Unit 5: Input / Output Management

- **Device Management** is an important function of the operating system. It is responsible for managing all the hardware devices of the computer system.
- It may also include the management of the storage device as well as the management of all the input and output devices of the computer system.
- It is the responsibility of the operating system to keep track of the status of all the devices in the computer system
- An operating system manages the devices in a computer system with the help of device controllers and device drivers.
  - Each device in the computer system is equipped with the help of device controller.
  - The devices controllers are connected with each other through a system bus.
- Device management generally performs the following tasks:
  - ✓ Installing device and component-level drivers and related software
  - ✓ Configuring a device so it performs as expected using the bundled operating system, business/workflow software and/or with other hardware devices.
  - ✓ Implementing security measures and processes.
  - ✓ Monitoring the status of every device.

Classification of IO devices

✓ I/O devices can be roughly divided into two categories: **block devices** and **character devices.**

Block Devices

- A block device is one that stores information in fixed-size blocks, each one with its own address.
- Common block sizes range from 512 to 65,536 bytes.
- All transfers are in units of one or more entire (consecutive) blocks.
- The essential property of a block device is that it is possible to read or write each block independently of all the other ones.
- Examples: Hard disks, Blu-ray discs, USB sticks etc.

Character Devices

- A character device delivers or accepts a stream of characters, without regard to any block structure.
- It is not addressable and does not have any seek operation.
- Examples: Printers, network interfaces, mouse, joystick etc.

NOTE: This classification scheme is *not perfect.* Some devices do not fit in. ***Clocks***, for example, are not block addressable. Nor do they generate or accept character streams. All they do is cause interrupts at well-defined intervals. ***Memory-mapped screens*** do not fit the model well either. Nor do ***touch screens***, for that matter. Still, *the model of block and character devices is general enough that it can be used as a basis for making some of the operating system software dealing with I/O device independent*. The file system, for example, deals just with abstract block devices and leaves the device-dependent part to lower-level software.

Components of I/O Devices

- I/O Devices has two components:
  1. Mechanical Component
     - The mechanical component is device itself.
  2. Electronic Component
     - The electronic component is called Device Controller.

Device Controller

- Electronic component which controls the device.
- It may handle multiple devices.
- There may be more than one controller per mechanical component (example: hard drive).
- Controller's tasks are:
  - It converts serial bit stream to block of bytes
  - Perform error correction if necessary
  - Block of bytes is first assembled bit by bit in buffer inside the controller
  - After verification, the block has been declared to be error free, and then it can be copied to main memory.

Memory Mapped I/O

- Each controller has a few registers that are used for communicating with the CPU.
- By writing into these registers, the operating system can command the device to deliver data, accept data, switch itself on or off, or otherwise perform some action.
- By reading from these registers, the operating system can learn what the device's state is, whether it is prepared to accept a new command, and so on.
- In addition to the control registers, many devices have a data buffer that the operating system can read and write
- There are two ways to communicate with the control registers and the device buffers
    1. I/O Ports
    2. Memory Mapped I/O

NOTE: Another way is the hybrid of these two

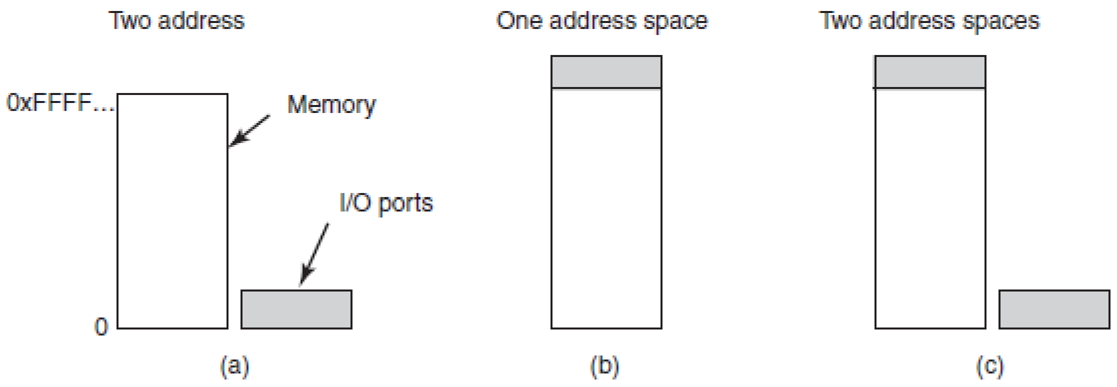


Figure: (a) Separate I/O and memory space. (b) Memory-mapped I/O. (c) Hybrid

- **In I/O Ports**
    - Each control register is assigned an I/O port number, an 8- or 16-bit integer.
    - The set of all the I/O ports form the I/O port space, which is protected so that ordinary user programs cannot access it (only the operating system can).
    - Using a special I/O instruction such as IN REG, PORT,the CPU can read in control register PORT and store the result in CPU register REG.
    - Similarly, using instruction like OUT PORT,REG the CPU can write the contents of REG to a control register

- In this scheme, the address spaces for memory and I/O are different

- **In Memory Mapped I/O**
  - Each control register is assigned a unique memory address to which no memory is assigned. This system is called memory mapped I/O.
  - Memory-mapped I/O uses the same address space to address both memory and I/O devices.
  - In most systems, the assigned addresses are at or near the top of the address space
  - In all cases, when the CPU wants to read a word, either from memory or from an I/O port, it puts the address it needs on the bus' address lines and then asserts a READ signal on a bus' control line.
  - A second signal line is used to tell whether I/O space or memory space is needed.
    - If it is memory space, the memory responds to the request.
    - If it is I/O space, the I/O device responds to the request.
    - If there is only memory space [as in figure (b)], every memory module and every I/O device compares the address lines to the range of addresses that it services.
    - If the address falls in its range, it responds to the request
  - Memory Mapped I/O is used for high-speed I/O devices.
  - During this, the OS also allocates a buffer in the system memory and informs the I/O device to send data to the processor using that buffer. Thus, an I/O device operates asynchronously with the processor and interrupts it when done.
  - The advantages of Memory Mapped I/O :
    - It can be implemented using high level language (C or C ++)
    - No special protection mechanism is needed to keep user processes from performing I/O.
    - Every instruction that can reference memory can also reference control registers
  - The disadvantages of Memory Mapped I/O :
    - Most computers nowadays have some form of caching of memory words. Caching a device control register would be disastrous.

  - If there is only one address space, then all memory modules and all I/O devices must examine all memory references to see which ones to respond to

Direct Memory Access (DMA) Operation

- **DMA** is a feature of computer systems that allow certain hardware subsystems (e.g. I/O Devices) to access main memory (RAM), independent of CPU.
- Direct memory access (DMA) is a mode of data transfer between the memory and I/O devices. This happens without the involvement of the processor.
- During Direct Memory Access, the CPU is idle and has no control over the memory buses. The DMA controller takes over the buses and directly manages data transfer between the memory unit and I/O devices.
- This feature is useful at any time that the CPU cannot keep up with the rate of data transfer, or when the CPU needs to perform work while waiting for a relatively slow I/O data transfer.
- Many hardware systems use DMA, including disk drive controllers, graphics cards, network cards and sound cards.

Operations of DMA transfer

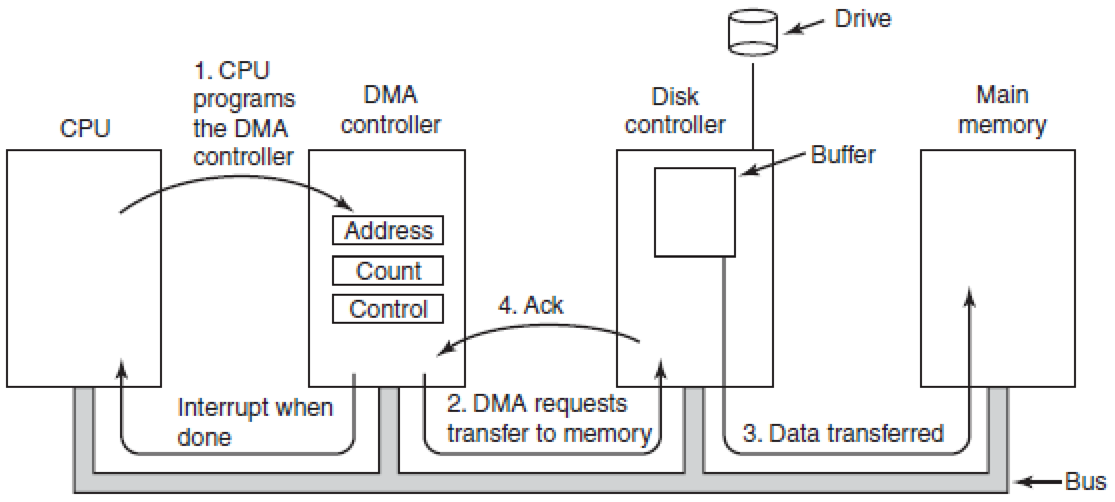


Figure: Operation of a DMA transfer

- **Step 1**: The CPU programs the DMA controller by setting its registers so it knows what to transfer where. It also issues a command to the disk controller telling it to read data from the disk into its internal buffer and verify the checksum. When valid data are in the disk controller's buffer, DMA can begin.
- **Step 2:** The DMA controller initiates the transfer by issuing a read request over the bus to the disk controller.
- **Step 3:** The write to memory is another standard bus cycle (step 3).
    - Data is transferred from disk controller to memory
- **Step 4:** When the write is complete, the disk controller sends an acknowledgement signal to the DMA controller, also over the bus
- The DMA controller then increments the memory address to use and decrements the byte count.
- If the byte count is still greater than 0, steps 2 through 4 are repeated until the count reaches 0.
- At that time, the DMA controller interrupts the CPU to let it know that the transfer is now complete.
- When the operating system starts up, it does not have to copy the disk block to memory; it is already there.

Interrupts

- The hardware mechanism that allows a device to notify the CPU is called interrupt.
- The interrupt is a signal emitted by hardware or software when a process or an event needs immediate attention.
- At the hardware level, interrupts work as follows.
    - When an I/O device has finished the work given to it, it causes an interrupt (assuming that interrupts have been enabled by the operating system).
    - It does this by asserting a signal on a bus line that it has been assigned.
    - This signal is detected by the interrupt controller chip on the parentboard, which then decides what to do.
    - If no other interrupts are pending, the interrupt controller handles the interrupt immediately.
    - However, if another interrupt is in progress, or another device has made a simultaneous request on a higher-priority interrupt request line on the bus the

device is just ignored for the moment. In this case it continues to assert an interrupt signal on the bus until it is serviced by the CPU.

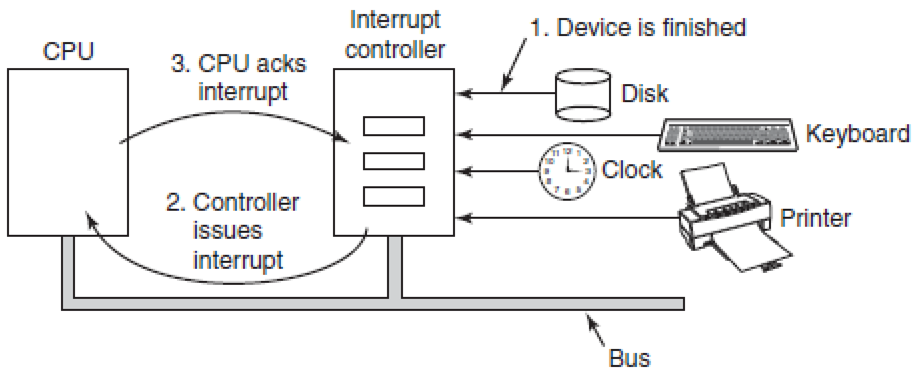


Figure: How an interrupt happens. The connections between the devices and the controller actually use interrupt lines on the bus rather than dedicated wires.

NOTE

- Two types of interrupts : *Hardware Interrupts* and *Software Interrupts*

Hardware Interrupts

- If a processor receives the interrupt request from an external I/O devices it is termed as a hardware interrupts.
- Hardware interrupts are further divided into *maskable* and *non-maskable* interrupt.
    - *Maskable Interrupt:* The hardware interrupt that can be ignored or delayed for some time if the processor is executing a program with higher priority are termed as maskable interrupts.
    - *Non-Maskable Interrupt:* The hardware interrupts that can neither be ignored nor delayed and must immediately be serviced by the processor are termed as non-maskeable interrupts.

Software Interrupts

- The software interrupts are the interrupts that occur when a condition is met or a system call occurs.
- The software interrupt is a type of interrupt caused by instructions in the program.

I/O Handling

Goals of IO Software

- **Device independence**
  - A key concept in the design of I/O software is known as device independence.
  - What it means is that we should be able to write programs that can access any I/O device without having to specify the device in advance.
  - For example, a program that reads a file as input should be able to read a file on a hard disk, a DVD, or on a USB stick without having to be modified for each different device
- **Uniform naming**
  - Closely related to device independence is the goal of uniform naming
  - The name of a file or a device should simply be a string or an integer and not depend on the device in any way.
  - All files and devices are addressed the same way: by a path name.
- **Error handling**
  - Another important issue for I/O software is error handling.
  - In general, errors should be handled as close to the hardware as possible.
  - If the controller discovers a read error, it should try to correct the error itself if it can. If it cannot, then the device driver should handle it.
  - Only if the lower layers are not able to deal with the problem should the upper layers be told about it.
  - In many cases, error recovery can be done transparently at a low level without the upper levels even knowing about the error.
- **Synchronous (blocking) vs. asynchronous (interrupt-driven) transfers**
  - Most physical I/O is asynchronous—the CPU starts the transfer and goes off to do something else until the interrupt arrives.
  - User programs are much easier to write if the I/O operations are blocking—after a read system call the program is automatically suspended until the data are available in the buffer.
  - It is up to the operating system to make operations that are actually interrupt-driven look blocking to the user programs.

