two networked entities.
- Data can be put into the socket at one end, and read out sequentially at the other end. Sockets are normally full-duplex, allowing for bi-directional data transfer.
- The select( ) system call allows servers (or other applications) to identify sockets which have data waiting, without having to poll all available sockets.

2.3 Clocks and Timers.

- Three types of time services are commonly needed in modern systems:
- Get the current time of day.
- Get the elapsed time (system or wall clock) since a previous event.
- Set a timer to trigger event X at time T.
- Unfortunately time operations are not standard across all systems.

- **A programmable interrupt timer, PIT** can be used to trigger operations and to measure elapsed time. It can be set to trigger an interrupt at a specific future time, or to trigger interrupts periodically on a regular basis.
- The scheduler uses a PIT to trigger interrupts for ending time slices.The disk system may use a PIT to schedule periodic maintenance cleanup, such as flushing buffers to disk.
- Networks use PIT to abort or repeat operations that are taking too long to complete. I.e resending packets if an acknowledgement is not received before the timer goes off.
- More timers than actually exist can be simulated by maintaining an ordered list of timer events, and setting the physical timer to go off when the next scheduled event should occur.
- On most systems the system clock is implemented by counting interrupts generated by the PIT.

- Unfortunately this is limited in its resolution to the interrupt frequency of the PIT, and may be subject to some drift over time.
- An alternate approach is to provide direct access to a high frequency hardware counter, which provides much higher resolution and accuracy, but which does not support interrupts.

2.4 <u>**Blocking and Nonblocking I/O:**</u>

- Some control over how the wait for I/O to complete is accommodated is available to the programmer of user applications.
- Most I/O requests are considered blocking requests, meaning that control does not return to the application until the I/O is complete.
- The delayed from systems calls such read() and write() can be quite long.

- Using systems calls that block is sometimes called synchronous programming.

In most cases, the wait is not really a problem because the program can not do anything else until the I/O is finished. However, in cases such as network programming with multiple clients or with graphical user interface programming, the program may wish to perform other activity as it continues to wait for more data or input from users.

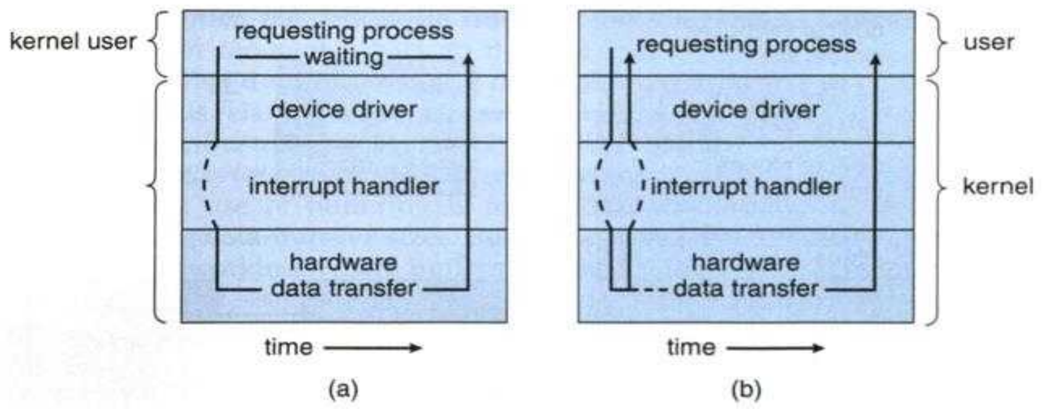
One solution for these situations is to use multiple threads so that one part of the program is not waiting for unrelated I/O to complete. Another alternative is to use asynchronous programming techniques with nonblocking system calls.

An asynchronous call returns immediately, without waiting for the I/O to complete. The completion of the I/O is later communicated to the application either through the setting of some variable in the application or through the triggering of a signal or call-back routine that is executed outside the linear control flow of the application.

Blocking I/O system calls (a) do not return until the I/O is complete. Nonblocking I/O system calls return immediately. The process is later notified when the I/O is complete.

A good example of nonblocking behavior is the select() system call for network sockets. Using select(), an application can monitor several resources at the same time and can also poll for network activity without blocking.

The select() system call identifies if data is pending or not, then read() or write() may be used knowing that they will complete immediately.



Two I/O methods: A) Synchronous and b) Asynchronous.

3. Kernel I/O Subsystem:

Kernels provide many services related to I/O. Several services—Scheduling, buffering, caching, spooling, device reservation, and error handling are provided by the kernel's I/O subsystem and build on the hardware and device- driver infrastructure.

I/O Scheduling:
Scheduling I/O requests can greatly improve overall efficiency. Priorities can also play a part of request scheduling.
Buffering and caching can also help, and can allow for more flexible scheduling options.
On systems with many devices, separate request queues are often kept for each device:

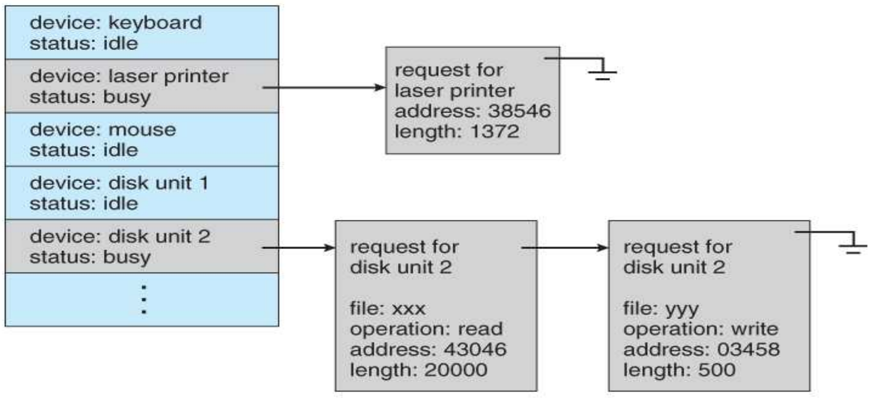


Figure 13.9 - Device-status table.

3.2 Buffering:

A Buffer is a memory area that stores data while they are transferred between two devices or between a device and an application.
Buffering of I/O is performed for 3 major reasons:

1.Speed differences between two devices. See Figure below A slow device may write data into a buffer, and when the buffer is full, the entire buffer is sent to the fast device all at once.
So that the slow device still has somewhere to write while this is going on, a second buffer is used, and the two buffers alternate as each becomes full.

This is known as double buffering. (Double buffering is often used in (animated) graphics, so that one screen image can be generated in a buffer while the other (completed) buffer is displayed on the screen.
This prevents the user from ever seeing any half-finished screen images.)

2.Data transfer size differences. Buffers are used in particular in networking systems to break messages up into smaller packets for transfer, and then for re-assembly at the receiving side.

3.To support copy semantics. For example, when an application makes a request for a disk write, the data is copied from the user's memory area into a kernel buffer.

Now the application can change their copy of the data, but the data which eventually gets written out to disk is the version of the data at the time the write request was made.

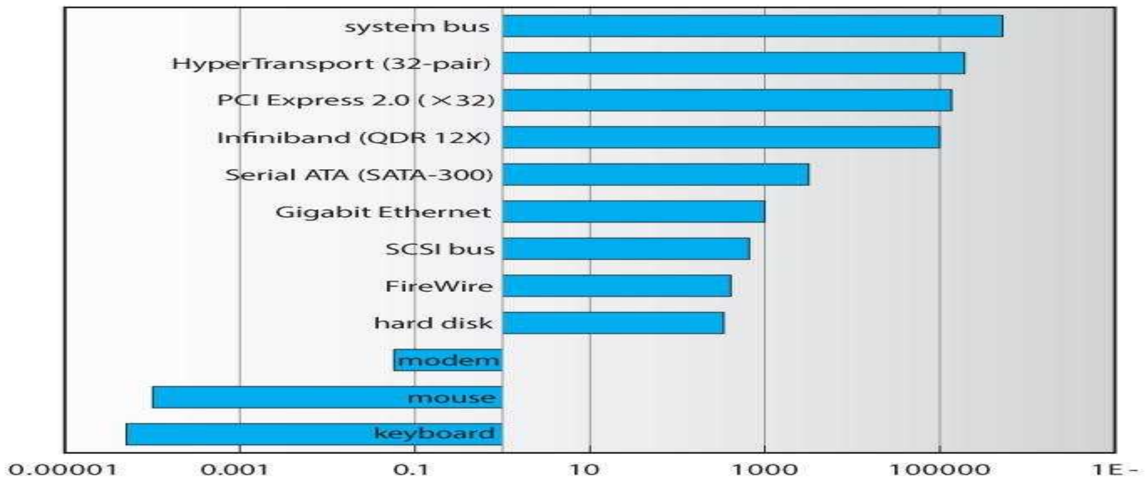


Figure - Sun Enterprise 6000 device-transfer rates (logarithmic).

3.3 Caching:

Caching involves keeping a copy of data in a faster-access location than where the data is normally stored.Buffering and caching are very similar, except that a buffer may hold the only copy of a given data item, whereas a cache is just a duplicate copy of some other data stored elsewhere.

Buffering and caching go hand-in-hand, and often the same storage space may be used for both purposes. For example, after a buffer is written to disk, then the copy in memory can be used as a cached copy, (until that buffer is needed for other purposes.)

3.4 Spooling and Device Reservation:

A spool (Simultaneous Peripheral Operations On-Line) buffers data for (peripheral) devices such as printers that cannot support interleaved data streams.

If multiple processes want to print at the same time, they each send their print data to files stored in the spool directory. When each file is closed, then the application sees that print job as complete, and the print scheduler sends each file to the appropriate printer one at a time.

