Input Output Devices

The OS controls every resources including providing important abstraction like *a process* etc. I/O devices are no exception. The common tasks includes

- Initializing the I/O devices
- Catch interrupts
- Handle errors
- Providing a common interface between the devices and others in the system to establish the so called 'device independence'.

A large part of any OS is the code controllong I/O devices. And we are interested in the context of OS the programming part and not on how does it do it work internally. However, OS designers should have a general background of the working and internal structure of the devices.

The block and the Character devices

I/O devices are classified primarily as i) Block or ii) character devices.

- A block device stores information in blocks of fixed sizes (say, 1/2 KB to 64 KB) each having a unique address
- All blocks can be independently be read or written
- Technically from any block you may raech any other by *seek* operation

The boundary between block and character devices is thin (e.g., tape can be used as a block device).

- A character device reads or writes a stream of charcacter without any consideration of any block structure.
- It is not addressable and does not have any *seek* operation

Moreover, some devices may not fit either as a character or a block device (say, the clock, or the touch screen etc.). By nature some devices are block (disk, CD etc.) and some are character (Keyboard, printer). However, the model; block or character allow the OS to deal with most of the devices in an uniform manner at the higher level despite a huge variation of the data transfer speed.

Block and Character devices ... contd.

Block Devices:

- A block device stores information in blocks of fixed sizes (say, 1/2 KB to 64 KB) each having a unique address
- All blocks can be independently be read or written
- Technically from any block you may raech any other by *seek* operation

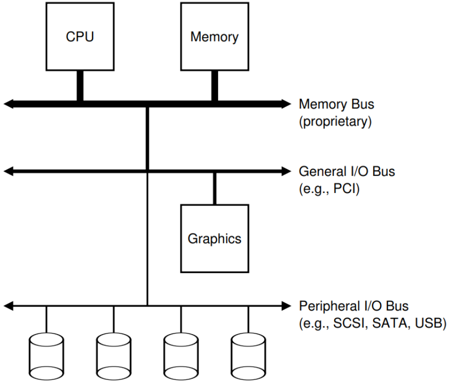
Character Devices:

- A character device reads or writes a stream of charcacter without any consideration of any block structure.
- It is not addressable and does not have any *seek* operation

The boundary between block and character devices is thin (e.g., tape can be used as a block device). Moreover, some devices may not fit either as a character or a block device (say, the clock, or the touch screen etc.). By nature some devices are block (disk, CD etc.) and some are character (Keybord, printer). However, the model; block or character allow the OS to deal with most of the devices in an uniform manner at the higher level.

I/O system

We see a single CPU attached to the main memory via a memory bus. Through general I/O bus (PCI may be); graphics and some other higher-performance I/O devices may be connected. In a lower hierarchy we have the peripheral bus, such as SCSI, SATA, or USB to connect slow devices.
A high performance bus is quite costly. So, devices demanding high bandwidth are placed near to the CPU. Lower performance components are kept further away.

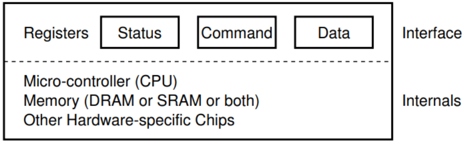


I/O system Interface

We know the OS offers a clean and symmetrical interface for all the I/O devices irrespective of their complexity. For an I/O device we have

- i) a h/w interface (all devices have some specified interface and protocol for interaction), and
- ii) is the internal structure. This part of the device is implementation specific and is responsible for implementing the abstraction the device presents to the system. Very simple devices will have one or a few hardware chips to implement their functionality; more complex devices will include a simple CPU, some general purpose memory, and other device-specific chips to get their job done.