- — However, some very high-performance applications need to control all the details of the I/O, so some operating systems make asynchronous I/O available to them.

- **Buffering**
  - — Another issue for the I/O software is buffering.
  - — Often data that come off a device cannot be stored directly in their final destination.
    - ▪ For example, when a packet comes in off the network, the operating system does not know where to put it until it has stored the packet somewhere and examined it.
  - — Also, some devices have severe real-time constraints (for example, digital audio devices), so the data must be put into an output buffer in advance to decouple the rate at which the buffer is filled from the rate at which it is emptied, in order to avoid buffer underruns.
  - — Buffering involves considerable copying and often has a major impact on I/O performance.
- **Sharable vs. dedicated devices**
  - — Some I/O devices, such as disks, can be used by many users at the same time. (Sharable)
    - ▪ No problems are caused by multiple users having open files on the same disk at the same time.
  - — Other devices, such as printers, have to be dedicated to a single user until that user is finished. Then another user can have the printer.
    - ▪ Having two or more users writing characters intermixed at random to the same page will definitely not work.
  - — Introducing dedicated (unshared) devices also introduces a variety of problems, such as deadlocks.
  - — Again, the operating system must be able to handle both shared and dedicated devices in a way that avoids problems

Handling I/O

- There are three fundamentally different ways that I/O can be performed.
  1. Programmed I/O

2. Interrupt-driven I/O
3. I/O using DMA

Programmed I/O

- In this technique, the processor executes a program giving direct control of I/O operations. Processor issues a command to the I/O module and waits for the operation to complete. Also, the processor keeps checking the I/O module status until it finds the completion of the operation.
- Data transfer through this mode requires constant monitoring of the peripheral device by the CPU and also monitor the possibility of new transfer once the transfer has been initiated. Thus CPU stays in a loop until the I/O device indicates that it is ready for data transfer. Thus programmed I/O is a time consuming process that keeps the processor busy needlessly and leads to wastage of the CPU cycles.
    - ☞ Simplest method
    - ☞ CPU does all the work
    - ☞ Data is exchanged between the processor and I/O Module
    - ☞ The processor executes a program that gives it direct control of the I/O operation
    - ☞ Polling or Busy Waiting mechanism is followed
- The simplest form of I/O is to have the CPU do all the work. This method is called **programmed I/O**.
- It is simplest to illustrate how programmed I/O works by means of an example.
    - — Consider a user process that wants to print the eight-character string ‘‘ABCDEFGH’’ on the printer via a serial interface. (Go through book for explanation)

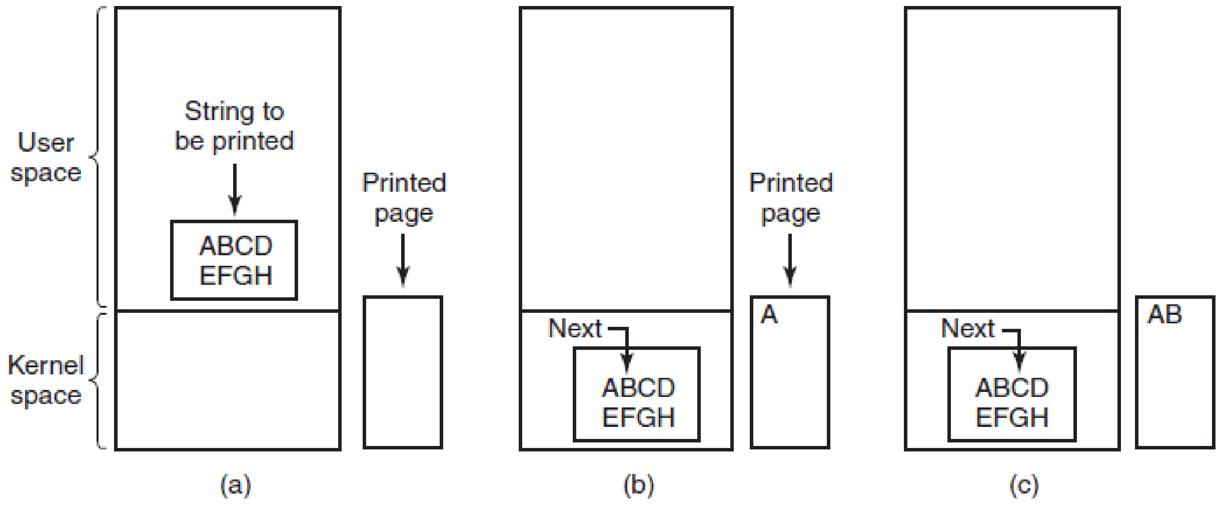


Figure: Steps in printing a string

The actions followed by the operating system are briefly summarized in figure (pseudocode) below.

```
copy_from_user(buffer, p, count);                /* p is the kernel buffer */
for (i = 0; i < count; i++) {                    /* loop on every character */
        while (*printer_status_reg != READY) ;   /* loop until ready */
        *printer_data_register = p[i];           /* output one character */
}
return_to_user( );
```

Figure: Writing a string to the printer using programmed I/O.

- Programmed I/O is simple but has the disadvantage of tying up the CPU full time until all the I/O is done.
- If the time to ‘‘print’’ a character is very short (because all the printer is doing is copying the new character to an internal buffer), then busy waiting is fine.
- Also, in an embedded system, where the CPU has nothing else to do, busy waiting is fine.
- However, in more complex systems, where the CPU has other work to do, busy waiting is inefficient. A better I/O method is needed.

Advantages of Programmed I/O

- Simple to implement
- Suitable for embedded devices, where CPU has nothing else to do
- Little hardware support

Disadvantages of Programmed I/O

- Not suitable for complex machines
- Busy-waiting solution hence inefficient
- Ties up CPU for long period with no useful work

Interrupt Driven I/O

- In Programmed I/O, CPU is kept busy unnecessarily. This situation can very well be avoided by using an interrupt driven method for data transfer..
- The way to allow the CPU to do something else while waiting for the I/O module to become ready is to use interrupts.
- Here, the Interrupt system is used so that the CPU does not have to watch and wait for the I/O module.
  - ☞ The CPU issues the command to the I/O module
  - ☞ The CPU then continues with what it was doing
  - ☞ The I/O module, issues the command to the I/O device and waits for the I/O to complete
  - ☞ Upon completion of one byte input or output, the I/O module sends an interrupt signal to the CPU
  - ☞ The CPU finishes what it was doing, then handles the interrupt
    - — This will involve moving the resulting data to its proper location
  - ☞ Once done with the interrupt, the CPU resumes execution of the program
- This is much more efficient than Programmed I/O as the CPU is not waiting during the (time-consuming) I/O process.
- Example: printing a string (as in above case)

```
copy_from_user(buffer, p, count);
enable_interrupts( );
while (*printer_status_reg != READY) ;
*printer_data_register = p[0];
scheduler( );
```

(a)

```
if (count == 0) {
    unblock_user( );
} else {
    *printer_data_register = p[i];
    count = count – 1;
     i = i + 1;
}
acknowledge_interrupt( );
return_from_interrupt( );
```

(b)

Figure: Writing a string to the printer using interrupt-driven I/O. (a) Code executed at the time the print system call is made. (b) Interrupt service procedure for the printer.

Advantages

- CPU does not have to watch and wait for the I/O module
- Fast and efficient

Disadvantage

- The processor must suspend its work and later resume it. If there are many devices, each can issue an interrupt and the processor must be able to attend each of these, based on some priority
- Overhead of interrupt handling

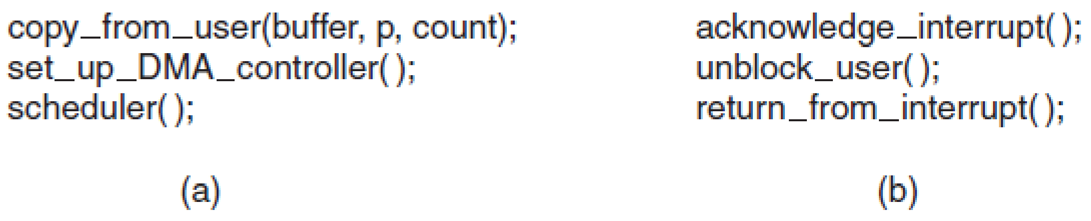
Programmed I/O vs Interrupt Driven I/O

Programmed I/O	**Interrupt Driven I/O**
Data transfer is initiated by the means of instructions stored in the computer program. Whenever there is a request for I/O transfer the instructions are executed from the program.	The I/O transfer is initiated by the interrupt command issued to the CPU.
In programmed I/O. processor has to check each I/O device in sequence and in effect 'ask' each one if it needs communication with the processor. This checking is achieved by continuous polling cycle and hence processor cannot execute other instructions in sequence.	External asynchronous input is used to tell the processor that 110 device needs its service and hence processor does not have to check whether I/O device needs it service or not.
During polling, processor is busy. So it decrements the system throughput.	In interrupt driven I/O, the processor is allowed to execute its instructions in sequence and only stop to service I/O device when it is told to do so by the device itself. This increases system throughput.
It is implemented without interrupt hardware support	It is implemented using interrupt hardware support
It does not depend on interrupt status	Interrupt must be enabled to process interrupt driven IO
It does not need initialization of stack	It needs initialization of stack

IO using DMA

- In essence, DMA is programmed I/O, only with the DMA controller doing all the work, instead of the main CPU. This strategy requires special hardware (the DMA controller) but frees up the CPU during the I/O to do other work
- Differ from Programmed I/O and Interrupt-Driven I/O, Direct Memory Access is a technique for transferring data within main memory and external device without passing it through the CPU.
- DMA allows an I/O module to communicate with memory so that CPU does not have to be interrupted for each data movement.
- DMA acts as interface between I/O device and memory
- DMA is a way to improve processor activity and I/O transfer rate by taking-over the job of transferring data from processor, and letting the processor to do other tasks

- It is more efficient to use DMA method when large volume of data has to be transferred.
- For DMA to be implemented, processor has to share its' system bus with the DMA module. Therefore, the DMA module must use the bus only when the processor does not need it, or it must force the processor to suspend operation temporarily. The latter technique is more common to be used and it is referred to as **cycle stealing**.
    - ☞ Processor issues command to DMA module, by sending necessary information to DMA module.
    - ☞ Processor does other work.
    - ☞ DMA acquire control on the system, and transfers data to and from within memory and I/O device.
    - ☞ DMA sends a signal to processor when the transfer is complete, system control is return to processor.
- Example: Printing string (as in above cases)



```
copy_from_user(buffer, p, count);
set_up_DMA_controller( );
scheduler( );

(a)
```

```
acknowledge_interrupt( );
unblock_user( );
return_from_interrupt( );

(b)
```

Figure: Printing a string using DMA. (a) Code executed when the print system call is made. (b) Interrupt-service procedure.

— The big win with DMA is reducing the number of interrupts from one per character to one per buffer printed.

Advantages

- Allows I/O device to read from/write to memory without going through the CPU
- Allows for faster processing since the processor can be working on something else while the peripheral can be populating memory

Disadvantages

- Requires a DMA controller to carry out the operation, which increases the cost of the system

I/O Software Layers

- I/O software is typically organized in four layers, as shown in figure below.
- Each layer has a well-defined function to perform and a well-defined interface to the adjacent layers.
- The functionality and interfaces differ from system to system.

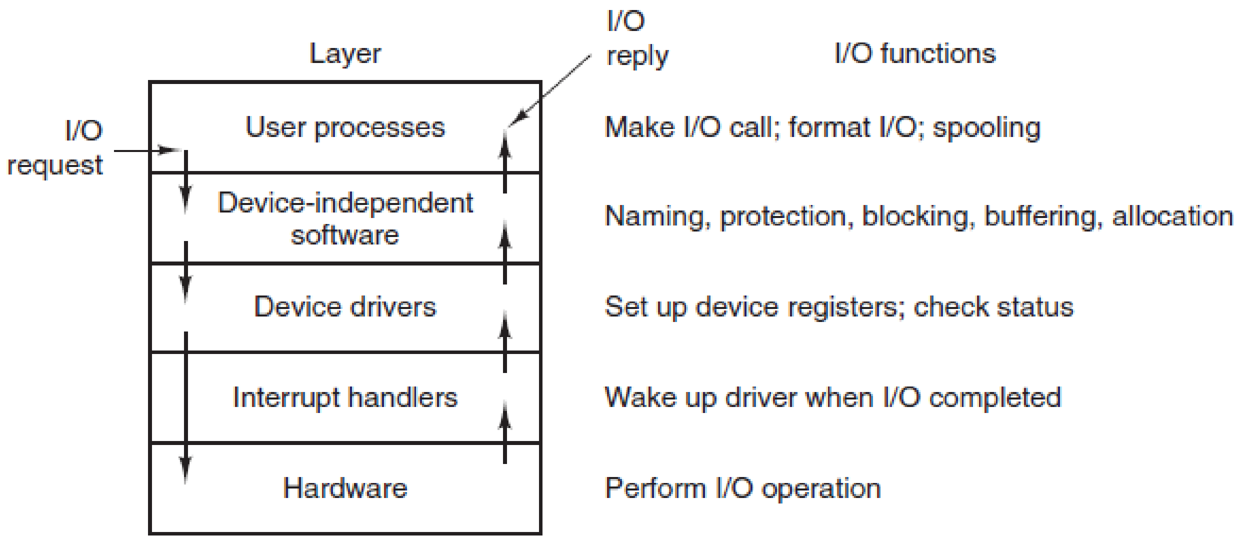

Layer
I/O reply
I/O functions
I/O request
User processes
Device-independent software
Device drivers
Interrupt handlers
Hardware
Make I/O call; format I/O; spooling
Naming, protection, blocking, buffering, allocation
Set up device registers; check status
Wake up driver when I/O completed
Perform I/O operation

Figure: Layers of the I/O system and the main functions of each layer

Interrupt Handlers

- Interrupt Handler is a program that runs when an interrupt is generated by hardware or software. The interrupt handler is also known as Interrupt Service Routine (ISR). ISR handles the request and sends it to the CPU. When the ISR is complete, the process gets resumed.
- When the interrupt happens, the interrupt procedure does whatever it has to in order to handle the interrupt. Then it can unblock the driver that was waiting for it.
- The interrupt mechanism accepts an address ─ a number that selects a specific interrupt handling routine/function from a small set. In most architectures, this address is an offset stored in a table called the interrupt vector table. This vector contains the memory addresses of specialized interrupt handlers.