Support is provided for viewing the spool queues, removing jobs from the queues, moving jobs from one queue to another queue, and in some cases changing the priorities of jobs in the queues.

Spool queues can be general (any laser printer) or specific (printer number 42.) OSes can also provide support for processes to request / get exclusive access to a particular device, and/or to wait until a device becomes available

3.5 Error Handling:

I/O requests can fail for many reasons, either transient (buffers overflow) or permanent (disk crash).
I/O requests usually return an error bit (or more) indicating the problem. UNIX systems also set the global variable errno to one of a hundred or so well-defined values to indicate the specific error that has occurred.
(See errno.h for a complete listing, or man errno.)
Some devices, such as SCSI devices, are capable of providing much more detailed information about errors, and even keep an on-board error log that can be requested by the host.

3.6 Kernel Data Structures:

The kernel maintains a number of important data structures pertaining to the I/O system, such as the open file table.

These structures are object-oriented, and flexible to allow access to a wide variety of I/O devices through a common interface. (See Figure below.)
Windows NT carries the object-orientation one step further, implementing I/O as a message-passing system from the source through various intermediaries to the device.

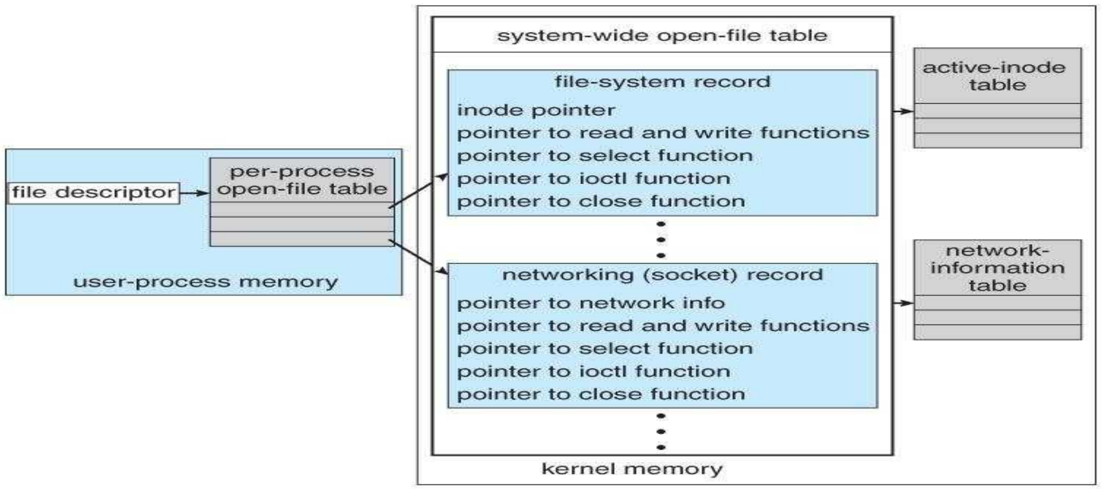

system-wide open-file table
file-system record
inode pointer
pointer to read and write functions
pointer to select function
pointer to ioctl function
pointer to close function
active-inode table
file descriptor
per-process open-file table
user-process memory
networking (socket) record
pointer to network info
pointer to read and write functions
pointer to select function
pointer to ioctl function
pointer to close function
network-information table
kernel memory

Figure - UNIX I/O kernel structure.

The I/O subsystem supervises:

- The management of the name space for files and devices.
- Access control to files and devices.

- Operation control
- File system space allocation
- Device allocation
- Buffering, catching, and spooling
- I/O scheduling

4. **Transforming I/O Requests to Hardware Operations**:--

- Users request data using file names, which must ultimately be mapped to specific blocks of data from a specific device managed by a specific device driver.

- DOS uses the colon separator to specify a particular device (e.g. C:, LPT:, etc.) UNIX uses a **mount table** to map filename prefixes (e.g. /usr) to specific mounted devices.

- Where multiple entries in the mount table match different prefixes of the filename the one that matches the longest prefix is chosen. (e.g. /usr/home instead of /usr where both exist in the mount table and both match the desired file).

- UNIX **uses special device files**, usually located in /dev, to represent and access physical devices directly.
  Each device file has a major and minor number associated with it, stored and displayed where the file size would normally go.

- The major number is an index into a table of device drivers, and indicates which device driver handles this device. (E.g. the disk drive handler).

- The minor number is a parameter passed to the device driver, and indicates which specific device is to be accessed, out of the many which may be handled by a particular device driver.

A series of lookup tables and mappings makes the access of different devices flexible, and somewhat transparent to users.

Below Figure showing the steps taken to process a (blocking) read request:

1. A process issues a blocking read() system call to a file descriptor of a file that has been **opened** previously.

2. The system call code in the kernel checks the parameters for correctness. In the case of input, if the data are already available in the buffer cache, the data are returned to the process and the I/O request is completed.

3. Otherwise, a physical I/O needs to be performed, so the process is removed from the run queue and is placed on the wait queue for the device, and the I/O request is scheduled.

   The I/O subsystem sends the request to the device driver. Depending on the operating system, the request is sent via a subroutine call or via an in-kernel message.

4. The device driver allocates kernel buffer space to receive the data, and schedules the I/O.
   The driver sends commands to the device controller by writing into the device control registers.
5. The device controller operates the device hardware to perform the data transfer.

6. The driver may poll for status and data, or it may have set up a DMA transfer into kernel memory. We assume that the transfer is managed by a DMA controller, which generates an interrupt when the transfer completes.

7. 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.

8. The device driver receives the signal, determines which I/O request completed, determines the requests status, and signal the kernel I/O system that the request has been completed.

9. The kernel transfers data or return codes to the address space of the requesting process, and moves the process from the wait queue back to the ready queue.

10. Moving the process to the ready queue unblocked the process.

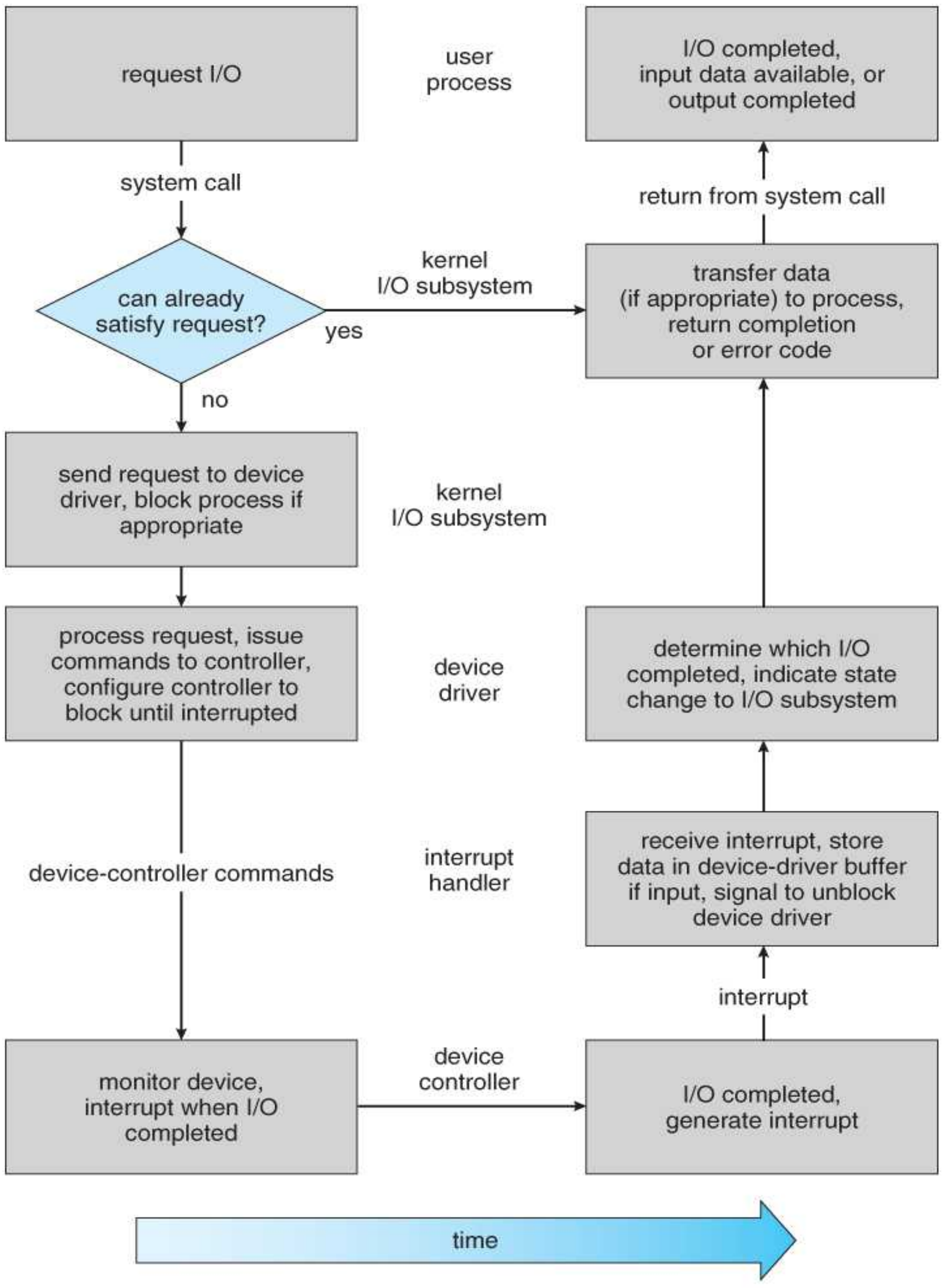

request I/O
user process
I/O completed, input data available, or output completed
system call
return from system call
can already satisfy request?
kernel I/O subsystem
yes
transfer data (if appropriate) to process, return completion or error code
no
send request to device driver, block process if appropriate
kernel I/O subsystem
process request, issue commands to controller, configure controller to block until interrupted
device driver
determine which I/O completed, indicate state change to I/O subsystem
device-controller commands
interrupt handler
receive interrupt, store data in device-driver buffer if input, signal to unblock device driver
interrupt
monitor device, interrupt when I/O completed
device controller
I/O completed, generate interrupt
time

Figure - The life cycle of an I/O request.