For better understanding and for a unified view consider a canonical device shown below:



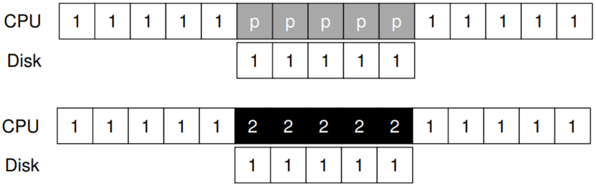
Protocol : Polling

We see that the OS interacts with the canonical devices through a few registers; most common are *status*, *command* and *data*. A typical canonical protocol (for writing) can be
While (STATUS == BUSY) ; wait until device is not busy
Write data to DATA register AND Write command to COMMAND register
(Doing so starts the device and executes the command)
While (STATUS == BUSY) ; // wait until finish
The protocol (polling – very ineffcient) has four steps where the OS

- 1. waits until the device is ready to receive a command. by repeatedly reading the status register; we
- 2. sends some data down to the data register; one can imagine
- 3. writes a command to the command register; to make the device obeying the command for certain action.
- 4. waits for the device to finish (it may then get an error code to indicate success or failure).

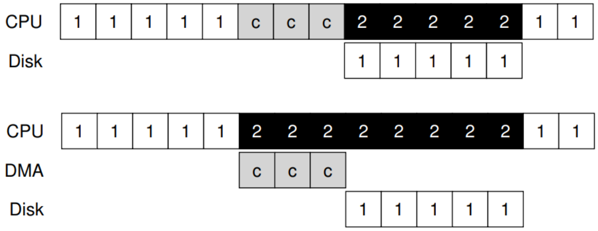
Protocol : Interrupt driven

The polling method keeps CPU busy without deliveriung anything. Thus an interrupt driven I/O is a preferred choice. Here, the OS instead of making the CPU a busy-wait puts the calling process to a sleep and schedule other process to run. On completion of the task the I/O device can raise a h/w interrupt that forces the CPU to run an interrupt handler (to run the ISR).
The handler is just a piece of operating system code that will finish the request (for example, by reading data and perhaps an error code from the device) and wake the process waiting for the I/O, which can then proceed as desired.
The figures compared the loss in polling – during the writing (disk I/O) the CPU runs process 2 for a while to increase CPU utilisation.
Interrupt driven I/O is good for slow devices. However for high speed devices we may use programmed I/O as polling time is low to avoid interrupt overhead.



DMA

It is better to make direct transfer for a large chuck of data between memory and I/O (or memory to memory) making CPU free to do other tasks*.
In the timeline shown in the 1st figure, Process 1 is running and then wishes to write some data to the disk. It then initiates I/O which must copy the data from memory to the device explicitly, one word at a time (marked c in the diagram). When the copy is complete, the I/O begins on the disk and the CPU can finally be used for something else.
From the timeline shown in the second figure, we see that the copying of data is now handled by the DMA controller. Because the CPU is free during that time, the OS can do something else, here choosing to run Process 2. Process 2 thus gets to use more CPU before Process 1 runs again.



I/O : I/O mapped or Memory Mapped I/O

To communicate with the I/O devices two methods are used

- The first, oldest method is I/O Mapped I/O is to have explicit I/O instructions.
    - These instructions specify a way for the OS to send data to specific device registers and thus allow the construction of the protocols described above.
    - Instruction like IN and OUT (as in x86) are used to read/write the data and command registers; obviously these instructions are privileged
- The second method to interact with devices is known as memory mapped I/O.
    - With this approach, the hardware makes device registers available as if they were memory locations.
    - To access a particular register, the OS issues a load (to read) or store (to write) the address; the hardware then routes the load/store to the device instead of main memory.

As memory mapped I/O is more flexible it is more popular now. However, I/O mapped I/O is also in use in many system.

Device controller

As OS does not deal with the device itself (its inner working and the mechanics) the device controller is actually important.

- In many system the controller is a chip on a PCB that is inserted directly on a bus slot on motherboard (e.g., PCIe)
- The mechanical component is the device itself connected to the controller through a cable following a standard interface like SCSI, SATA, USB etc.
- A controller is capabnle of handling multiple devices(2 to 8 or more) as well.