- Main Steps in interrupt handling (system dependent, so may not be same for all)
    1. Save any registers (including the PSW) that have not already been saved by the interrupt hardware.

2. Set up a context for the interrupt-service procedure. Doing this may involve setting up the TLB, MMU and a page table.
3. Set up a stack for the interrupt service-procedure.
4. Acknowledge the interrupt controller. If there is no centralized interrupt controller, reenable interrupts.
5. Copy the registers from where they were saved (possibly some stack) to the process table.
6. Run the interrupt-service procedure. It will extract information from the interrupting device controller’s registers.
7. Choose which process to run next. If the interrupt has caused some high-priority process that was blocked to become ready, it may be chosen to run now.
8. Set up the MMU context for the process to run next. Some TLB setup may also be needed.
9. Load the new process’ registers, including its PSW.
10. Start running the new process.

Device drivers

- Device drivers are software modules that can be plugged into an OS to handle a particular device.
- Device drivers encapsulate device-dependent code and implement a standard interface in such a way that code contains device-specific register reads/writes
- Each I/O device attached to a computer needs some device-specific code for controlling it. This code, called the **device driver**, is generally written by the device’s manufacturer and delivered along with the device.
- Since each operating system needs its own drivers, device manufacturers commonly supply drivers for several popular operating systems.
- Each device driver normally handles one device type, or at most, one class of closely related devices.
- A device driver performs the following jobs −
    - To accept request from the device independent software above to it.
    - Interact with the device controller to take and give I/O and perform required error handling
    - Making sure that the request is executed successfully

- Since the designers of every operating system know that pieces of code (drivers) written by outsiders will be installed in it, it needs to have an architecture that allows such installation. This means having a well-defined model of what a driver does and how it interacts with the rest of the operating system.
- Device drivers are normally positioned below the rest of the operating system, as is illustrated in figure below

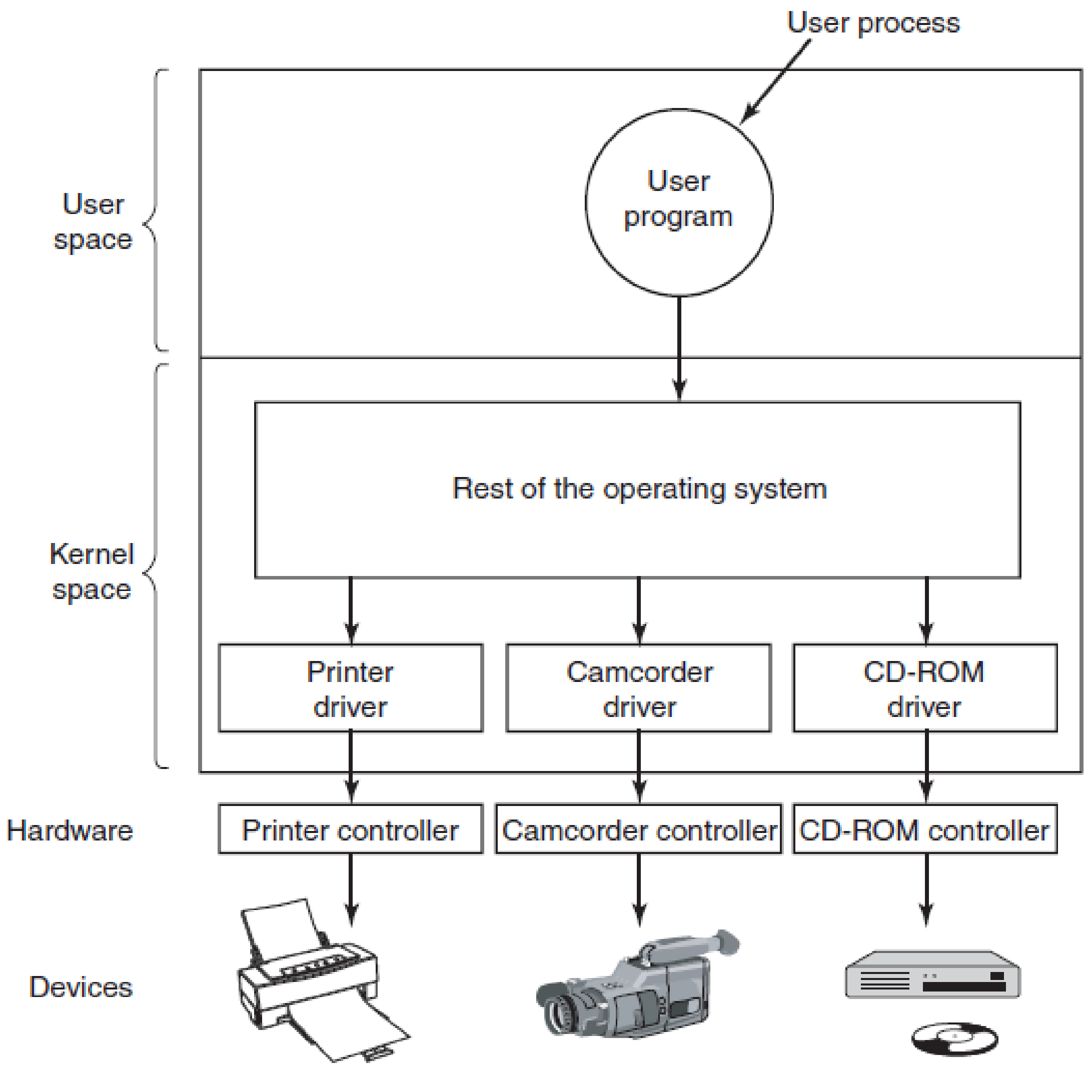


Figure: Logical positioning of device drivers. In reality all communication between drivers and device controllers goes over the bus.

Device-Independent I/O Software

- Although some of the I/O software is device specific, other parts of it are device independent.
- The basic function of the device-independent software is to perform the I/O functions that are common to all devices and to provide a uniform interface to the user-level software. Though it is difficult to write completely device independent software but we can write some modules which are common among all the devices
- Following is a list of functions of device-independent I/O Software –
    - ☑ Uniform interfacing for device drivers
    - ☑ Buffering
    - ☑ Error reporting
    - ☑ Allocating and releasing dedicated devices
    - ☑ Providing a device-independent block size

User-Space I/O Software

- These are the libraries which provide richer and simplified interface to access the functionality of the kernel or ultimately interactive with the device drivers.
- Most of the user-level I/O software consists of library procedures with some exception like spooling system which is a way of dealing with dedicated I/O devices in a multiprogramming system.
- I/O Libraries (e.g., stdio) are in user-space to provide an interface to the OS resident device-independent I/O SW. For example putchar(), getchar(), printf() and scanf() are example of user level I/O library stdio available in C programming.

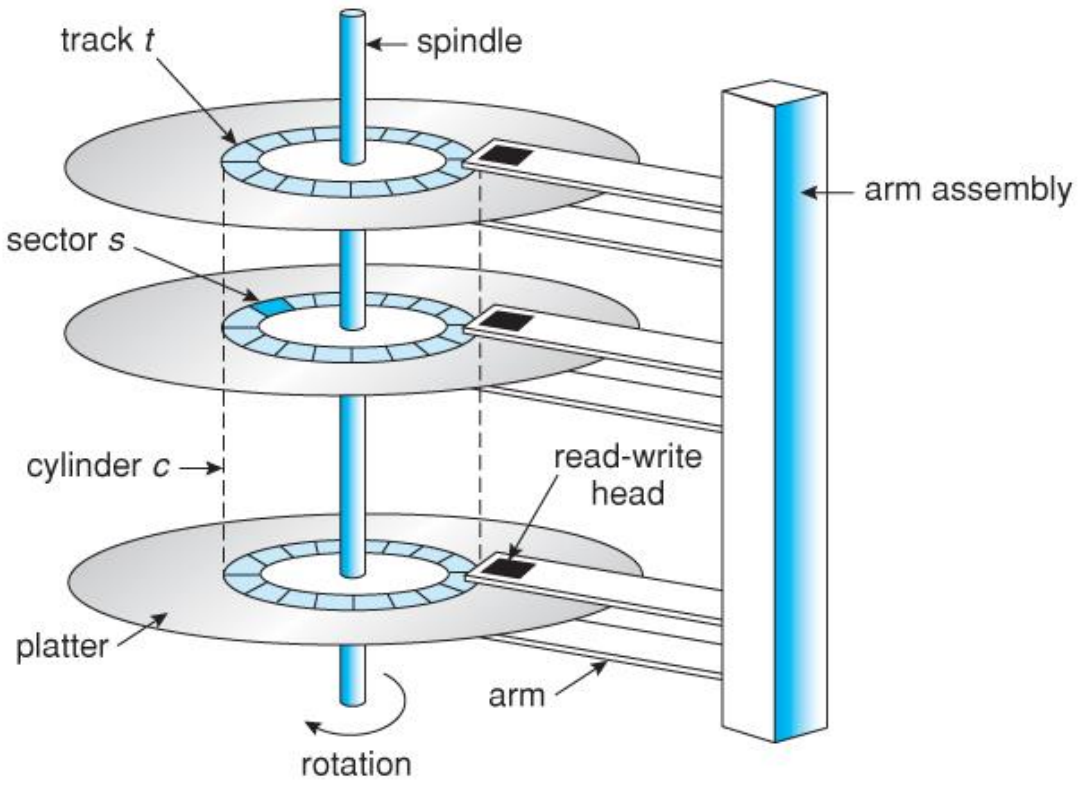
Disk Management

- Disks come in a variety of types. The most common ones are the magnetic hard disks. They are characterized by the fact that reads and writes are equally fast, which makes them suitable as secondary memory (paging, file systems, etc.). Arrays of these disks are sometimes used to provide highly reliable storage.
- For distribution of programs, data, and movies, optical disks (DVDs and Blu-ray) are also important.
- Solid-state disks are increasingly popular as they are fast and do not contain moving parts.

▪

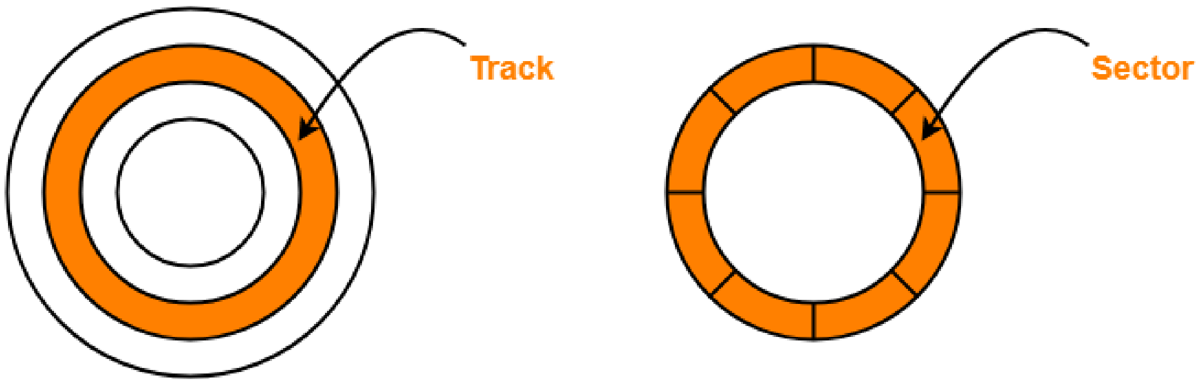
Disk Structure

- We consider hard disk (a magnetic disk)as an example.



- Hard disks are organized as a concentric stack of disks. An individual disk is referred to as a **platter.**
  - Each platter consists of **two surfaces**: a *lower* and an *upper* **surface**.
- Each surface of a platter consists of a fixed number of concentric circles called as **tracks**
  - These tracks are circular areas on the surface of a platter that decrease in circumference as we move towards the center of the platter.
  - Data is first written to the outermost track.
- The tracks are further divided into **sectors** which are the smallest divisions in the disk.
  - Sectors divide track sections and store data.

- A **cylinder** is formed by combining the tracks at a given radius of a disk pack.
- The platters within the hard disk are connected by a **spindle** that runs through the middle of the platters.
  - The spindle moves in a unidirectional manner along its axis (either clockwise or counterclockwise).
  - The movement of the spindle causes the platters to rotate as well.
- Each surface on a platter contains a **read/write head** that is used to read or write data onto the disk.
  - The read/write head can move back and forth along the surface of a platter
  - Each platter has two surfaces- top and bottom and both the surfaces are used to store the data.
  - Each surface has its own read / write head.
- Head has to reach at a particular track and then wait for the rotation of the platter.
  - The rotation causes the required sector of the track to come under the head.
- Read/write heads are in turn connected to a single **arm assembly**.



NOTE : All the tracks of a disk have the same storage capacity.

Disk Performance Parameters

- The time taken by the disk to complete an I/O request is called as **disk service time** or **disk access time**
- Components that contribute to the service time are:
  - Seek time
  - Rotational latency
  - Data transfer rate
  - Controller overhead
  - Queuing delay

Seek Time

- The time taken by the read / write head to reach the desired track is called as **seek time**.
- It is the component which contributes the largest percentage of the disk service time.
- The lower the seek time, the faster the I/O operation

Rotational Latency-

- The time taken by the desired sector to come under the read / write head is called as ***rotational latency.***
- It depends on the rotation speed of the spindle.