5. STREAMS :--

- The streams mechanism in UNIX provides a bi-directional pipeline between a user process and a device driver, onto which additional modules can be added.
- The user process interacts with the stream head. The device driver interacts with the device end.
- Zero or more stream modules can be pushed onto the stream, using ioctl( ). These modules may filter and/or modify the data as it passes through the stream. Each module has a read queue and a write queue.
- Flow control can be optionally supported, in which case each module will buffer data until the adjacent module is ready to receive it. Without flow control, data is passed along as soon as it is ready.
- User processes communicate with the stream head using either read( ) and write( ) (or putmsg(and getmsg( ) for message passing). Streams I/O is asynchronous (non-blocking), except for the interface between the user process and the stream head.

The device driver must respond to interrupts from its device - If the adjacent module is not prepared to accept data and the device driver's buffers are all full, then data is typically dropped.

Streams are widely used in UNIX, and are the preferred approach for device drivers. For example, UNIX implements sockets using streams.

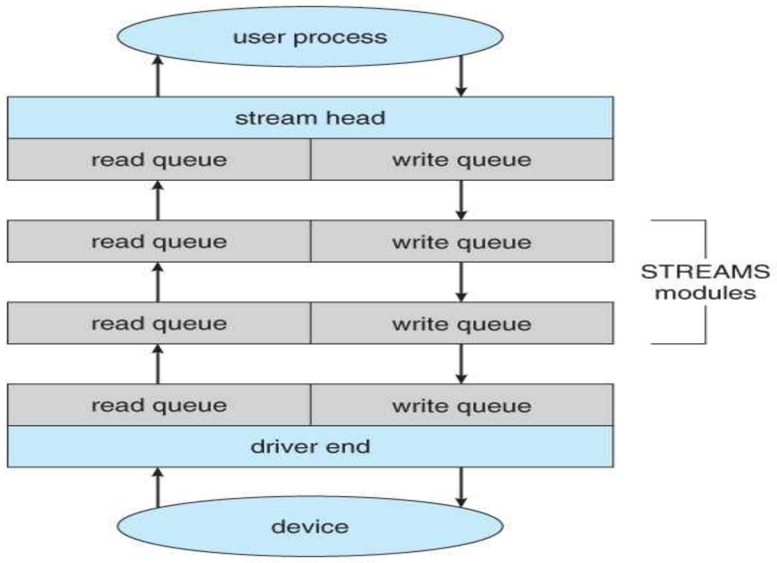


Figure - The SREAMS structure.

6. Performance :--

- The I/O system is a major factor in overall system performance, and can place heavy loads on other major components of the system (interrupt handling, process switching, memory access, bus contention, and CPU load for device drivers just to name a few).

- Interrupt handling can be relatively expensive (slow), which causes programmed I/O to be faster than interrupt-driven I/O when the time spent busy waiting is not excessive.
  Network traffic can also put a heavy load on the system.

Consider for example the sequence of events that occur when a single character is typed in a telnet session, as shown in figure .

(And the fact that a similar set of events must happen in reverse to echo back the character that was typed.)

Sun uses in-kernel threads for the telnet daemon, increasing the supportable number of simultaneous telnet sessions from the hundreds to the thousands.

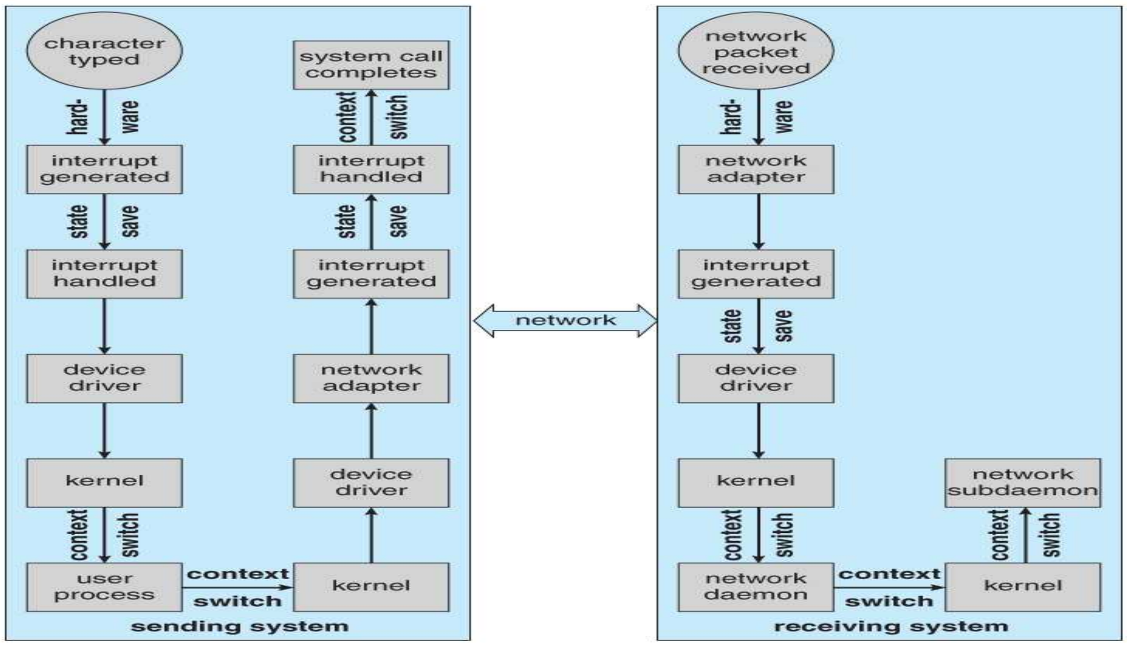


Figure 13.15 - Intercomputer communications.

Other systems use front-end processors to off-load some of the work of I/O processing from the CPU. For example a terminal concentrator can multiplex with hundreds of terminals on a single port on a large computer.

- Several principles can be employed to increase the overall efficiency of I/O processing:
  Reduce the number of context switches.Reduce the number of times data must be copied.

- Reduce interrupt frequency, using large transfers, buffering, and polling where appropriate.

- Increase concurrency using DMA.

- Move processing primitives into hardware, allowing their operation to be concurrent with CPU and bus operations.
- Balance CPU, memory, bus, and I/O operations, so a bottleneck in one does not idle all the others.

- The development of new I/O algorithms often follows a progression from application level code to on-board hardware implementation, as shown in Figure Lower-level implementations are faster and more efficient, but higher-level ones are more flexible and easier to modify.
- Hardware-level functionality may also be harder for higher-level authorities (e.g. the kernel) to control.

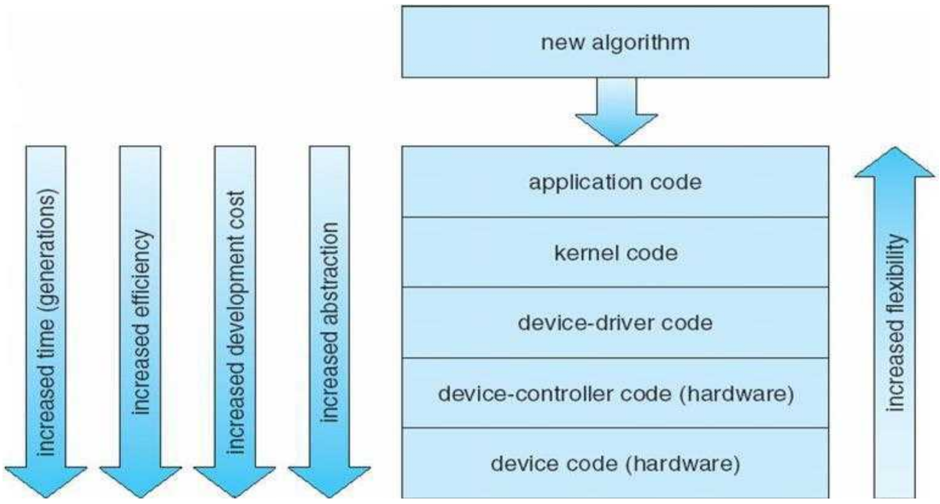


Figure: Device-functionality progression

Chapter -13

MASS STORAGE STRUCTURE

TOPICS:

- **Disk Structure**
- **Disk Scheduling**
- **Disk Management**
- **Swap –space Management**
- **RAID Structure**

Chapter -13

MASS STORAGE STRUCTURE

TOPICS:

- **Disk Structure**
- **Disk Scheduling**
- **Disk Management**
- **Swap –space Management**
- **RAID Structure**

Disk Structure

- The traditional head-sector-cylinder, HSC numbers are mapped to linear block addresses by numbering the first sector on the first head on the outermost track as sector 0.
- Numbering proceeds with the rest of the sectors on that same track, and then the rest of the tracks on the same cylinder before proceeding through the rest of the cylinders to the center of the disk.
- In modern practice these linear block addresses are used in place of the **HSC numbers for a variety of reasons:**
    1. The linear length of tracks near the outer edge of the disk is much longer than for those tracks located near the center, and therefore it is possible to squeeze many more sectors onto outer tracks than onto inner ones.
    2. All disks have some bad sectors, and therefore disks maintain a few spare sectors that can be used in place of the bad ones. The mapping of spare sectors to bad sectors in managed internally to the disk controller.
    3. Modern hard drives can have thousands of cylinders, and hundreds of sectors per track on their outermost tracks. These numbers exceed the range of HSC numbers for many (older) operating systems, and

therefore disks can be configured for any convenient combination of HSC values that falls within the total number of sectors physically on the drive.