The controller does many low level operations that helps OS. A disk, for example, might be formatted with 2,000,000 sectors of 512 bytes per track. For a read operation a series of bits is returned by the device that comprises of

- A preamble (cylinder no., sector no, sector size, etc,)
- 4096 (512 x 8) bits and ECC

The controller converts the serial bit stream into a block of bytes (assembled bit by bit in a buffer) and does error correction. Finally the verified and error free data is copied to main memory. Similar actions are taken by the respective controllers for different devices.

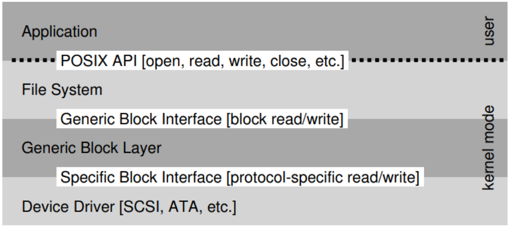
Talking to the device controller

The OS must interact with the device controller through a number of registers inside the controller for

- Initialisation
- R/W the register for a number of purposes regrading the data transfer (modes etc.) and might also be able to reach the buffers the controller maintains (i.e., Video RAM – a buffer inside the controller available for the programs or OS writing to it for the display)
- We have two alternatives to access i) Controller ports are Memory mapped; ii) having an I/O port number which the CPU access through special I/O instruction (i.e., having two different address space; memory for the real memory device and I/O specifically for the I/O devices (in fact the controllers)

Device Driver : Towards common interface

How can we keep most of the OS device-neutral, thus hiding the details of device interactions from major OS subsystems? The problem is solved through the age-old technique of abstraction. At the lowest level, a piece of software in the OS must know in detail how a device works. We call this piece of software a device driver, and any specifics of device interaction are encapsulated within.
Let us see the Linux file system software stack handling disk I/O. Note that a file system (and certainly, an application above) is completely oblivious to the specifics of which disk class it is using; it simply issues block read and write requests to the generic block layer, which routes them to the appropriate device driver, which handles the details of issuing the specific request.


Application
user
POSIX API [open, read, write, close, etc.]
File System
Generic Block Interface [block read/write]
kernel mode
Generic Block Layer
Specific Block Interface [protocol-specific read/write]
Device Driver [SCSI, ATA, etc.]

Hard Disk Drive

Here, we dive into more detail about one device in particular: the hard disk drive. These drives have been the main form of persistent data storage in computer systems for decades.
Let's start by understanding the interface to a modern disk drive.

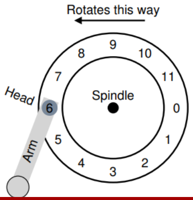
- Let's start by understanding the interface to a modern disk drive. The basic interface for all modern drives is straightforward. The drive consists of a large number of sectors (512-byte blocks), each of which can be read or written.
- Multi-sector operations are possible; indeed, many file systems will read or write 4KB at a time (or more). However, when updating the disk, the only guarantee drive manufactures make is that a single 512- byte write is atomic (i.e., it will either complete in its entirety or it won't complete at all); thus, if an untimely power loss occurs, only a portion of a larger write may complete (sometimes called a torn write).

A simple Hard Disk Drive

- Let's start by understanding the interface to a modern disk drive. The basic interface for all modern drives is straightforward. The drive consists of a large number of sectors (512-byte blocks), each of which can be read or written.
- Multi-sector operations are possible; indeed, many file systems will read or write 4KB at a time (or more). However, when updating the disk, the only guarantee drive manufactures make is that a single 512- byte write is atomic (i.e., it will either complete in its entirety or it won't complete at all); thus, if an untimely power loss occurs, only a portion of a larger write may complete (sometimes called a torn write).