Average rotational latency = 1 / 2 x Time taken for full rotation

Data Transfer Rate

- The amount of data that passes under the read / write head in a given amount of time is called as ***data transfer rate***.
- The time taken to transfer the data is called as ***transfer time.***
- Data Transfer Rate depends on the following factors-
  - Number of bytes to be transferred
  - Rotation speed of the disk
  - Density of the track
  - Speed of the electronics that connects the disk to the computer

Transfer Time = Size of data be transferred / Data Transfer Rate

Data Transfer Rate = Number of surfaces × Number of sectors per track × Number of bytes per sector × Number of rotations in one second

Or

Data Transfer Rate = Number of heads × Capacity of one track × Number of rotations in one second

Controller Overhead

- The overhead imposed by the disk controller is called as ***controller overhead.***
- Disk controller is a device that manages the disk.

Queuing Delay

- The time spent waiting for the disk to become free is called as ***queuing delay.***

Some Formulas

Disk access time = Seek time + Rotational delay + Transfer time + Controller overhead + Queuing delay

Capacity of disk = Total number of surfaces ×Number of tracks per surface × Number of sectors per track × Number of bytes per sector

Storage (recording) density of a track = Capacity of the track / Circumference of the track

Capacity of a track = Recording density of the track × Circumference of the track

Or

Capacity of a track = Number of sectors per track × Number of bytes per sector

NOTE: ***Disk Response Time*** *is the average of time spent by each request waiting for the IO operation.*

Some Examples

Total number of tracks per surface = (Outer radius – Inner radius) / Inter track gap

1. Consider a disk pack with the following specifications- 16 surfaces, 128 tracks per surface, 256 sectors per track and 512 bytes per sector. Calculate the capacity of disk pack.

Solution:

Given,

Number of surfaces = 16

Number of tracks per surface = 128

Number of sectors per track = 256

Number of bytes per sector = 512 bytes

Capacity of disk pack

= Total number of surfaces x Number of tracks per surface x Number of sectors per track x Number of bytes per sector

= 16 x 128 x 256 x 512 bytes

= 228 bytes

= 256 MB

2. What is the average access time for transferring 512 bytes of data with the following specifications-

   Average seek time = 5 msec

   Disk rotation = 6000 RPM

   Data rate = 40 KB/sec

   Controller overhead = 0.1 msec

Solution:

Given,

Average seek time = 5 msec

Disk rotation = 6000 RPM

Data rate = 40 KB/sec

Controller overhead = 0.1 msec

Time Taken For One Full Rotation-

- 6000 rotations takes 1 minutes (i.e. 60 sec)
- 1 rotation takes 60/6000 seconds
  = 0.01 sec
  = 10 msec

Average rotational delay

= 1/2 x Time taken for one full rotation

= 1/2 x 10 msec

= 5 msec

Transfer time

= (512 bytes / 40 KB) sec

= (512 bytes / 40 * 1024 bytes)sec

= 0.0125 sec

= 12.5 msec

Now, Average access time

= Average seek time + Average rotational delay + Transfer time + Controller overhead + Queuing delay

= 5 msec + 5 msec + 12.5 msec + 0.1 msec + 0

= 22.6 msec

Exercise:

1. Consider a disk with a sector size of 512 bytes, 50 sector per track, 2000 tracks per surface, five double sided platters, and average seek time of 10 msec.
   a) What is capacity of disk in bytes?
   b) Compute the capacity of a track and a surface.
   c) If the disk platter rotate at 5400 rpm, then approximately what is the maximum rotational delay and average rotational delay. (Ans: 0.011 sec, 0.0055 sec)
2. Consider a disk has 16 surfaces, 128 tracks/surface, 256 sectors/ track, 512 bytes/sector. If the disk rotates at 3600 rpm, calculate data transfer rate. (Ans: 128MB/Sec)

Disk Scheduling Algorithm

☑ Disk scheduling is a technique used by the operating system to schedule multiple requests for accessing the disk.

☑ Disk scheduling is also known as I/O scheduling.

☑ The time required to read or write a disk block is determined mainly by following factors:

- ☞ Seek time (the time to move the arm to the proper cylinder).
- ☞ Rotational delay (how long for the proper sector to appear under the reading head).
- ☞ Actual data transfer time.
- ☑ For most disks, the seek time dominates the other times, so reducing the mean seek time can improve system performance substantially
- ☑ The purpose of disk scheduling algorithms is to reduce the total seek time
- ☑ 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. Some of them are :

- FCFS Algorithm
- SSTF Algorithm
- SCAN Algorithm
- C-SCAN Algorithm
- LOOK Algorithm
- C-LOOK Algorithm

FCFS (First-Come First-Served) Algorithm

- 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.
- As the name suggests, this algorithm entertains requests in the order they arrive in the disk queue.
- It is the simplest disk scheduling algorithm.

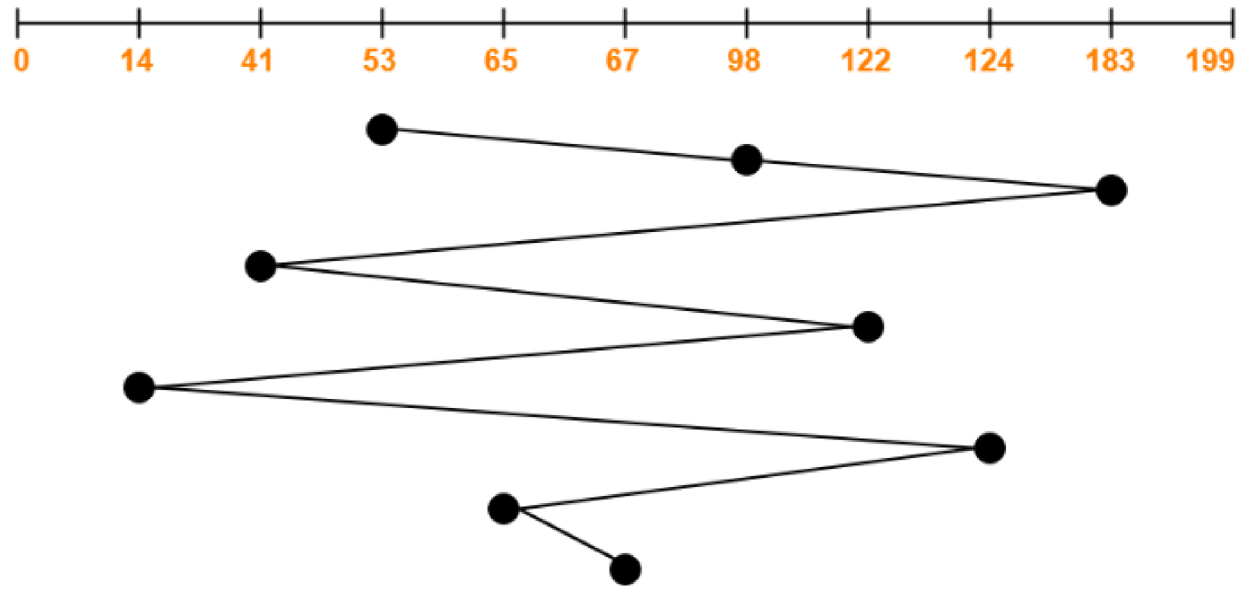
<u>Advantages-</u>

- It is simple, easy to understand and implement.
- No indefinite postponement
- Every request gets a fair chance

<u>Disadvantages-</u>

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

Example: Consider a disk queue with requests for I/O to blocks on cylinders 98, 183, 41, 122, 14, 124, 65, 67. The FCFS scheduling algorithm is used. The head is initially at cylinder number 53. The cylinders are numbered from 0 to 199. What is the total head movements incurred while servicing these requests (i.e. find total seek time if one movement takes 1 nano seconds)



Total head movements incurred while servicing these requests

= (98 – 53) + (183 – 98) + (183 – 41) + (122 – 41) + (122 – 14) + (124 – 14) + (124 – 65)

+ (67 – 65)

= 45 + 85 + 142 + 81 + 108 + 110 + 59 + 2

= 632

Exercise:

- Suppose the order of request is- (82,170,43,140,24,16,190) And current position of Read/Write head is : 50 (cylinders are numbered from 0 to 199) . What is the total seek time? (Ans: 642)

Shortest Seek Time First (SSTF) Algorithm

- ☑ 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.
- ☑ It allows the head to move to the closest track in the service queue.
- ☑ It breaks the tie in the direction of head movement.
- ☑ SSTF is certainly an improvement over FCFS as it decreases the average response time and increases the throughput of system.

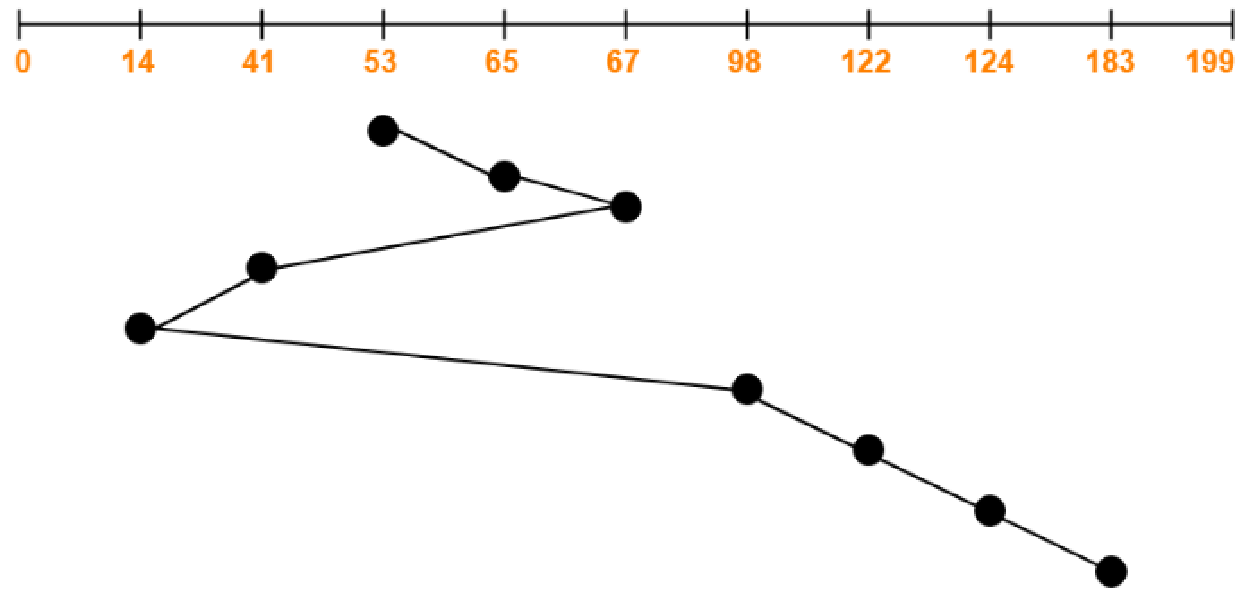
<u>Advantages</u>

- ☑ It reduces the total seek time as compared to FCFS.
- ☑ It provides increased throughput.
- ☑ It provides less average response time and waiting time.

<u>Disadvantages</u>

- ☑ Overhead to calculate seek time in advance
- ☑ It may cause starvation for some requests.
- ☑ Switching direction on the frequent basis slows the working of algorithm.

Example: Consider a disk queue with requests for I/O to blocks on cylinders 98, 183, 41, 122, 14, 124, 65, 67. The SSTF scheduling algorithm is used. The head is initially at cylinder number 53 moving towards larger cylinder numbers on its servicing pass. The cylinders are numbered from 0 to 199. Calculate the total head movement (in number of cylinders) incurred while servicing these requests.



Total head movements incurred while servicing these requests

= (65 – 53) + (67 – 65) + (67 – 41) + (41 – 14) + (98 – 14) + (122 – 98) + (124 – 122)

+ (183 – 124)

= 12 + 2 + 26 + 27 + 84 + 24 + 2 + 59

= 236

Exercise:

a) Consider a disk system with 100 cylinders. The requests to access the cylinders occur in following sequence- 4, 34, 10, 7, 19, 73, 2, 15, 6, 20

   Assuming that the head is currently at cylinder 50, what is the time taken to satisfy all requests if it takes 1 msec to move from one cylinder to adjacent one and shortest seek time first policy is used? (Answer :119 msec)

b) Consider the following disk request sequence for a disk with 100 tracks
   45, 21, 67, 90, 4, 89, 52, 61, 87, 25

   Head pointer starting at 50. Find the number of head movements in cylinders using SSTF scheduling. (Ans: 136)

SCAN Algorithm

- ☑ SCAN Algorithm is also called as Elevator Algorithm. This is because its working resembles the working of an elevator.
- ☑ As the name suggests, this algorithm scans all the cylinders of the disk back and forth.
- ☑ In SCAN algorithm the disk head 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.
  - ➢ Head starts from current track of the disk and move in a particular direction up to the ending cylinder, servicing all the requests in between.
  - ➢ After reaching the end, head reverses its direction and move towards another end (up to requested number) servicing all the requests in between.
  - ➢ The same process repeats.