- There is a limit to how closely packed individual bits can be placed on a physical media, but that limit is growing increasingly more packed as technological advances are made.
- Modern disks pack many more sectors into outer cylinders than inner ones, using **one of two approaches**:
- With ***Constant Linear Velocity, CLV,*** the density of bits is uniform from cylinder to cylinder. Because there are more sectors in outer cylinders, the disk spins slower when reading those cylinders, causing the rate of bits passing under the read-write head to remain constant. This is the approach used by modern CDs and DVDs.
- With ***Constant Angular Velocity, CAV,*** the disk rotates at a constant angular speed, with the bit density decreasing on outer cylinders. (These disks would have a constant number of sectors per track on all cylinders.)

Disk scheduling is done by operating systems to schedule I/O requests arriving for the disk. Disk scheduling is also known as I/O scheduling.
Disk scheduling is important because:

- Multiple I/O requests may arrive by different processes and only one I/O request can be served at a time by the disk controller. Thus other I/O requests need to wait in the waiting queue and need to be scheduled.
- Two or more request may be far from each other so can result in greater disk arm movement.
- Hard drives are one of the slowest parts of the computer system and thus need to be accessed in an efficient manner.

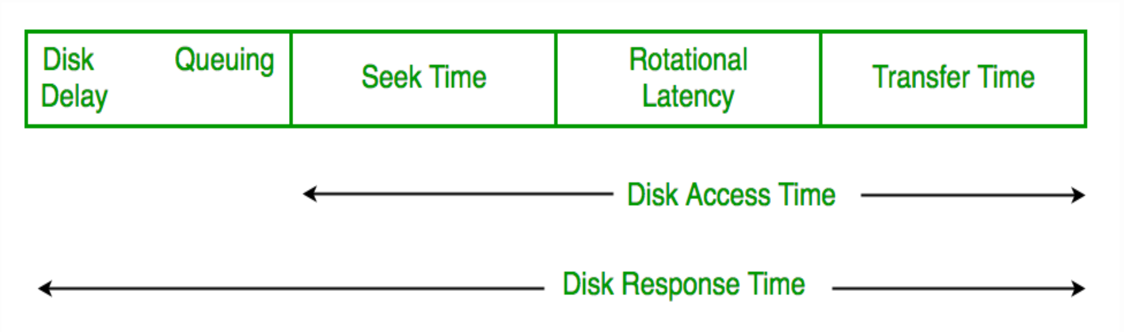
There are many Disk Scheduling Algorithms but before discussing them let's have a quick look at some of the important terms:

- Seek Time**:**Seek time is the time taken to locate the disk arm to a specified track where the data is to be read or write. So the disk scheduling algorithm that gives minimum average seek time is better.
- Rotational Latency: Rotational Latency is the time taken by the desired sector of disk to rotate into a position so that it can access the read/write heads. So the disk scheduling algorithm that gives minimum rotational latency is better.
- Transfer Time: Transfer time is the time to transfer the data. It depends on the rotating speed of the disk and number of bytes to be transferred.
- Disk Access Time: Disk Access Time is:

Disk Access Time = Seek Time +

Rotational Latency +

Transfer Time



- Disk Response Time: Response Time is the average of time spent by a request waiting to perform its I/O operation. *Average Response time* is the response time of the all requests. *Variance Response Time* is measure of how individual request are serviced with respect to average response time. So the disk scheduling algorithm that gives minimum variance response time is better.

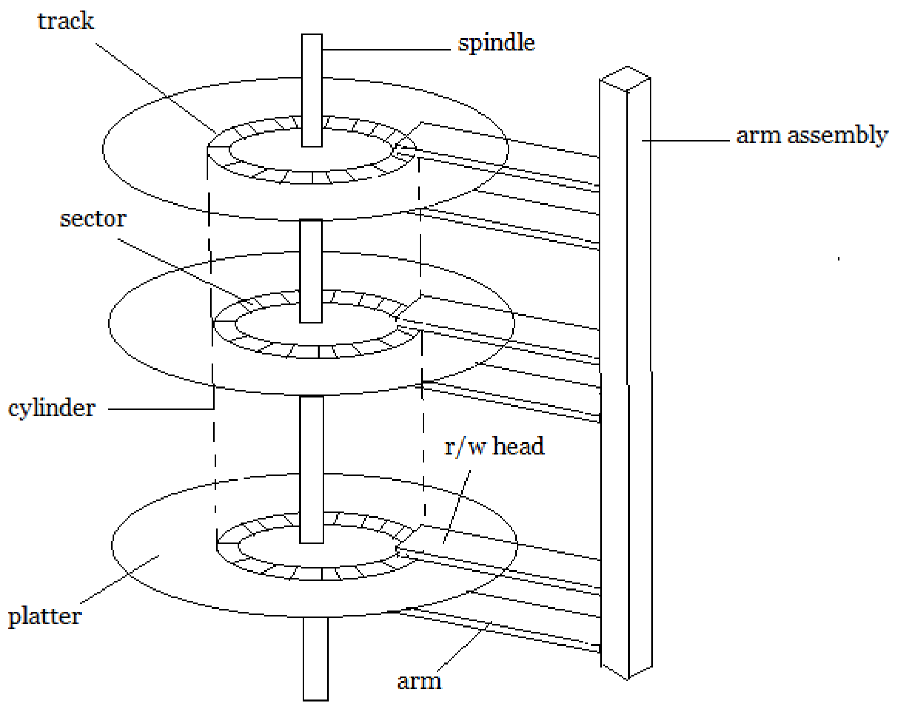
Secondary Storage and Disk Scheduling Algorithms

Secondary storage devices are those devices whose memory is non volatile, meaning, the stored data will be intact even if the system is turned off. Here are a few things worth noting about secondary storage.

- Secondary storage is also called auxiliary storage.
- Secondary storage is less expensive when compared to primary memory like RAMs.
- The speed of the secondary storage is also lesser than that of primary storage.
- Hence, the data which is less frequently accessed is kept in the secondary storage.
- A few examples are magnetic disks, magnetic tapes, removable thumb drives etc.

Magnetic Disk Structure

In modern computers, most of the secondary storage is in the form of magnetic disks. Hence, knowing the structure of a magnetic disk is necessary to understand how the data in the disk is accessed by the computer.



Structure of a magnetic disk

A magnetic disk contains several **platters**. Each platter is divided into circular shaped **tracks**. The length of the tracks near the centre is less than the length of the tracks farther from the centre. Each track is further divided into **sectors**, as shown in the figure.

Tracks of the same distance from centre form a cylinder. A read-write head is used to read data from a sector of the magnetic disk.

The speed of the disk is measured as two parts:

- **Transfer rate:** This is the rate at which the data moves from disk to the computer.
- **Random access time:** It is the sum of the seek time and rotational latency.

Seek time is the time taken by the arm to move to the required track. **Rotational latency** is defined as the time taken by the arm to reach the required sector in the track.

Even though the disk is arranged as sectors and tracks physically, the data is logically arranged and addressed as an array of blocks of fixed size. The size of a block can be **512** or **1024** bytes. Each logical block is mapped with a sector on the disk, sequentially. In this way, each sector in the disk will have a logical address.

Disk Scheduling Algorithms:

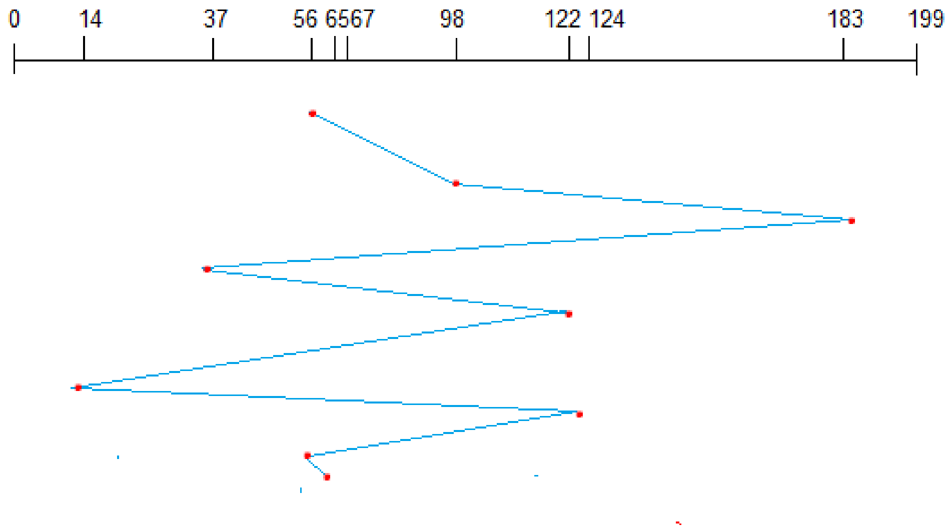
On a typical multiprogramming system, there will usually be multiple disk access requests at any point of time. So those requests must be scheduled to achieve good efficiency. Disk scheduling is similar to process scheduling. Some of the disk scheduling algorithms are described below.

First Come First Serve

This algorithm performs requests in the same order asked by the system. Let's take an example where the queue has the following requests with cylinder numbers as follows:

98, 183, 37, 122, 14, 124, 65, 67

Assume the head is initially at cylinder **56**. The head moves in the given order in the queue i.e., **56→98→183→...→67**.