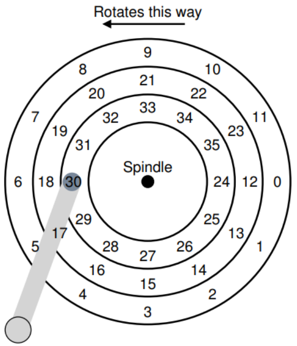
A simple Hard Disk Drive ... contd.

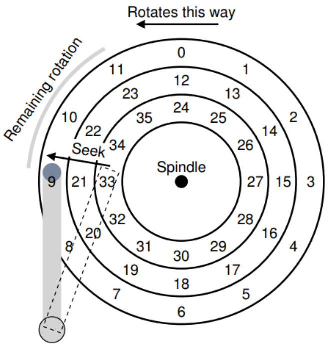
Assume we have a simple disk with a single track. This track has just 12 sectors, each of which is 512 bytes in size (our typical sector size, recall) and addressed therefore by the numbers 0 through 11. The single platter we have here rotates around the spindle, to which a motor is attached. Of course, the track by itself isn't too interesting; we want to be able to read or write those sectors, and thus we need a disk head, attached to a disk arm, as we now see. In the figure, the disk head, attached to the end of the arm, is positioned over sector 6, and the surface is rotating counter-clockwise.
Single-track Latency: The Rotational Delay
In order to access sector 0 we need not move the r/w head – we simply wait for the required sector to come underneath. This delay is rotational delay and given by $R/2$ (on an average) where R is the full rotation time



A simple Hard Disk Drive ... contd.

Multiple tracks – Seek Time:
Let's thus look at ever-so-slightly more realistic disk surface, this one with three tracks (Figure left). The head is currently positioned over the innermost track (which contains sectors 24 through 35); the next track over contains the next set of sectors (12 through 23), and the outermost track contains the first sectors (0 through 11).



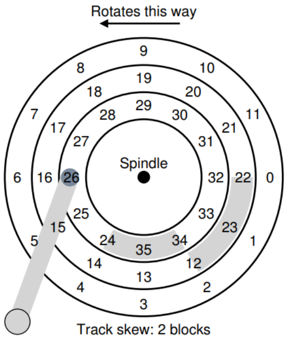


A simple Hard Disk Drive ... contd.

What would happen for a read to sector 11. To service this read, the drive has to first move the disk arm to the correct track (in this case, the outermost one), in a process known as a seek. Seeks, along with rotations, are one of the most costly disk operations. The seek, it should be noted, has many phases: first an acceleration phase as the disk arm gets moving; then coasting as the arm is moving at full speed, then deceleration as the arm slows down; finally settling as the head is carefully positioned over the correct track. The settling time is often quite significant, e.g., 0.5 to 2 ms, as the drive must be certain to find the right track (imagine if it just got close instead!). After the seek, the disk arm has positioned the head over the right track. A depiction of the seek is found in the Figure (right) above. So, during the seek, the arm has been moved to the desired track, and the platter of course has rotated, in this case about 3 sectors. Thus, sector 9 is just about to pass under the disk head, and we must only endure a short rotational delay to complete the transfer. When sector 11 passes under the disk head, the final phase of I/O will take place, known as the transfer, where data is either read from or written to the surface. And thus, we have a complete picture of I/O time: first a seek, then waiting for the rotational delay, and finally the transfer

A simple Hard Disk Drive ... contd.

Many drives employ some kind of track skew to make sure that sequential reads can be properly serviced even when crossing track boundaries. In our simple example disk, this might appear as seen in Figure.
Sectors are often skewed like this because when switching from one track to another, the disk needs time to reposition the head (even to neighboring tracks). Without such skew, the head would be moved to the next track but the desired next block would have already rotated under the head, and thus the drive would have to wait almost the entire rotational delay to access the next block.



A simple Hard Disk Drive ... contd.