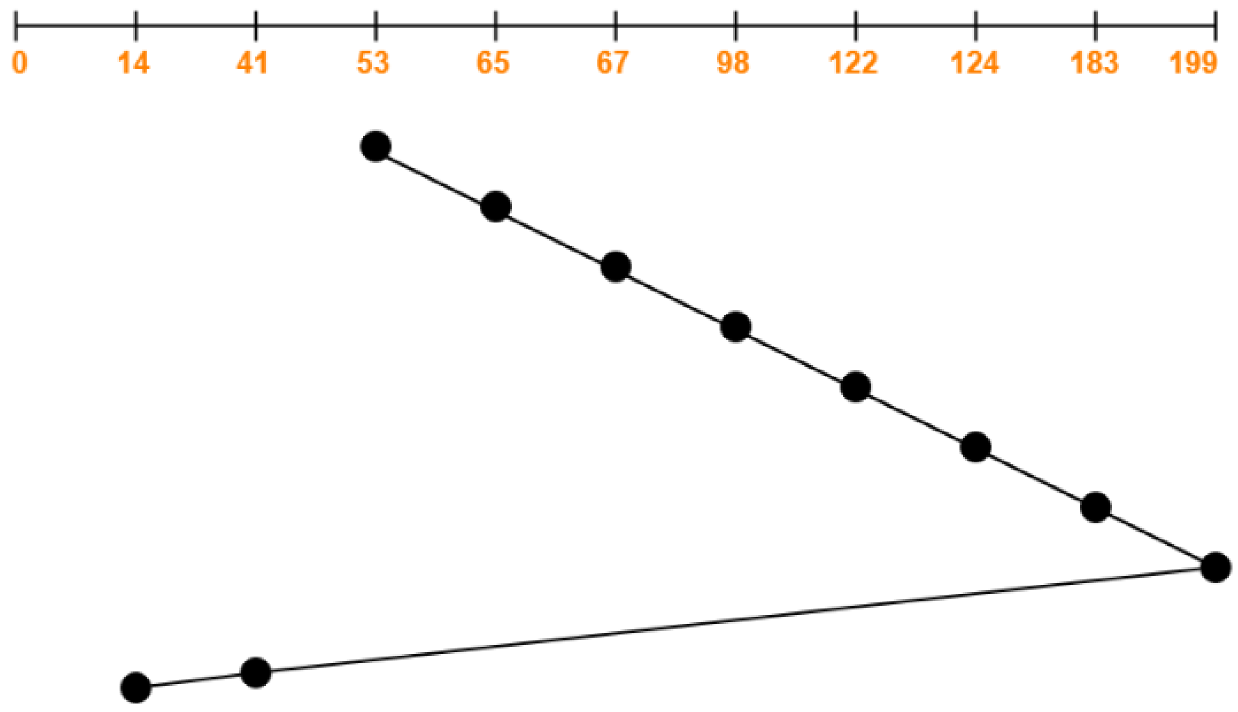
<u>Advantages-</u>

- ☑ It is simple, easy to understand and implement.
- ☑ It does not lead to starvation.
- ☑ It provides low variance in response time and waiting time.

<u>Disadvantages-</u>

- ☑ It causes long waiting time for the cylinders just visited by the head.
- ☑ It causes the head to move till the end of the disk even if there are no requests to be serviced.

Example: Consider a disk queue with requests for I/O to blocks on cylinders 98, 183, 41, 122, 14, 124, 65, 67. The SCAN scheduling algorithm is used. The head is initially at cylinder number 53 moving towards larger cylinder numbers on its servicing pass. The cylinders are numbered from 0 to 199. Compute total head movement (in number of cylinders) incurred while servicing these requests.



Total head movements incurred while servicing these requests

= (65 – 53) + (67 – 65) + (98 – 67) + (122 – 98) + (124 – 122) + (183 – 124) + (199 – 183) + (199 – 41) + (41 – 14)

= 12 + 2 + 31 + 24 + 2 + 59 + 16 + 158 + 27

= 331

Alternatively,

Total head movements incurred while servicing these requests

= (199 – 53) + (199 – 14)

= 146 + 185

= 331

Exercise: Suppose the requests to be addressed are-82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move "towards the larger value". Calculate the seek time using SCAN algorithm if one movement takes 1 nano sec. (Ans: 332 nano sec)

C-SCAN (Circular SCAN) Algorithm

- ☑ In C-SCAN algorithm, the head 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.

- ☑ Circular-SCAN Algorithm is an improved version of the SCAN Algorithm.
  - ☞ Head starts from one end of the disk (innermost/outermost cylinder) and move towards the other end servicing all the requests in between.
  - ☞ After reaching the other end, head reverses its direction.
  - ☞ It then returns to another end without servicing any request in between.
  - ☞ The same process repeats.

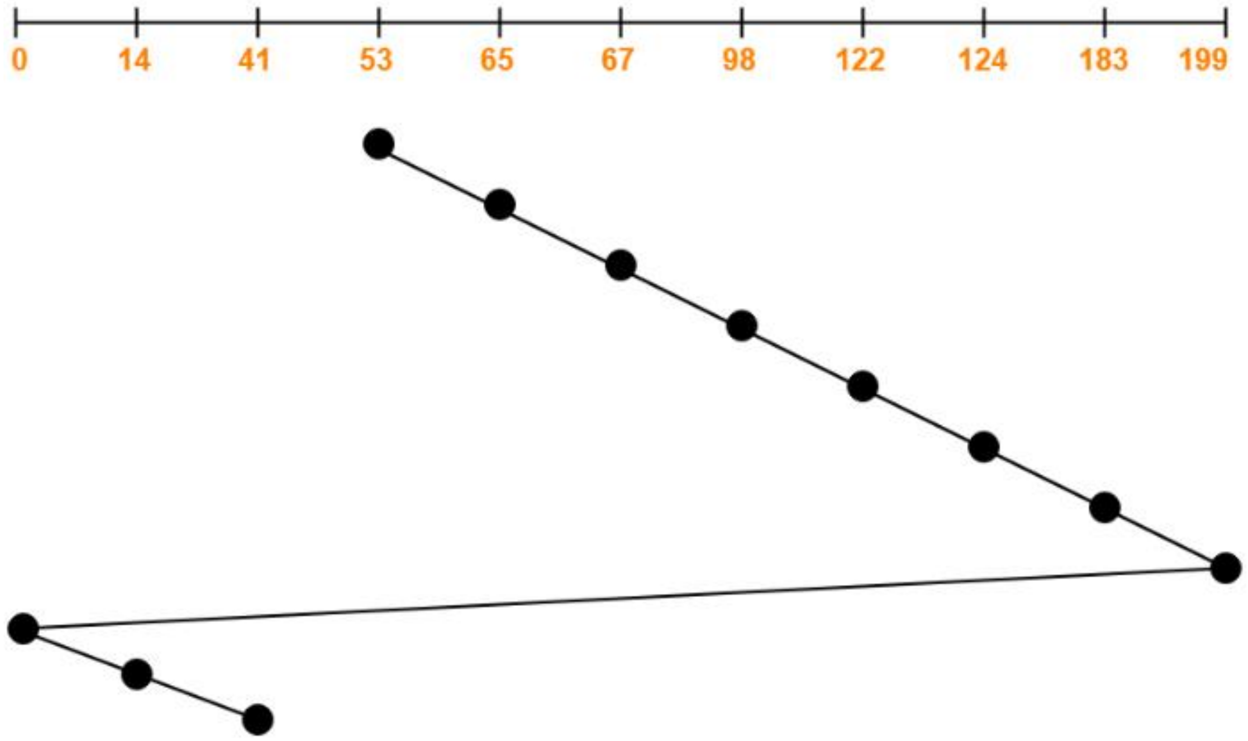
<u>Advantages-</u>

- ☑ The waiting time for the cylinders just visited by the head is reduced as compared to the SCAN Algorithm.
- ☑ It provides uniform waiting time.
- ☑ It provides better response time.

<u>Disadvantages-</u>

- ☑ It causes more seek movements as compared to SCAN Algorithm.
- ☑ It causes the head to move till the end of the disk even if there are no requests to be serviced.

Example: Consider a disk queue with requests for I/O to blocks on cylinders 98, 183, 41, 122, 14, 124, 65, 67. The C-SCAN scheduling algorithm is used. The head is initially at cylinder number 53 moving towards larger cylinder numbers on its servicing pass. The cylinders are numbered from 0 to 199. Calculate total head movement (in number of cylinders) incurred while servicing these requests.



Total head movements incurred while servicing these requests

= (65 – 53) + (67 – 65) + (98 – 67) + (122 – 98) + (124 – 122) + (183 – 124) +

(199 – 183) + (199 – 0) + (14 – 0) + (41 – 14)

= 12 + 2 + 31 + 24 + 2 + 59 + 16 + 199 + 14 + 27

= 386

Alternatively,

Total head movements incurred while servicing these requests

= (199 – 53) + (199 – 0) + (41 – 0)

= 146 + 199 + 41

= 386

Exercise: Consider, a disk contains 200 tracks (0-199) and the request queue contains track number 82, 170, 43, 140, 24, 16,190, respectively. The current position of R/W head is 50, and the direction is towards the larger value. Calculate the total number of cylinders moved by head using C-SCAN disk scheduling. (Ans: 391)

LOOK Algorithm

- ☑ 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
- ☑ LOOK Algorithm is an improved version of the SCAN Algorithm.
  - ☞ Head starts from the first request at one end of the disk and moves towards the last request (not the last cylinder) at the other end servicing all the requests in between.
  - ☞ After reaching the last request at the other end, head reverses its direction.
  - ☞ It then returns to the last request (up to requested number) at the opposite end servicing all the requests in between.
  - ☞ The same process repeats.
- ❖ The main difference between SCAN Algorithm and LOOK Algorithm is-
  - — SCAN Algorithm scans all the cylinders of the disk starting from one end to the other end even if there are no requests at the ends.
  - — LOOK Algorithm scans all the cylinders of the disk starting from the first request at one end to the last request at the other end.

<u>Advantages-</u>

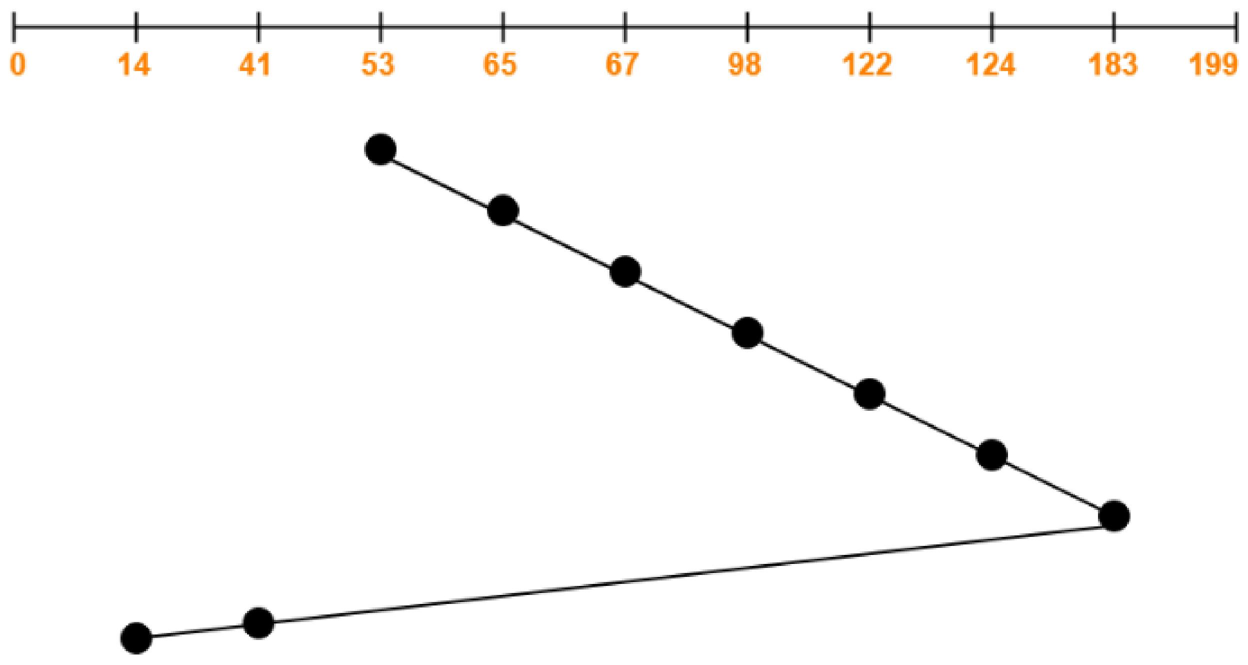
- — It does not causes the head to move till the ends of the disk when there are no requests to be serviced.
- — It provides better performance as compared to SCAN Algorithm.
- — It does not lead to starvation.
- — It provides low variance in response time and waiting time.

<u>Disadvantages-</u>

- — There is an overhead of finding the end requests.

— It causes long waiting time for the cylinders just visited by the head.

Example : Consider a disk queue with requests for I/O to blocks on cylinders 98, 183, 41, 122, 14, 124, 65, 67. The LOOK scheduling algorithm is used. The head is initially at cylinder number 53 moving towards larger cylinder numbers on its servicing pass. The cylinders are numbered from 0 to 199. Find the total head movement (in number of cylinders) incurred while servicing these requests.



Total head movements incurred while servicing these requests

= (65 – 53) + (67 – 65) + (98 – 67) + (122 – 98) + (124 – 122) + (183 – 124) + (183 – 41) + (41 – 14)

= 12 + 2 + 31 + 24 + 2 + 59 + 142 + 27

= 299

Alternatively,

Total head movements incurred while servicing these requests

= (183 – 53) + (183 – 14)

= 130 + 169

= 299

Exercise: 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. (Ans: 209)

C Look Algorithm

- ☑ 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.
- ☑ Circular-LOOK Algorithm is an improved version of the LOOK Algorithm.
    - ☞ Head starts from the first request at one end of the disk and moves towards the last request at the other end servicing all the requests in between.
    - ☞ After reaching the last request at the other end, head reverses its direction.
    - ☞ It then returns to the end request at that side without servicing any request in between.
    - ☞ The same process repeats.