FCFS: FCFS is the simplest of all the Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue.

Advantages:

- Every request gets a fair chance
- No indefinite postponement

Disadvantages:

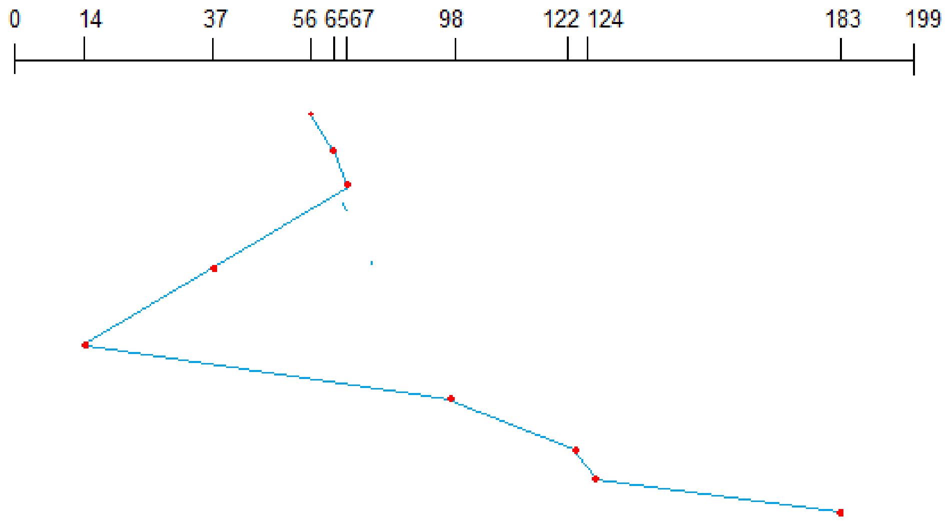
- Does not try to optimize seek time
- May not provide the best possible service

Shortest Seek Time First (SSTF)

Here the position which is closest to the current head position is chosen first. Consider the previous example where disk queue looks like,

98, 183, 37, 122, 14, 124, 65, 67

Assume the head is initially at cylinder **56**. The next closest cylinder to **56** is **65**, and then the next nearest one is **67**, then **37**, **14**, so on.



In SSTF (Shortest Seek Time First), requests having shortest seek time are executed first. So, the seek time of every request is calculated in advance in the queue and then they are scheduled according to their calculated seek time. As a result, the request near the disk arm will get executed first. SSTF is certainly an improvement over FCFS as it decreases the average response time and increases the throughput of system.

Shortest seek time first (SSTF) algorithm selects the disk I/O request which requires the least disk arm movement from its current position regardless of the direction. It reduces the total seek time as compared to FCFS.

Advantages:

- Average Response Time decreases
- Throughput increases

Disadvantages:

- Overhead to calculate seek time in advance

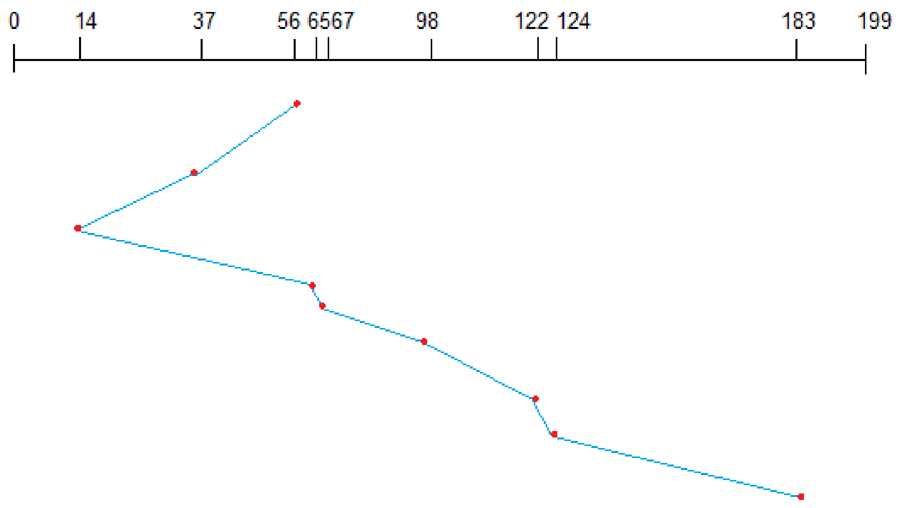
- Can cause Starvation for a request if it has higher seek time as compared to incoming requests
- High variance of response time as SSTF favours only some requests

SCAN algorithm

- ➢ This algorithm is also called the elevator algorithm because of it's behavior. Here, first the head moves in a direction (say backward) and covers all the requests in the path.
- ➢ Then it moves in the opposite direction and covers the remaining requests in the path. This behavior is similar to that of an elevator. Let's take the previous example,

98, 183, 37, 122, 14, 124, 65, 67

Assume the head is initially at cylinder **56**. The head moves in backward direction and accesses **37** and **14**. Then it goes in the opposite direction and accesses the cylinders as they come in the path.



- ➢ In SCAN algorithm the disk arm moves into a particular direction and services the requests coming in its path and after reaching the end of disk, it reverses its direction and again services the request arriving in its path.
- ➢ So, this algorithm works as an elevator and hence also known as **elevator algorithm.** As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

Advantages:

- High throughput
- Low variance of response time
- Average response time

Disadvantages:

- Long waiting time for requests for locations just visited by disk arm.

CSCAN:

- In SCAN algorithm, the disk arm again scans the path that has been scanned, after reversing its direction. So, it may be possible that too many requests are waiting at the other end or there may be zero or few requests pending at the scanned area.
- These situations are avoided in *CSAN* algorithm in which the disk arm instead of reversing its direction goes to the other end of the disk and starts servicing the requests from there.
- So, the disk arm moves in a circular fashion and this algorithm is also similar to SCAN algorithm and hence it is known as C-SCAN (Circular SCAN).

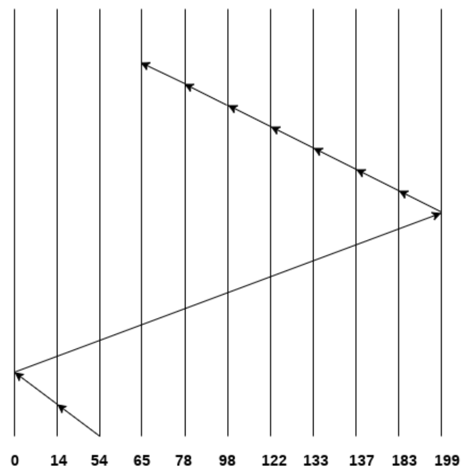
- In C-SCAN algorithm, the arm of the disk moves in a particular direction servicing requests until it reaches the last cylinder, then it jumps to the last cylinder of the opposite direction without servicing any request then it turns back and start moving in that direction servicing the remaining requests.

Example

Consider the following disk request sequence for a disk with 100 tracks

98, 137, 122, 183, 14, 133, 65, 78

Head pointer starting at 54 and moving in left direction. Find the number of head movements in cylinders using C-SCAN scheduling.



No. of cylinders crossed = 40 + 14 + 199 + 16 + 46 + 4 + 11 + 24 + 20 + 13 = 387

Advantages:

- Provides more uniform wait time compared to SCAN

LOOK: It is similar to the SCAN disk scheduling algorithm except for the difference that the disk arm in spite of going to the end of the disk goes only to the last request to be serviced in front of the head and then reverses its direction from there only. Thus it prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

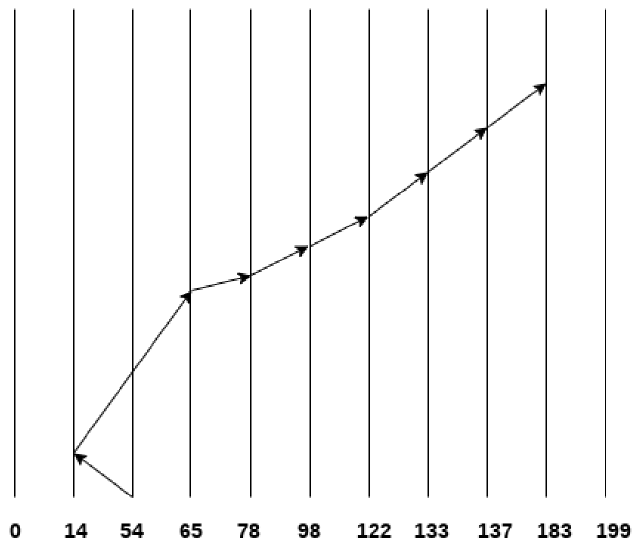
- It is like SCAN scheduling Algorithm to some extant except the difference that, in this scheduling algorithm, the arm of the disk stops moving inwards (or outwards) when no more request in that direction exists.
- This algorithm tries to overcome the overhead of SCAN algorithm which forces disk arm to move in one direction till the end regardless of knowing if any request exists in the direction or not.

Example

Consider the following disk request sequence for a disk with 100 tracks

98, 137, 122, 183, 14, 133, 65, 78

Head pointer starting at 54 and moving in left direction. Find the number of head movements in cylinders using LOOK scheduling.



Number of cylinders crossed = 40 + 51 + 13 + +20 + 24 + 11 + 4 + 46 = 209

C Look Scheduling

- C Look Algorithm is similar to C-SCAN algorithm to some extent. In this algorithm, the arm of the disk moves outwards servicing requests until it reaches the highest request cylinder, then it jumps to the lowest request cylinder without servicing any request then it again start moving outwards servicing the remaining requests.
- It is different from C SCAN algorithm in the sense that, C SCAN force the disk arm to move till the last cylinder regardless of knowing whether any request is to be serviced on that cylinder or not.