Due to more track length; the outer tracks tend to have more sectors than inner tracks. These tracks are often referred to as multi-zoned disk drives, where the disk is organized into multiple zones, and where a zone is consecutive set of tracks on a surface with the same number of sectors per track, and outer zones have more sectors than inner zones.
Modern disk has cache (a.k.a track buffer: 8/16 MB) to hold data read from or written to the disk. For example, when reading a sector from the disk, the drive might decide to read in all of the sectors on that track and cache them in its memory; doing so allows the drive to quickly respond to any subsequent requests to the same track.
On writes, the drive has a choice: should it acknowledge the write has completed when it has put the data in its memory, or after the write has actually been written to disk? The former is called **write back** (is risky in certain cases) caching, and the latter **write through** caching.

A simple Hard Disk Drive ... contd.

I/O Time Now that we have an abstract model of the disk, we can use a little analysis to better understand disk performance. In particular, we can now represent I/O time as the sum of three major components:

$$T_{I/O} = T_{seek} + T_{rotation} + T_{transfer}$$

Note that the rate of I/O ($R_{I/O}$), which is often more easily used for comparison between drives (as we will do below), is easily computed from the time. Simply divide the size of the transfer by the time it took: $R_{I/O} = \frac{Size_{transfer}}{T_{I/O}}$

To get a better feel for I/O time, let us perform the following calculation. Assume there are two workloads we are interested in. The first, known as the random workload, issues small (e.g., 4KB) reads to random locations on the disk. Random workloads are common in many important applications, including database management systems. The second, known as the sequential workload, simply reads a large number of sectors consecutively from the disk, without jumping around. Sequential access patterns are quite common and thus important as well.

A simple Hard Disk Drive ... contd.

I/O Time For a random trasfer of 4 KB block for the Seagate Cheetah disk (with 15000 rpm and a maximum transfer rate of 125 MB/s and an average seek time of 4 ms) we see that the full rotational delay is 1000/250 ms (rotational speed 15000/60 or 250 rps) or 4 ms. So, the average rotational delay is 2 ms. THe transfer time for 4 KB is 32 microsecond (4 KB x 1000/ 125) only. Now, $T_{I/O} = T_{seek} + T_{rotation} + T_{transfer} = 4 + 2 + 0.032 = 6.032ms$
So, for the Cheetah $R_{I/O}$ is $0.004/0.006032 = 0.66MB/s$ under the random workload. For the Barracuda (Seagate) disk with rotational speed of 7500 rpm, 9 ms of average seek and a maximum transfer rate of 105 MB/s the $R_{I/O}$ is $0.004/0.0132 = 0.31MB/s$ (approx).

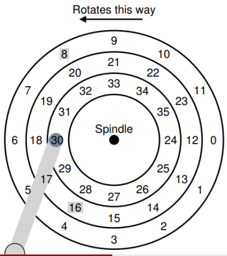
For sequential workload there is a single seek and rotation before a very long transfer. Assuming the transfer size is 100 MB. Thus, $T_{I/O}$ for the Chetaah and Barracuda is about 800 ms and 950 ms, respectively. The rates of I/O are thus very nearly the peak transfer rates of 125 MB/s and 105 MB/s, respectively. [Note that average disk seek time for N track is considered N/3; where the basis of the calculation is the average seek distance $= \sum_{x=0}^{N} \sum_{y=0}^{N} |x - y|$ divided by possible no. of seeks (N^2).

Disc Scheduling Algorithms

Because of the high cost of I/O, the OS has historically played a role in deciding the order of I/Os issued to the disk. More specifically, given a set of I/O requests, the disk scheduler examines the requests and decides which one to schedule next. Unlike job scheduling, where the length of each job is usually unknown, with disk scheduling, we can make a good guess at how long a "job" (i.e., disk request) will take by bestimating the seek and possible the rotational delay of a request and may take a strategic policy decision (shortest job first)
SSTF: Shortest Seek Time First One early disk scheduling approach is known as shortest-seek-time-first (SSTF) (also called shortest-seek-first or SSF). SSTF orders the queue of I/O requests by track, picking requests on the nearest track to complete first. It is not a good policy as starvation may happen when you have a series of requests from nearby tracks while a far-off track access is pending.
Elevator (a.k.a. SCAN or C-SCAN): The algorithm, originally called SCAN, simply moves across the disk servicing requests in order across the tracks.
C-SCAN is similar to SCAN but moves from one end to the other and then reverses the direction to cover the tracks in that direction.
SCAN and its cousins do not represent the best scheduling technology. In particular, SCAN (or SSTF even) do not actually adhere as closely to the principle of SJF as they could. In particular, they ignore rotation.