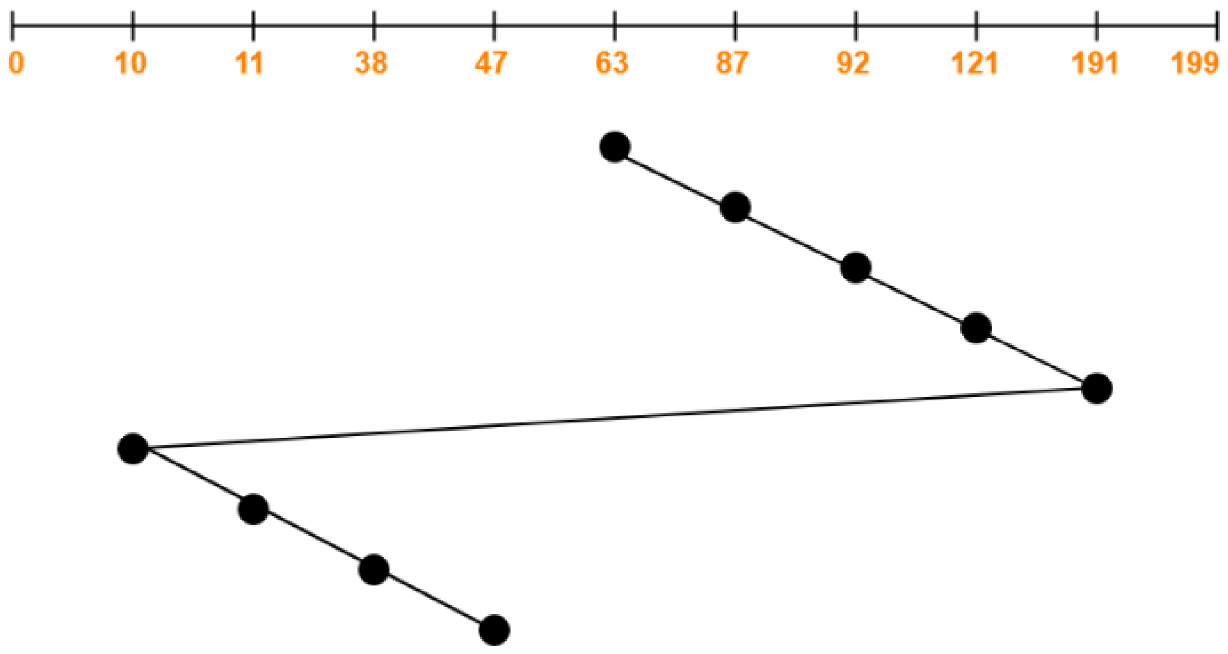
<u>Advantages-</u>

- ☑ It does not causes the head to move till the ends of the disk when there are no requests to be serviced.
- ☑ It reduces the waiting time for the cylinders just visited by the head.
- ☑ It provides better performance as compared to LOOK Algorithm.
- ☑ It does not lead to starvation.
- ☑ It provides low variance in response time and waiting time.

Disadvantages-

☑ There is an overhead of finding the end requests.

Example: Consider a disk queue with requests for I/O to blocks on cylinders 47, 38, 121, 191, 87, 11, 92, 10. The C-LOOK scheduling algorithm is used. The head is initially at cylinder number 63 moving towards larger cylinder numbers on its servicing pass. The cylinders are numbered from 0 to 199. Find the total head movement (in number of cylinders) incurred while servicing these requests



Total head movements incurred while servicing these requests

= (87 – 63) + (92 – 87) + (121 – 92) + (191 – 121) + (191 – 10) + (11 – 10) + (38 – 11) + (47 – 38)

= 24 + 5 + 29 + 70 + 181 + 1 + 27 + 9

= 346

Alternatively,

Total head movements incurred while servicing these requests

= (191 – 63) + (191 – 10) + (47 – 10)

= 128 + 181 + 37

= 346

Exercise:

a) 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. (Ans: 298)
b) Consider a disk queue with requests for I/O to blocks on cylinders 98, 183, 41, 122, 14, 124, 65, 67. The C-LOOK scheduling algorithm is used. The head is initially at cylinder number 53 moving towards larger cylinder numbers on its servicing pass. The cylinders are numbered from 0 to 199. Find the total head movement (in number of cylinders) incurred while servicing these requests. (Ans: 326)

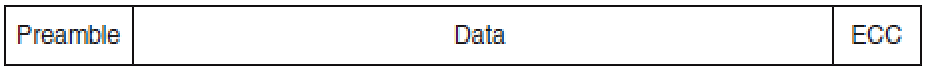
Disk Formatting

Disk formatting is a process of configuring data storage device such as hard disk drive, solid-state drive, floppy disk, or USB flash drive to use for the first time.

— During this procedure, any existing data on the device will be erased. The configuration process involves wiping all the data on a storage device like a hard drive, a solid-state drive, or a flash drive before installing an operating system.
— Before installing any OS, it is mandatory to format the disk drive to go through the process. Alternatively, we can wipe the disk in case of a storage space or malware issue. Formatting erases all the data on a disk.

- A hard disk consists of a stack of aluminum, alloy, or glass platters typically 3.5 inch in diameter (or 2.5 inch on notebook computers). On each platter is deposited a thin magnetizable metal oxide.
- After manufacturing, there is no information whatsoever on the disk.
- Before the disk can be used, each platter must receive a **low-level format** done by software.
    - The format consists of a series of concentric tracks
    - Each track contains some number of sectors

-
    - There is short gap between the sectors.
- The **format of a sector** is shown below:



-
    - The **preamble** starts with a certain bit pattern that allows the hardware to recognize the start of the sector.
        - It also contains the cylinder and sector numbers and some other information.
    - The size of the **data** portion is determined by the low level formatting program.
        - Most disks use 512-byte sectors.
    - The **ECC** (Error Correcting Code) field contains redundant information that can be used to recover from read errors.
        - The size and content of this field varies from manufacturer to manufacturer, depending on how much disk space the designer is willing to give up for higher reliability and how complex an ECC code the controller can handle.
        - A 16-byte ECC field is not unusual.
- Furthermore, all hard disks have some number of **spare sectors** allocated to be used to replace sectors with a manufacturing defect.
- The **disk formatting procedure involves three sub-processes.**
    1. **Low-level Formatting**
    2. **Partitioning**
    3. **High-level Formatting**

Low-level Formatting:

⇒ This process includes marking out cylinders and tracks for an empty hard disk, further dividing these tracks into multiple sectors using markers. This process completely wipes all data stored on a disk, making it unrecoverable. Manufacturers perform low-level formatting before they ship the hard disks.

Cylinder Skew

- The position of sector 0 on each track is offset from the previous track when the low-level format is laid down. This offset is called cylinder skew.
- It is done to improve performance.
- It allows the disk to read multiple tracks in one continuous operation without losing data

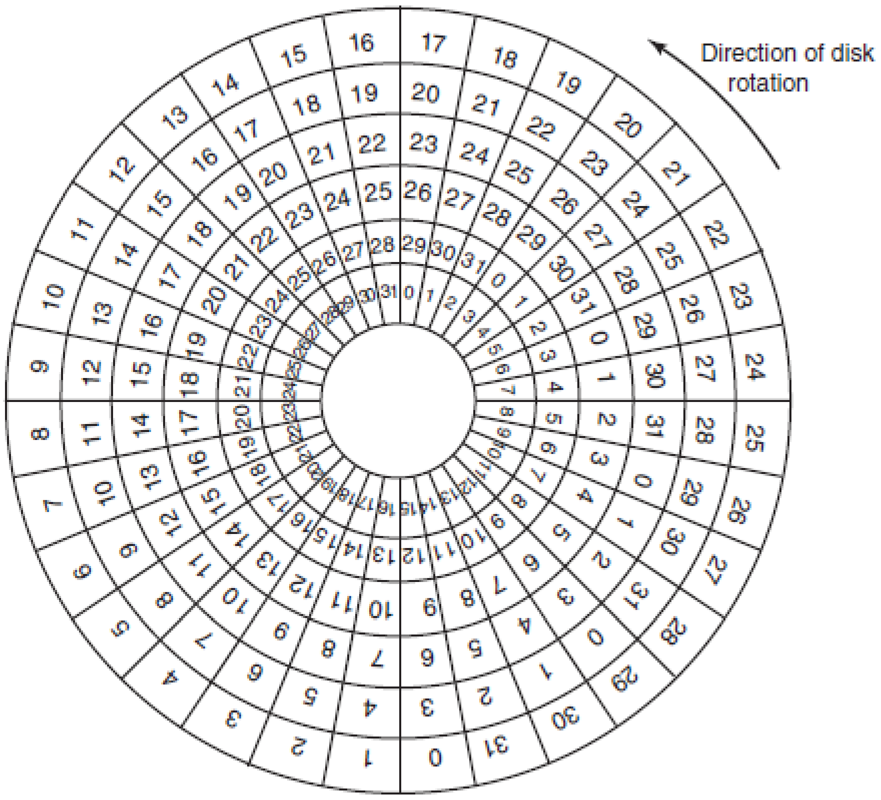
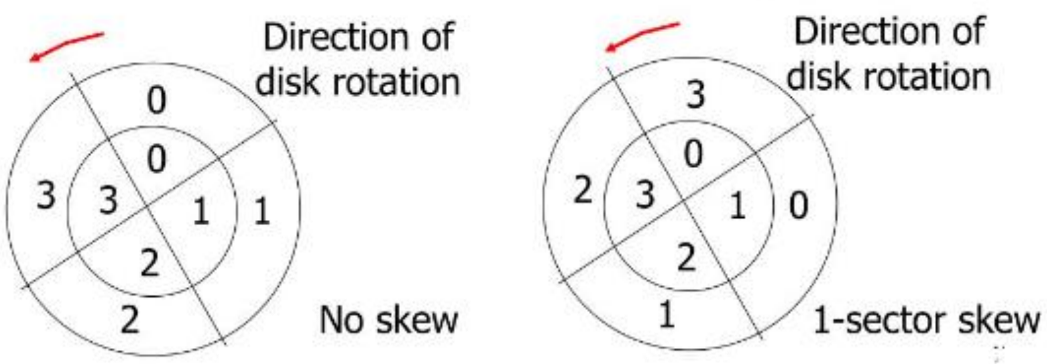


Figure: An illustration of cylinder skew



Interleaving

- Consider, for example, a controller with a one-sector buffer that has been given a command to read two consecutive sectors.
    - After reading the first sector from the disk and doing the ECC calculation, the data must be transferred to main memory.
    - While this transfer is taking place, the next sector will fly by the head.
    - When the copy to memory is complete, the controller will have to wait almost an entire rotation time for the second sector to come around again
- Such problem is solved by using interleaving.
- Interleaving is the process of numbering the sectors in interleaved fashion i.e. placing gaps between two sectors.
- It gives the controller some breathing space between consecutive sectors in order to copy the buffer to main memory.
    - Single Interleaving : 1 sector gap while numbering the sectors
    - Double Interleaving : 2 sectors gap while numbering the sectors

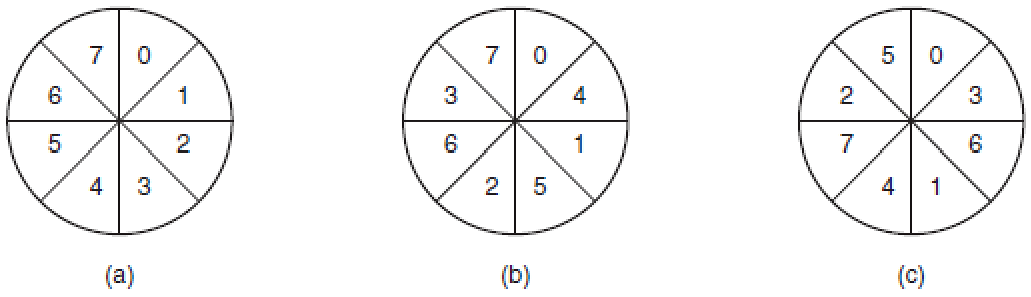


Figure: (a) No interleaving. (b) Single interleaving. (c) Double interleaving.

Partitioning:

⇒ As the name suggests, partitioning means creating divisions. It is a process of diving the hard disk into one or more regions, called partitions. An operating system can recognize these partitions as separate drives and can boot from the same.

⇒ Each partition is like a separate disk

- Sector 0 is MBR – Contains boot code + partition table
- Partition table has starting sector and size of each partition

⇒ It can be performed by the users and it will affect the disk performance.

High-level formatting:

- ⇒ Done for each partition
  - o Specifies boot block, free list, root directory, empty file system
- ⇒ The process of high-level formatting involves writing a file system, cluster size, and partition label on the drive. Users usually perform the procedure to erase the drive's contents to make it ready for a clean install.
- ⇒ Firstly High-level formatting clears the data on hard-disk, then it will generate boot information, then it will initialize FAT after this it will go for label logical bad sectors when partition has existed.
- ⇒ Formatting done by the user is the high-level formatting.

Error Handling

- Manufacturing defects introduce bad sectors, that is, sectors that do not correctly read back the value just written to them.
- If the defect is very small, say, only a few bits, it is possible to use the bad sector and just let the ECC correct the errors every time. If the defect is bigger, the error cannot be masked.
- There are two general approaches to bad blocks: deal with them in the controller or deal with them in the operating system.
- In the former approach, before the disk is shipped from the factory, it is tested and a list of bad sectors is written onto the disk. For each bad sector, one of the spares is substituted for it.
  - o Substituting a spare for the
  - o Shifting all the sectors to bypass the bad one
- In following figure (a), we see a single disk track with 30 data sectors and two spares. Sector 7 is defective. What the controller can do is remap one of the spares as sector 7 as shown in figure (b).
- The other way is to shift all the sectors up one, as shown in figure (c).
- In both cases the controller has to know which sector is which. It can keep track of this information through internal tables (one per track) or by rewriting the preambles to give the remapped sector numbers. If the preambles are rewritten, the method of figure (c) is

more work (because 23 preambles must be rewritten) but ultimately gives better performance because an entire track can still be read in one rotation.