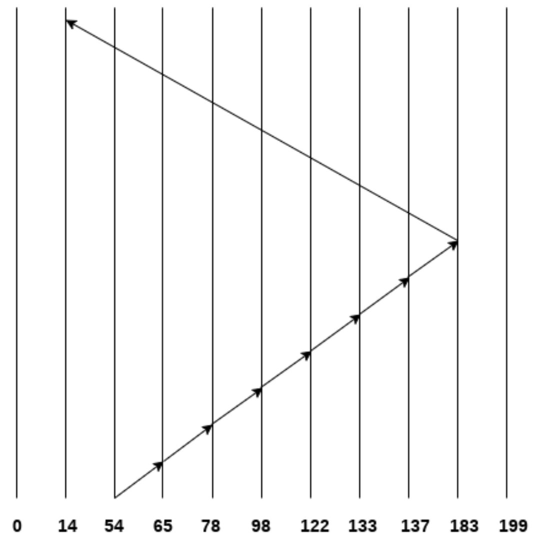
Each algorithm is unique in its own way. Overall Performance depends on the number and type of requests.

Example

Consider the following disk request sequence for a disk with 100 tracks

98, 137, 122, 183, 14, 133, 65, 78

Head pointer starting at 54 and moving in left direction. Find the number of head movements in cylinders using C LOOK scheduling.



Number of cylinders crossed = 11 + 13 + 20 + 24 + 11 + 4 + 46 + 169 = 298

Selection of a disk scheduling Algorithm:

- With very low loads all algorithms are equal, since there will normally only be one request to process at a time.
- For slightly larger loads, SSTF offers better performance than FCFS, but may lead to starvation when loads become heavy enough.
- For busier systems, SCAN and LOOK algorithms eliminate starvation problems.
- The actual optimal algorithm may be something even more complex than those discussed here, but the incremental improvements are generally not worth the additional overhead.

Disk Management:

Disk Formatting

- Before a disk can be used, it has to be **low-level formatted**, which means laying down all of the headers and trailers marking the beginning and ends of each sector.
- Included in the header and trailer are the linear sector numbers, and **error-correcting codes, ECC,** which allow damaged sectors to not only be detected, but in many cases for the damaged data to be recovered (depending on the extent of the damage.)
- Sector sizes are traditionally 512 bytes, but may be larger, particularly in larger drives.

- ECC calculation is performed with every disk read or write, and if damage is detected but the data is recoverable, then a **soft error** has occurred.
- Soft errors are generally handled by the on-board disk controller, and never seen by the OS. (See below.)
- Once the disk is low-level formatted, the next step is to partition the drive into one or more separate partitions.
- This step must be completed even if the disk is to be used as a single large partition, so that the partition table can be written to the beginning of the disk.

Boot Block

- Computer ROM contains a **bootstrap** program (OS independent) with just enough code to find the first sector on the first hard drive on the first controller, load that sector into memory, and transfer control over to it.
- (The ROM bootstrap program may look in floppy and/or CD drives before accessing the hard drive, and is smart enough to recognize whether it has found valid boot code or not.)
- The first sector on the hard drive is known as the **Master Boot Record, MBR,** and contains a very small amount of code in addition to the **partition table.**
- The partition table documents how the disk is partitioned into logical disks, and indicates specifically which partition is the **active** or **boot** partition.
- The boot program then looks to the active partition to find an operating system, possibly loading up a slightly larger / more advanced boot program along the way.
- In a **dual-boot** (or larger multi-boot) system, the user may be given a choice of which operating system to boot, with a default action to be taken in the event of no response within some time frame.
- Once the kernel is found by the boot program, it is loaded into memory and then control is transferred over to the OS.
- The kernel will normally continue the boot process by initializing all important kernel data structures, launching important system services (e.g. network daemons, sched, init, etc.), and finally providing one or more login prompts.
- Boot options at this stage may include **single-user** a.k.a. **maintenance** or **safe** modes, in which very few system services are started - These modes are designed for system administrators to repair problems or otherwise maintain the system.

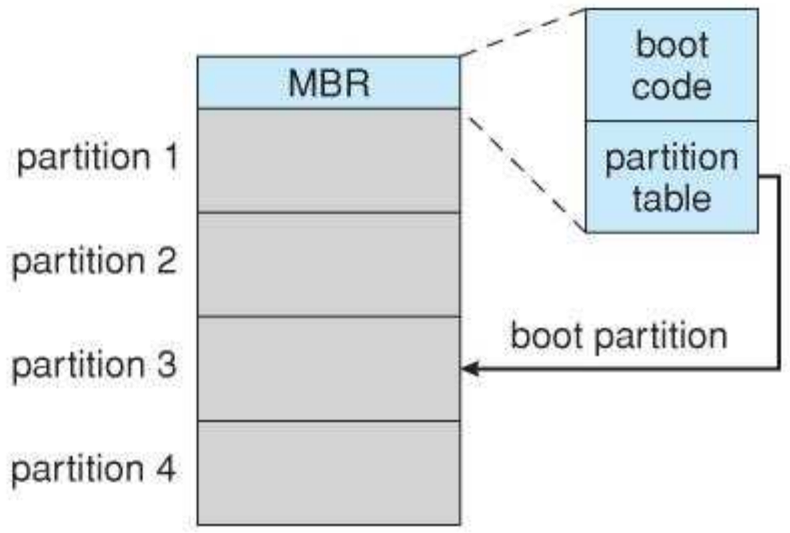


Figure - Booting from disk in Windows 2000.

Bad Blocks:

- No disk can be manufactured to 100% perfection, and all physical objects wear out over time.
- For these reasons all disks are shipped with a few bad blocks, and additional blocks can be expected to go bad slowly over time.
- If a large number of blocks go bad then the entire disk will need to be replaced, but a few here and there can be handled through other means.
- In the old days, bad blocks had to be checked for manually. Formatting of the disk or running certain disk-analysis tools would identify bad blocks, and attempt to read the data off of them one last time through repeated tries.
- Then the bad blocks would be mapped out and taken out of future service. Sometimes the data could be recovered, and sometimes it was lost forever. (Disk analysis tools could be either destructive or non-destructive.)
- Modern disk controllers make much better use of the error-correcting codes, so that bad blocks can be detected earlier and the data usually recovered.
- **Sector slipping** may also be performed, in which all sectors between the bad sector and the replacement sector are moved down by one, so that the linear progression of sector numbers can be maintained.
- If the data on a bad block cannot be recovered, then a **hard error** has occurred., which requires replacing the file(s) from backups, or rebuilding them from scratch.

Swap-Space Management:

- Modern systems typically swap out pages as needed, rather than swapping out entire processes. Hence the swapping system is part of the virtual memory management system.
- Managing swap space is obviously an important task for modern OS.

Swap-Space Use

- The amount of swap space needed by an OS varies greatly according to how it is used. Some systems require an amount equal to physical RAM; some want a multiple of that; some want an amount equal to the amount by which virtual memory exceeds physical RAM, and some systems use little or none at all!
- Some systems support multiple swap spaces on separate disks in order to speed up the virtual memory system.

Swap-Space Location

Swap space can be physically located in one of two locations:

- As a large file which is part of the regular filesystem. This is easy to implement, but inefficient. Not only must the swap space be accessed through the directory system, the file is also subject to fragmentation issues. Caching the block location helps in finding the physical blocks, but that is not a complete fix.
- As a raw partition, possibly on a separate or little-used disk. This allows the OS more control over swap space management, which is usually faster and more efficient. Fragmentation of swap space is generally not a big issue, as the space is re-initialized every time the system is rebooted. The downside of keeping swap space on a raw partition is that it can only be grown by repartitioning the hard drive.

Swap-Space Management: An Example

- Historically OSes swapped out entire processes as needed. Modern systems swap out only individual pages, and only as needed. (For example process code blocks and other blocks that have not been changed since they were originally loaded are normally just freed from the virtual memory system rather than copying them to swap space, because it is faster to go find them again in the filesystem and read them back in from there than to write them out to swap space and then read them back.)
- In the mapping system shown below for Linux systems, a map of swap space is kept in memory, where each entry corresponds to a 4K block in the swap space. Zeros indicate free slots and non-zeros refer to how many processes have a mapping to that particular block (>1 for shared pages only.)

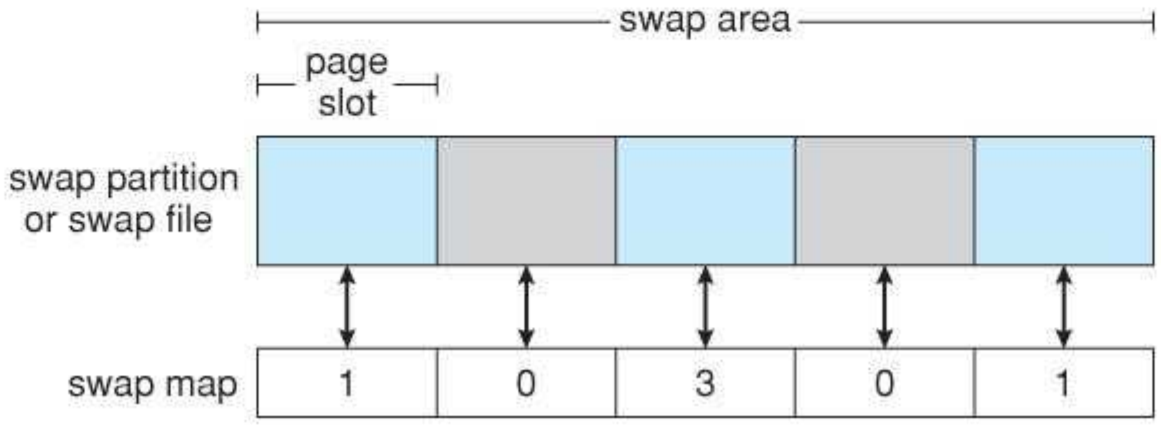


Figure 10.10 - The data structures for swapping on Linux systems.