Disc Scheduling Algorithms

Shortest positioning/access time first or SPTF (or SATF) scheduling : is better in most cases as it takes care of the rotation as well.
In the example, the head is currently positioned over sector 30 on the inner track. The scheduler thus has to decide: should it schedule sector 16 (on the middle track) or sector 8 (on the outer track) for its next request. So which should it service next? It depends relative to the the seek time and the rotational delay. If seek time is much higher than the rotational delay then SSTF (and other variants) are all fine. In our example then serving the outer track 8 is a better choice than to serve track 16 on the middle with a shorter seek time but a full rotational delay. IN modern devices the disk electronics makes the choice releiving the OS doing this complicated decision making process involving some high speed calculation.



Redundant Array of Inexpensive Disks: RAID

Disks are bottleneck of the system as they are slow, not very big and in case of a crash we may lose data if they are not backed-up. The quest for a low cost solution is RAID. In RAID we use multiple disks to make it bigger and fail safe. A hardware RAID is very much like a computer system, specialized for the task of managing a group of disks. The advantages of the RAID ovefr a single disk are

- **Performance:** Using multiple disks in parallel can greatly speed up I/O times.
- **Capacity:** Large data sets demand large disks.
- **Reliability:** spreading data across multiple disks (without RAID techniques) makes the data vulnerable to the loss of a single disk; with some form of redundancy, RAIDs can tolerate the loss of a disk and keep operating as if nothing were wrong.

RAIDs provide these advantages transparently to systems, i.e., a RAID just looks like a big disk to the host system. It enables one to simply replace a disk with a RAID and not change a single line of software; the OS and client applications continue to operate without modification.

RAID LEVEL 0 or STRIPING

This level is not actually offers RAID as there is no redundancy. Here we see a simple striping of blocks (say, block size is 4 KB) for a 4-disk array

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

This arrangement is good for requests of large chunk of data at a time – that leads to greater parallelism. A row is a considered a stripe and in this case it holds 16 KB. Another arrangement gets 2 - blocks tohgether before moving to the next disk (32 KB/stripe) as shown below.

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

DISK MAPPING ... RAID LEVEL 0

Now it is relevant to know how the logical block access request is mapped to a particular physical address (disk addressas – you have many in a RAID system) Considering our first striping; i.e., chunk size = 1 block = 4 KB we could use the following to map a logical address 'A' to

$$\text{Disk} = A \ \% \ number\_of\_disks \text{ and Offset} = A \ / \ number\_of\_disks$$

Let's see how these equations work for a simple example. Imagine in the Mapping example: Given that there are 4 disks, this would mean that the disk we are interested in is (14 disk 2. The exact block is calculated as (14 / 4 = 3): block 3. Thus, block 14 should be found on the fourth block (block 3, starting at 0) of the third disk (disk 2, starting at 0), which is exactly where it is.

Chunk size: is important as its affects the RAID performance. A small chunk size implies that many files will get striped across many disks, thus increasing the parallelism of reads and writes to a single file. However, the positioning time increases (= highest positioning time accross all drives for the request). A higher chunk size reduces intra file parallelism but posiitoning time reduces as well. In a nutshell the best chunk size depends on the workload.