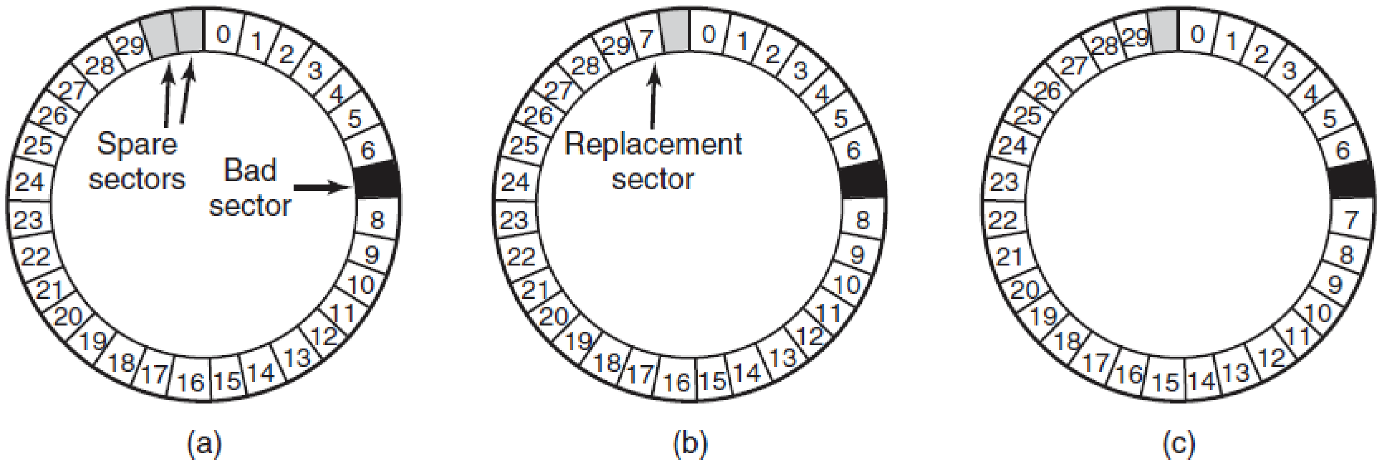


Figure: (a) A disk track with a bad sector. (b) Substituting a spare for the bad sector. (c) Shifting all the sectors to bypass the bad one.

- If the controller does not have the capability to transparently remap sectors as we have discussed, the operating system must do the same thing in software. This means that it must first acquire a list of bad sectors, either by reading them from the disk, or simply testing the entire disk itself. Once it knows which sectors are bad, it can build remapping tables. If the operating system wants to use the approach of figure(c), it must shift the data in sectors 7 through 29 up one sector.
- If the operating system is handling the remapping, it must make sure that bad sectors do not occur in any files and also do not occur in the free list or bitmap. One way to do this is to create a secret file consisting of all the bad sectors. If this file is not entered into the file system, users will not accidentally read it (or worse yet, free it).

RAID

- Redundant Array of Inexpensive Disks / Redundant Array of Independent Disks
- RAID is a way of storing the same data in different places in multiple disks.
- It works by placing data on multiple disks so that input/output operations can overlap in a balanced way in order to improve performance.
- *A technique which makes use of a combination of multiple disks instead of using a single disk for increased performance, data redundancy or both*
- There are different RAID levels

- A storage technology that combines multiple disks into a logical unit. Data is distributed across the disks in one of several ways called "RAID levels", depending on what level of redundancy and performance is required.
- The main idea behind RAID is to take some inexpensive disks and group them together, which will make the system see them as one single disk. This is done by using a RAID controller card that handle all I/O to the disks, and which knows where the stored data can be found.
- RAID provides parallel operations as data are distributed over different disk drives.
- The following are terms that are normally used in connection with RAID:
    - **Striping:** data is split between multiple disks for faster access
    - **Mirroring:** data is mirrored between multiple disks. Same data is written into another disk.
    - **Parity:** also referred to as a checksum. Parity is a calculated value used to mathematically rebuild data in case of failure of one of the disks.

Different RAID levels

RAID 0 – Striping

- Data (file) is split into blocks
- Stripe the blocks across disks in the system

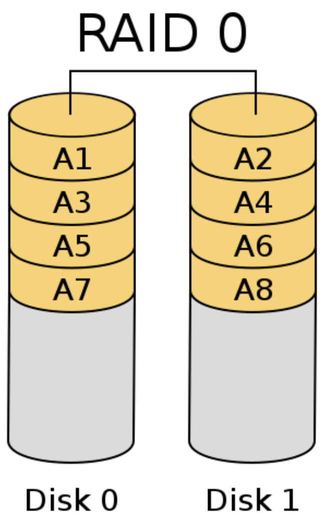


Figure: RAID 0

- In the above figure, odd blocks are placed at disk 0 and even blocks are placed at disk 1.

- It is easy to implement
- No parity calculation overhead is involved
- It provides good performance by spreading the load across many channels and drives.
- It provides no redundancy or error detection.
- Not true RAID because there is no fault tolerance.
- The failure of just one drive will result in all data in an array being lost.
- After certain amount of disk drives, performance does not increase significantly.
- Failure of either disk results in complete data loss in respective array
- It requires minimum 2 drives to implement.

RAID 1 - Mirroring

- This level performs mirroring of data in one disk to another.
- A complete file is stored in one disk. The second disk contains exact copy of the first disk.
- It offers redundancy and array will continue to work even if either disk fails.
- High read speed as either disk can be used if one disk is busy. Slow write performance as both disks have to be updated
- In case a drive fails, data do not have to be rebuild, they just have to be copied to the replacement drive.
- The main disadvantage is that the effective storage capacity is only half of the total drive capacity because all data get written twice.

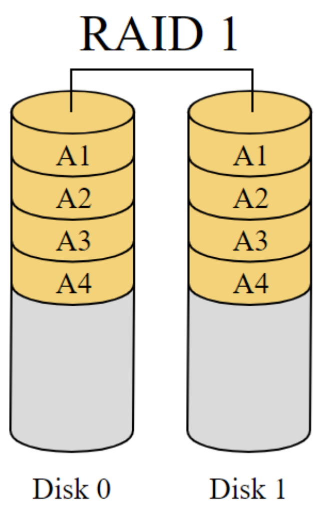


Figure: RAID 1

- It requires minimum 2 drives to implement.

RAID 2

- This level uses bit-level data stripping rather than block level with dedicated hamming code parity
- It uses ECC (Error Correcting Code) to monitor correctness of information in disk.
- The parity disk is used to reconstruct the corrupted or lost data.

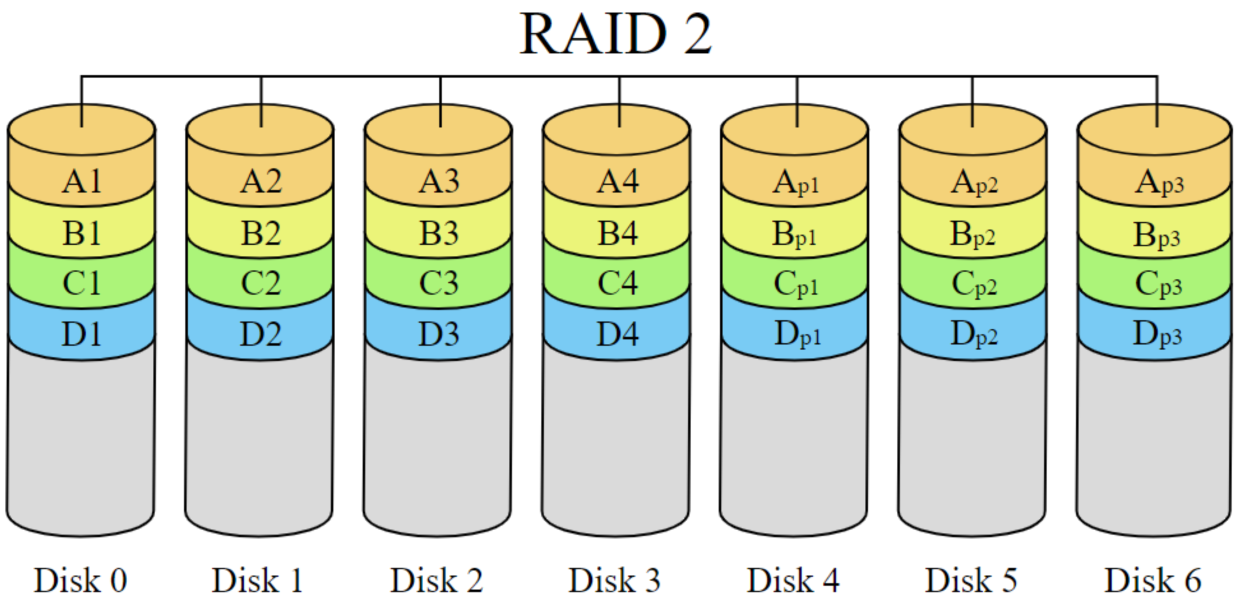


Figure: RAID 2

- Here, losing data from one disk do not cause problem, which can be handled by Hamming code on the fly.
- Big Problem is performance
  - Must have to read data plus ECC code from other disks
  - For a write, must have to modify data, ECC, and parity disks

RAID 3

- This level uses byte level stripping along with parity.
- One dedicated drive is used to store the parity information and in case of any drive failure the parity is restored using this extra drive.
- But in case the parity drive crashes then the causes problem
- Minimum 3 drives required
- No redundancy in case parity drive crashes.
- Data can be accessed parallel

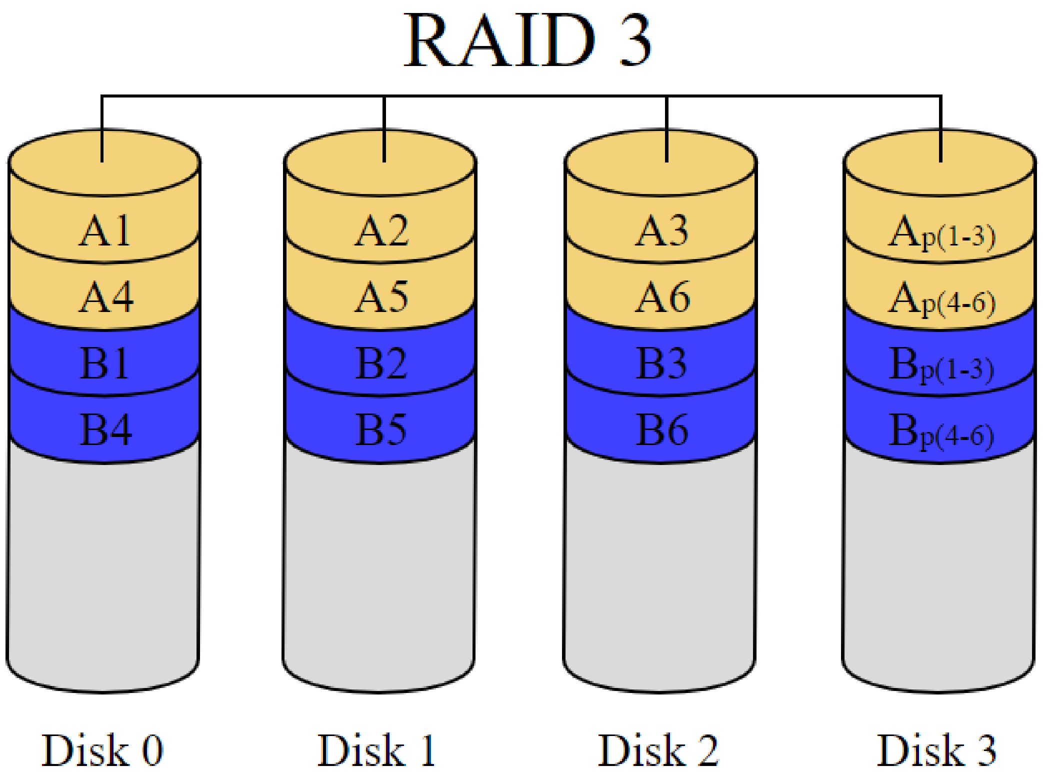


Figure :RAID 3

RAID 4

- This level is very much similar to RAID 3 apart from the feature where RAID 4 uses block level stripping rather than byte level
- One designated drive is used to store parity
- Minimum 3 drives required

- Provides good performance of random reads, while the performance of random writes is low due to the need to write all parity data to a single disk
- If a drive crashes then the lost block can be recomputed from parity drive by reading the entire set of drives.
- It creates heavy load on parity disk. It may become bottleneck.