RAID Structure

- The general idea behind RAID is to employ a group of hard drives together with some form of duplication, either to increase reliability or to speed up operations, (or sometimes both.)
- **RAID** originally stood for **Redundant Array of Inexpensive Disks,** and was designed to use a bunch of cheap small disks in place of one or two larger more expensive ones. Today RAID systems employ large possibly expensive disks as their components, switching the definition to **Independent** disks.

Improvement of Reliability via Redundancy

- The more disks a system has, the greater the likelihood that one of them will go bad at any given time. Hence increasing disks on a system actually **decreases** the **Mean Time To Failure, MTTF** of the system.
- If, however, the same data was copied onto multiple disks, then the data would not be lost unless **both** (or all) copies of the data were damaged simultaneously, which is a **MUCH** lower probability than for a single disk going bad. More specifically, the second disk would have to go bad before the first disk was repaired, which brings the **Mean Time To Repair** into play.
- For example if two disks were involved, each with a MTTF of 100,000 hours and a MTTR of 10 hours, then the **Mean Time to Data Loss** would be 500 * 10^6 hours, or 57,000 years!
- This is the basic idea behind disk **mirroring**, in which a system contains identical data on two or more disks.
- Note that a power failure during a write operation could cause both disks to contain corrupt data, if both disks were writing simultaneously at the time of the power failure.

- One solution is to write to the two disks in series, so that they will not both become corrupted (at least not in the same way) by a power failure. And alternate solution involves non-volatile RAM as a write cache, which is not lost in the event of a power failure and which is protected by error-correcting codes.

Improvement in Performance via Parallelism

- There is also a performance benefit to mirroring, particularly with respect to reads. Since every block of data is duplicated on multiple disks, read operations can be satisfied from any available copy, and multiple disks can be reading different data blocks simultaneously in parallel.
- Another way of improving disk access time is with ***striping***, which basically means spreading data out across multiple disks that can be accessed simultaneously.

With **bit-level striping** the bits of each byte are striped across multiple disks. For example if 8 disks were involved, then each 8-bit byte would be read in parallel by 8 heads on separate disks.

A single disk read would access 8 * 512 bytes = 4K worth of data in the time normally required to read 512 bytes. Similarly if 4 disks were involved, then two bits of each byte could be stored on each disk, for 2K worth of disk access per read or write operation.

Block-level striping spreads a filesystem across multiple disks on a block-by-block basis, so if block N were located on disk 0, then block N + 1 would be on disk 1, and so on.

This is particularly useful when filesystems are accessed in **clusters** of physical blocks. Other striping possibilities exist, with block-level striping being the most common.

RAID Levels

- ➢ Mirroring provides reliability but is expensive; Striping improves performance, but does not improve reliability.
- ➢ Accordingly there are a number of different schemes that combine the principals of mirroring and striping in different ways, in order to balance reliability versus performance versus cost.
- ➢ These are described by different ***RAID levels***, as follows: (In the diagram that follows, "C" indicates a copy, and "P" indicates parity, i.e. checksum bits.)

Raid Level 0 - This level includes striping only, with no mirroring.

Raid Level 1 - This level includes mirroring only, no striping.

Raid Level 2 - This level stores error-correcting codes on additional disks, allowing for any damaged data to be reconstructed by subtraction from the remaining undamaged data.

Note that this scheme requires only three extra disks to protect 4 disks worth of data, as opposed to full mirroring. (The number of disks required is a function of the error-correcting algorithms, and the means by which the particular bad bit(s) is(are) identified.)

Raid Level 3 - This level is similar to level 2, except that it takes advantage of the fact that each disk is still doing its own error-detection, so that when an error occurs, there is no question about which disk in the array has the bad data.

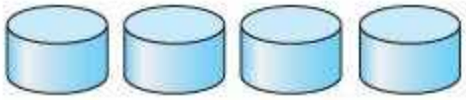
As a result a single parity bit is all that is needed to recover the lost data from an array of disks. Level 3 also includes striping, which improves performance. The downside with the parity approach is that every disk must take part in every disk access, and the parity bits must be constantly calculated and checked, reducing performance.

Hardware-level parity calculations and NVRAM cache can help with both of those issues. In practice level 3 is greatly preferred over level 2.

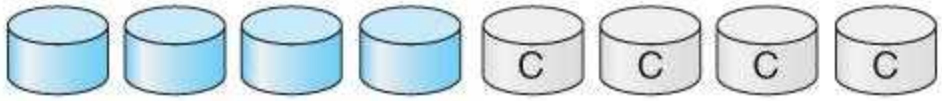
Raid Level 4 - This level is similar to level 3, employing block-level striping instead of bit-level striping. The benefits are that multiple blocks can be read independently, and changes to a block only require writing two blocks (data and parity) rather than involving all disks. Note that new disks can be added seamlessly to the system provided they are initialized to all zeros, as this does not affect the parity results.

Raid Level 5 - This level is similar to level 4, except the parity blocks are distributed over all disks, thereby more evenly balancing the load on the system. For any given block on the disk(s), one of the disks will hold the parity information for that block and the other N-1 disks will hold the data. Note that the same disk cannot hold both data and parity for the same block, as both would be lost in the event of a disk crash.

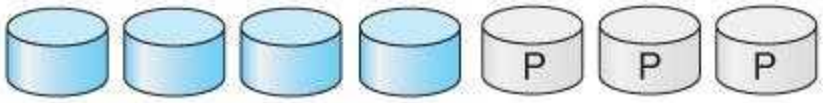
Raid Level 6 - This level extends raid level 5 by storing multiple bits of error-recovery codes, (such as the ***Reed-Solomon codes***), for each bit position of data, rather than a single parity bit. In the example shown below 2 bits of ECC are stored for every 4 bits of data, allowing data recovery in the face of up to two simultaneous disk failures. Note that this still involves only 50% increase in storage needs, as opposed to 100% for simple mirroring which could only tolerate a single disk failure.



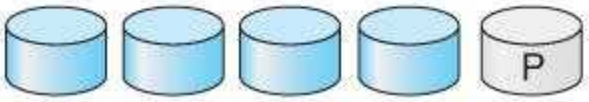
(a) RAID 0: non-redundant striping.



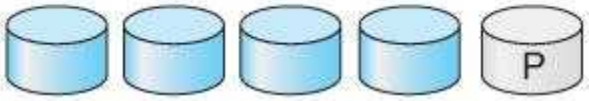
(b) RAID 1: mirrored disks.



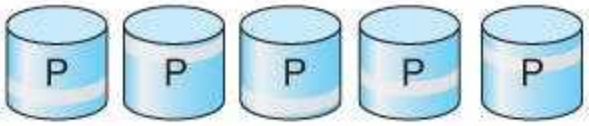
(c) RAID 2: memory-style error-correcting codes.



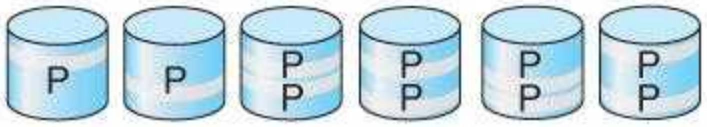
(d) RAID 3: bit-interleaved parity.



(e) RAID 4: block-interleaved parity.



(f) RAID 5: block-interleaved distributed parity.



(g) RAID 6: P + Q redundancy.

Figure - RAID levels.

- There are also two RAID levels which combine RAID levels 0 and 1 (striping and mirroring) in different combinations, designed to provide both performance and reliability at the expense of increased cost.
- **RAID level 0 + 1** disks are first striped, and then the striped disks mirrored to another set. This level generally provides better performance than RAID level 5.
- **RAID level 1 + 0** mirrors disks in pairs, and then stripes the mirrored pairs. The storage capacity, performance, etc. are all the same, but there is an advantage to this approach in the event of multiple disk failures, as illustrated below:.

In diagram (a) below, the 8 disks have been divided into two sets of four, each of which is striped, and then one stripe set is used to mirror the other set.

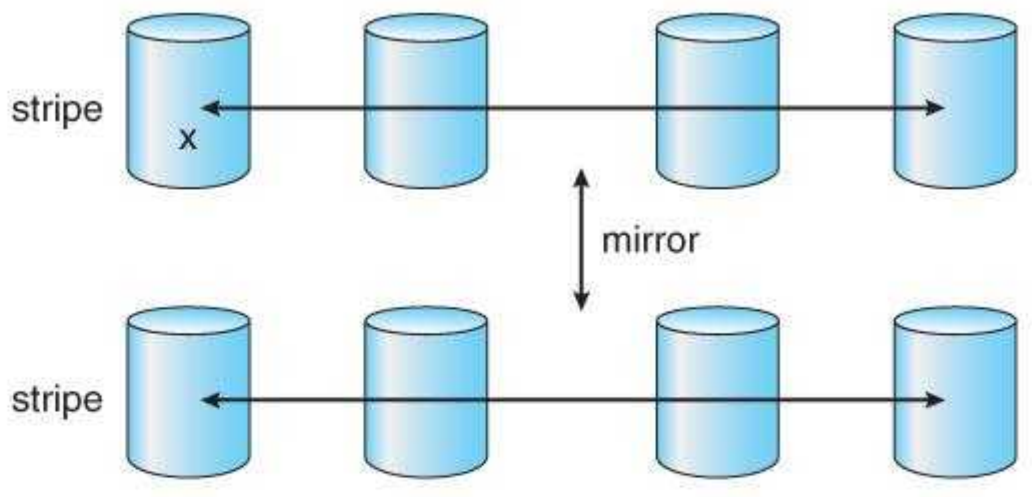
If a single disk fails, it wipes out the entire stripe set, but the system can keep on functioning using the remaining set.