RAID LEVEL 0: Analysis and Performance

RAID 0 offers capacity, and performance (all disks are used to their full capacity and all participate parallely in most of the cases. However, reliability suffers with data loss due to any crash.
Consider a RAID with chunk size 4 KB working with mixed workload of Random accesss request to transfer 10 KB and sequential access request of 10 MB on an average. The disk has a transfer rate of 50 MB/s while the average seek and average rotaional delay time are 7 ms and 3 ms, respectively. Let us assume the transfer rate for sequential access is S and that for random access is R. To tarnsfer 10 MB is sequential mode we need 10/50 or 200 ms and the disk latency is 10 ms. So, S=Amount of transfer/Total time to transfer= 10 MB/210ms=47.62 MB/s. The tarnsfer time for random request is 10KB/50MB = 0.195 ms. So, R= 10 KB/10.195 ms = 0.981 MB/s.
The latency of a single-block request should be just about identical to that of a single disk; after all, RAID-0 will simply redirect that request to one of its disks.
Steady state throughput is N x S where N is the number of disks and S is the sequential bandwidth of a single disk.
For a large number of random I/Os, we can again use all of the disks, and thus obtain N x R MB/s.

RAID LEVEL 1

RAID 1 offers tolerance to disk failure as the data blocks are mirrored (written in two different disks) In a typical mirrored system, we will assume that for each logical block, the RAID keeps two physical copies of it. Here is an example:

DISK 0	DISK 1	DISK 2	DISK 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

When reading a block from a mirrored array, the RAID has a choice: it can read either copy. For example, if a read to logical block 5 is issued to the RAID, it is free to read it from either disk 2 or disk 3. When writing a block, though, no such choice exists: the RAID must update both copies of the data, in order to preserve reliability. Do note, though, that these writes can take place in parallel; for example, a write to logical block 5 could proceed to disks 2 and 3 at the same time.

RAID LEVEL 1 ... contd.

RAID 1 offers 1/2 the capacity as it is mirroring (level = 2) the disk blocks. Reliability is the strong point. We can tolerate failure in one disk, With a little luck this may be more than a single failure. For example, that disk 0 and disk 2 both failed – still is no data loss! More generally, a mirrored system (with mirroring level of 2) can tolerate 1 disk failure for certain, and up to N/2 failures depending on which disks fail. In practice, we generally don't like to leave things like this to chance; thus most people consider mirroring to be good for handling a single failure.

performance: From the perspective of the latency of a single read request, we can see it is the same as the latency on a single disk; all the RAID-1 does is direct the read to one of its copies. A write is a little different: it requires two physical writes to complete before it is done. These two writes happen in parallel, and thus the time will be roughly equivalent to the time of a single write; however, because the logical write must wait for both physical writes to complete, it suffers the worst-case seek and rotational delay of the two requests, and thus (on average) will be slightly higher than a write to a single disk.

RAID LEVEL 1 ... contd.

THE RAID CONSISTENT-UPDATE PROBLEM: For any multi disk RAID system with mirroring there may be a problem of inconsistency while writing. Assume a write request is to be written in disk 0 and disk 1. The RAID issues write to disk 0 but before write to DISK 1 is issued there is a crash. In this case the copy between DISK 0 (NEW) and DISK 1 (OLD) would be different.

The general way to solve this problem is to use a write-ahead log of some kind to first record what the RAID is about to do (i.e., update two disks with a certain piece of data) before doing it. By taking this approach, we can ensure that in the event of a crash, the right thing will happen; by running a recovery procedure that replays all pending transactions to the RAID, we can ensure that no two mirrored copies (in the RAID-1 case) are out of sync. However, logging to disk on every write is prohibitively expensive, most RAID hardware includes a small amount of non-volatile RAM (e.g., battery-backed) where it performs this type of logging. Thus, consistent update is provided without the high cost of logging to disk.

RAID LEVEL 1 ... contd.