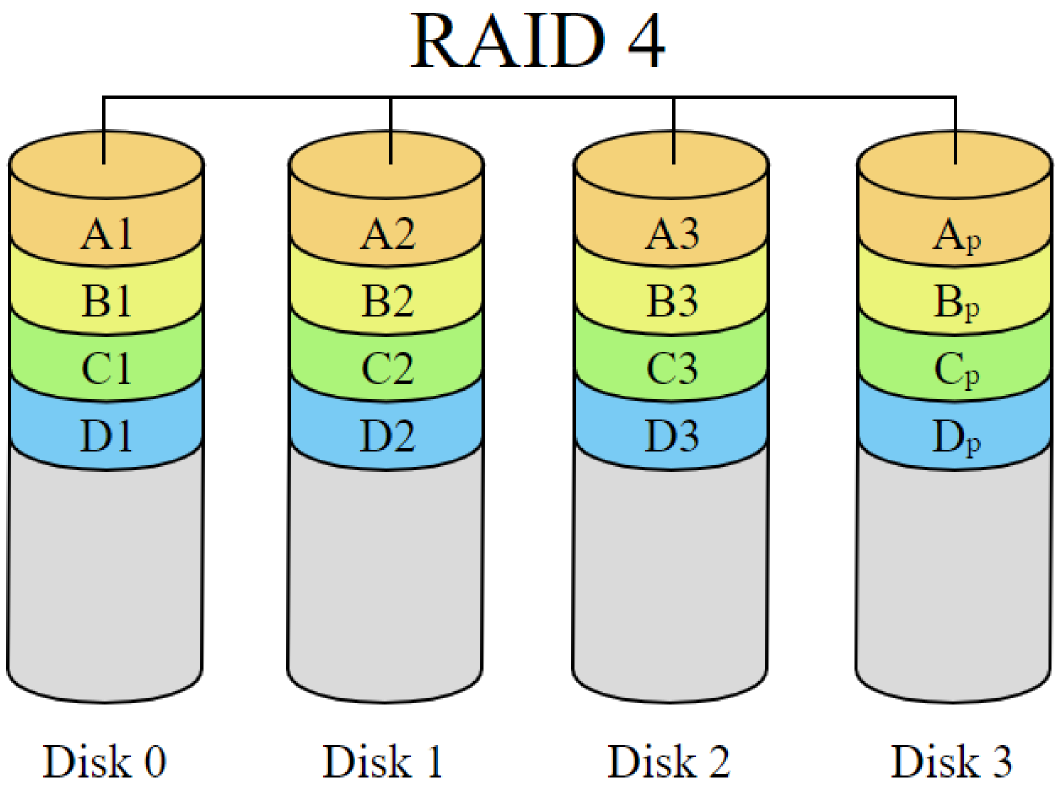


Figure: RAID 4

RAID 5

- This is based on block level stripping with parity
- In this level distributed parity concept came into the picture leaving behind the traditional dedicated parity disk as used in RAID 3 and RAID 4.
- Parity information is written to a different disk in the array for each stripe.
- In case of single disk failure data can be recovered with the help of distributed parity without affecting the operation and other read write operations.
- Block level stripping with Distributed parity
- Minimum 3 drives required
- High Performance

- Upon failure of a single drive, subsequent reads can be calculated from the distributed parity such that no data is lost
- In case of disk failure recovery may take longer time as parity has to be calculated from all available drives
- Cannot survive concurrent drive failures

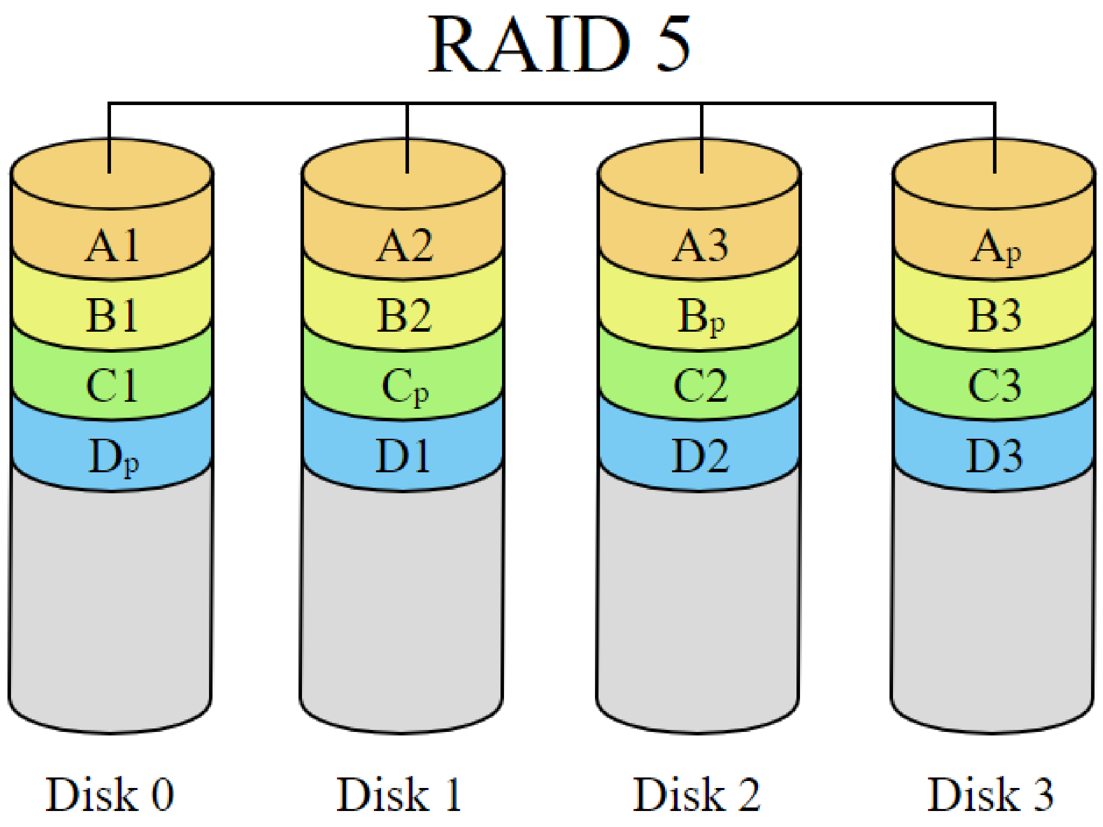


Figure: RAID 5

RAID 6

- This level is an enhanced version of RAID 5 adding extra benefit of dual parity.
- This level uses block level stripping with Dual distributed parity.
- So, extra redundancy.
- RAID 6 extends RAID 5 by adding another parity block; thus, it uses block-level striping with two parity blocks distributed across all member disks
- Two parity blocks are created
- Minimum 4 disks required
- Writing data takes longer time due to dual parity

- Can survive concurrent 2 drive failures in an array
- Extra Fault Tolerance and Redundancy

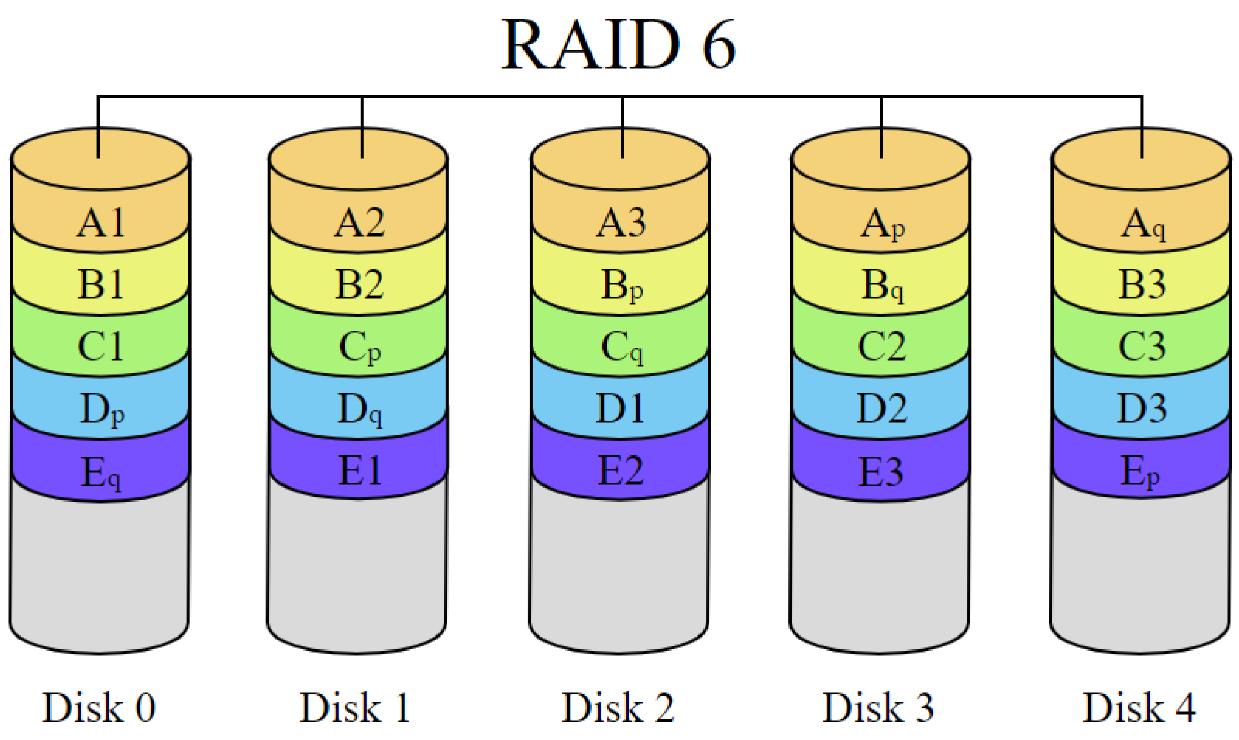


Figure: RAID 6

Stable Storage

- As we have seen, disks sometimes make errors. Good sectors can suddenly become bad sectors. Whole drives can die unexpectedly. RAIDs protect against a few sectors going bad or even a drive falling out. However, they do not protect against write errors laying down bad data in the first place. They also do not protect against crashes during writes corrupting the original data without replacing them by newer data.
- For some applications, it is essential that data never be lost or corrupted, even in the face of disk and CPU errors. Ideally, a disk should simply work all the time with no errors. Unfortunately, that is not achievable.
- What is achievable is a disk subsystem that has the following property:
    - When a write is issued to it, the disk either correctly writes the data or it does nothing, leaving the existing data intact.
- Such a system is called **stable storage** and is implemented in software. The goal is to keep the disk consistent at all costs.
- A slight variant of the original idea is described here.

- Before describing the algorithm, it is important to have a clear model of the possible errors. The model assumes that when a disk writes a block (one or more sectors), either the write is correct or it is incorrect and this error can be detected on a subsequent read by examining the values of the ECC fields.
    - In principle, guaranteed error detection is never possible
- The model also assumes that a correctly written sector can spontaneously go bad and become unreadable. However, the assumption is that such events are so rare that having the same sector go bad on a second (independent) drive during a reasonable time interval (e.g., 1 day) is small enough to ignore.
- The model also assumes the CPU can fail, in which case it just stops. Any disk write in progress at the moment of failure also stops, leading to incorrect data in one sector and an incorrect ECC that can later be detected. Under all these conditions, stable storage can be made 100% reliable in the sense of writes either working correctly or leaving the old data in place. Of course, it does not protect against physical disasters, such as an earthquake happening and the computer falling 100 meters into a fissure and landing in a pool of boiling magma. It is tough to recover from this condition in software.
- Stable storage uses a pair of identical disks with the corresponding blocks working together to form one error-free block. In the absence of errors, the corresponding blocks on both drives are the same. Either one can be read to get the same result. To achieve this goal, the following three operations are defined:
    1. **Stable writes:** A stable write consists of first writing the block on drive 1, then reading it back to verify that it was written correctly. If it was not, the write and reread are done again up to n times until they work. After n consecutive failures, the block is remapped onto a spare and the operation repeated until it succeeds, no matter how many spares have to be tried. After the write to drive 1 has succeeded, the corresponding block on drive 2 is written and reread, repeatedly if need be, until it, too, finally succeeds. In the absence of CPU crashes, when a stable write completes, the block has correctly been written onto both drives and verified on both of them.
    2. **Stable reads:** A stable read first reads the block from drive 1. If this yields an incorrect ECC, the read is tried again, up to n times. If all of

these give bad ECCs, the corresponding block is read from drive 2. Given the fact that a successful stable write leaves two good copies of the block behind, and our assumption that the probability of the same block spontaneously going bad on both drives in a reasonable time interval is negligible, a stable read always succeeds.

3. **Crash recovery:** After a crash, a recovery program scans both disks comparing corresponding blocks. If a pair of blocks are both good and the same, nothing is done. If one of them has an ECC error, the bad block is overwritten with the corresponding good block. If a pair of blocks are both good but different, the block from drive 1 is written onto drive 2

- In the **absence of CPU crashes, this scheme always works** because stable writes always write two valid copies of every block and spontaneous errors are assumed never to occur on both corresponding blocks at the same time. What about in the **presence of CPU crashes during stable writes?** It depends on precisely when the crash occurs. There are five possibilities, as depicted in following figure

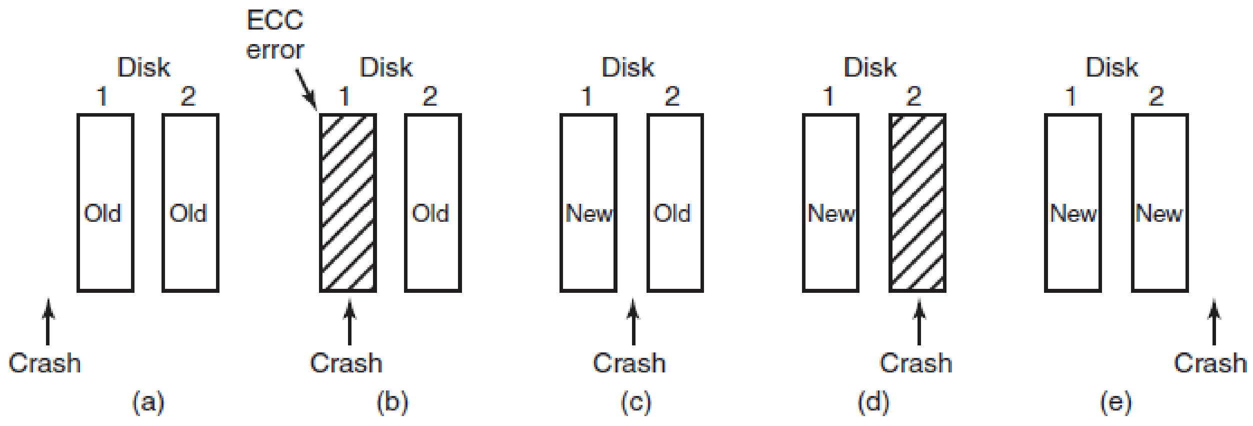


Figure: Analysis of the influence of crashes on stable writes

— In Fig.(a), the CPU crash happens before either copy of the block is written. During recovery, neither will be changed and the old value will continue to exist, which is allowed.

— In Fig. (b), the CPU crashes during the write to drive 1, destroying the contents of the block. However the recovery program detects this error and restores the block on drive 1 from drive 2. Thus the effect of the crash is wiped out and the old state is fully restored.

— In Fig.(c), the CPU crash happens after drive 1 is written but before drive 2 is written. The point of no return has been passed here: the recovery program copies the block from drive 1 to drive 2. The write succeeds.

— Fig. (d) is like Fig. (b): during recovery, the good block overwrites the bad block. Again, the final value of both blocks is the new one.

— Finally, in Fig. (e) the recovery program sees that both blocks are the same, so neither is changed and the write succeeds here, too.