However if a second disk from the other stripe set now fails, then the entire system is lost, as a result of two disk failures.

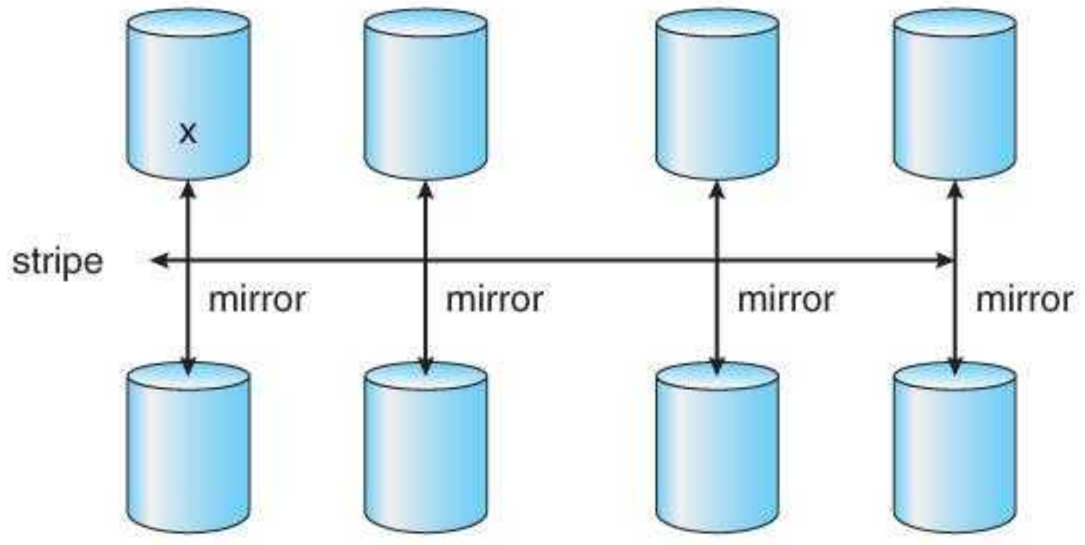
In diagram (b), the same 8 disks are divided into four sets of two, each of which is mirrored, and then the file system is striped across the four sets of mirrored disks.

If a single disk fails, then that mirror set is reduced to a single disk, but the system rolls on, and the other three mirror sets continue mirroring.

Now if a second disk fails, (that is not the mirror of the already failed disk), then another one of the mirror sets is reduced to a single disk, but the system can continue without data loss.



a) RAID 0 + 1 with a single disk failure.



b) RAID 1 + 0 with a single disk failure.

Figure - RAID 0 + 1 and 1 + 0

Selecting a RAID Level

- Trade-offs in selecting the optimal RAID level for a particular application include cost, volume of data, need for reliability, need for performance, and rebuild time, the latter of which can affect the likelihood that a second disk will fail while the first failed disk is being rebuilt.
- Other decisions include how many disks are involved in a RAID set and how many disks to protect with a single parity bit. More disks in the set increases performance but increases cost. Protecting more disks per parity bit saves cost,

but increases the likelihood that a second disk will fail before the first bad disk is repaired.

Extensions

- RAID concepts have been extended to tape drives (e.g. striping tapes for faster backups or parity checking tapes for reliability), and for broadcasting of data.

Problems with RAID

- RAID protects against physical errors, but not against any number of bugs or other errors that could write erroneous data.
- ZFS adds an extra level of protection by including data block checksums in all inodes along with the pointers to the data blocks. If data are mirrored and one copy has the correct checksum and the other does not, then the data with the bad checksum will be replaced with a copy of the data with the good checksum. This increases reliability greatly over RAID alone, at a cost of a performance hit that is acceptable because ZFS is so fast to begin with.

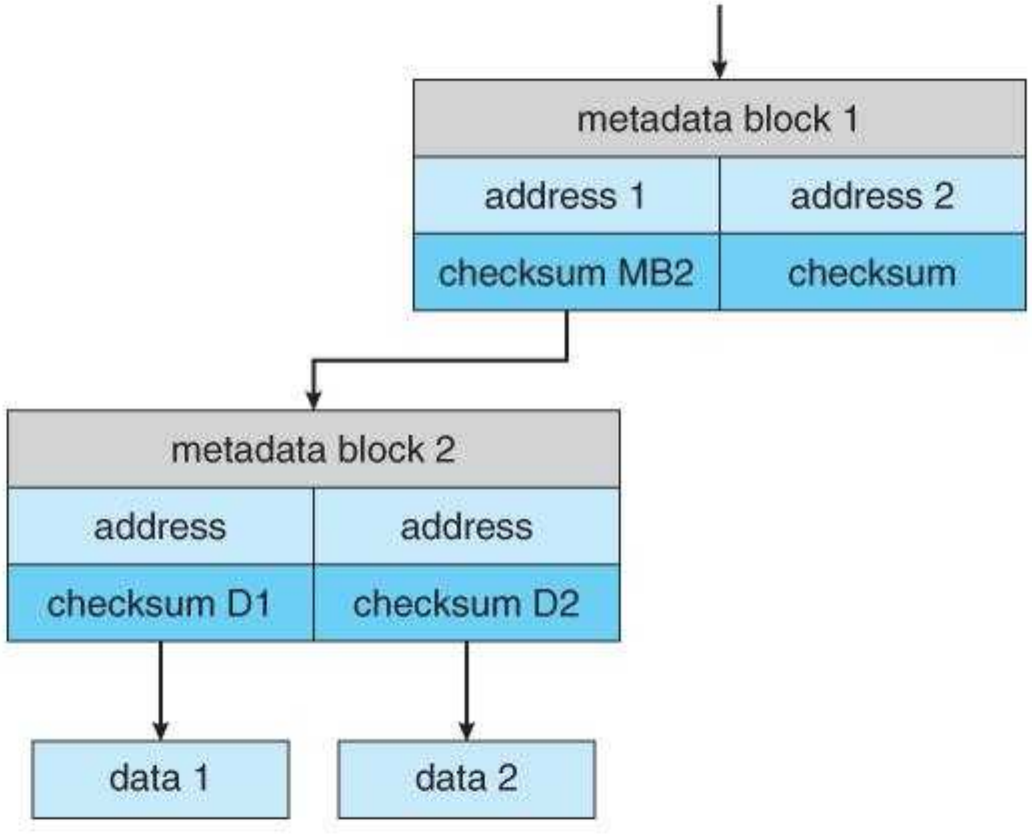


Figure 10.13 - ZFS checksums all metadata and data.

- Another problem with traditional filesystems is that the sizes are fixed, and relatively difficult to change. Where RAID sets are involved it becomes even harder to adjust filesystem sizes, because a filesystem cannot span across multiple filesystems.

- ZFS solves these problems by pooling RAID sets, and by dynamically allocating space to filesystems as needed. Filesystem sizes can be limited by quotas, and space can also be reserved to guarantee that a filesystem will be able to grow later, but these parameters can be changed at any time by the filesystem's owner. Otherwise filesystems grow and shrink dynamically as needed.

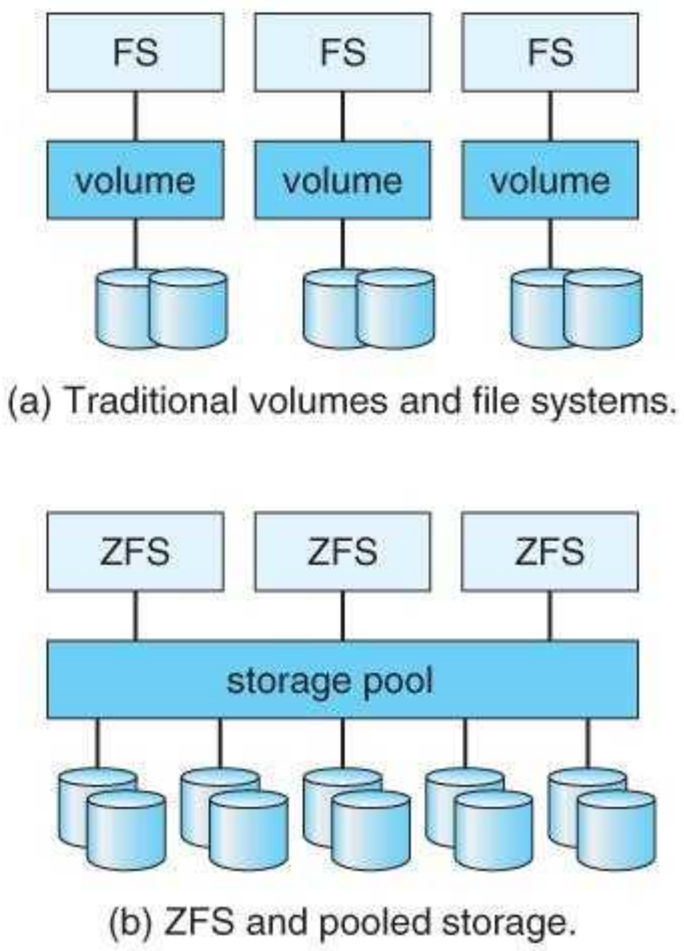

FS
FS
FS
volume
volume
volume
(a) Traditional volumes and file systems.
ZFS
ZFS
ZFS
storage pool
(b) ZFS and pooled storage.

Figure - (a) Traditional volumes and file systems. (b) a ZFS pool and file systems.