Steady state sequential writing throuhgput for mirroring level 2 is $\frac{n}{2} \times S$ i.e., half the peak bandwidth. For sequential reading we got the same performance level though we are reading a single copy from one of the 2 seprate sources. Unfortunately, we obtain the exact same performance during a sequential read. One might think that a sequential read could do better, because it only needs to read one copy of the data, not both. However, let's use an example to illustrate why this doesn't help much. Imagine we need to read blocks 0, 1, 2, 3, 4, 5, 6, and 7. Let's say we issue the read of 0 to disk 0, the read of 1 to disk 2, the read of 2 to disk 1, and the read of 3 to disk 3. We continue by issuing reads to 4, 5, 6, and 7 to disks 0, 2, 1, and 3, respectively. One might naively think that because we are utilizing all disks, we are achieving the full bandwidth of the array. However, during the track skipping no data is actually transferred. So, each disk will not be able to deliver its peak bandwidth. So, sequential read throughput will be same as writing i.e., $\frac{N}{2} \times S$
For random read we can distribute the reads all over the disks and get a throughput of $N \times R$. However, random writes would deliver $\frac{N}{2} \times R$

RAID LEVEL 4 : Saving space using Parity

Note : RAID LEVEL 2 and 3 are not used now and dropped from the discussion
The huge space penalty to mirror can be avoided by using parity bit. Lets see the arrangement in RAID LEVEL 4.

DISK 0	DISK 1	DISK 2	DISK 3	DISK 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

The Parity block contains redundant information computed from the data blocks. For example P1 is computed using data from 4, 5, 6 and 7 blocks. Actually XOR (even no. of 1's returns 0 and odd number of 1's returns 1) is performed to compute Parity as shown below

C0	C1	C2	C3	P
0	0	1	1	XOR(0,0,1,1) = 0
0	1	0	0	XOR(0,1,0,0) = 1

1-bit error can be restored using the parity bit. For example assume row 1 C2 bit is lost. It will be C2(row1) = XOR(C0, C1, C3, P) = XOR(0, 0, 1, 0) = 1.
Similarly, C3(row2) = XOR(0, 1, 0, 1) = 0.

RAID LEVEL 4 ... contd.

Capacity utilisation for RAID 4 is N-1 (due to parity disk). Tolerates 1-DISK failure. Sequential read performance is (N-1) x S as we can parallely read. For sequential write RAID 4 can write to all disk parallely; known to be a full stripe write. So, the effective bandwidth is again (N-1) x S. For random read it will be (N-1) x R.
Random write is most interesting. Suppose we have a write request to block 1. Now, we not only have to write block 1 but need to calculate and update parity. So, we have to read blocks 0, 2 and 3 as well. And then XOR it with new block (1) to get the corresponding parity bit for P0. This is known as **additive parity**. The number of reads to compute parity increases with the number of disks. To avoid we use **subtractive parity**. Take 5 data bits (4 + 1 for parity) C0 = 0, C1 = 0; P, C2 = 1, C3 = 1, P = XOR(0,0,1,1) = 0. Say we are updating C2 (new). The parity is then calculated without considering other bits (0, 1 and 3) by the following

P(new) = (C2(old) XOR C2(new)) XOR P(old) This operation can be done over a block (4096 x 8 bits). So, the new block and parity need be updated.

RAID LEVEL 4: Small Write Problem

For small write requests to update blocks, say, 4 and 13 (Also, subtractive parity) we see the bottleneck due to the parity disk that prevents parallelism (for small write requests)

DISK 0	DISK 1	DISK 2	DISK 3	DISK 4
0	1	2	3	P0
*4	5	6	7	+P1
8	9	10	11	P2
12	*13	14	15	+P3

Note that 4 and 13 can be written parallely. However, we need P1 and P3 to be updated as well and this cannot be done parallely as both are stored in the parity disk (ie., all writes are to be serialized due to the single parity disk)
Because the parity disk has to perform two I/Os (one read, one write) per logical I/O, we can compute the performance of small random writes in RAID-4 by computing the parity disk's performance on those two I/Os, and thus we achieve (R/2) MB/s and it is really poor and cannot be improved by adding more disks.

RAID LEVEL 5: Rotating Priority

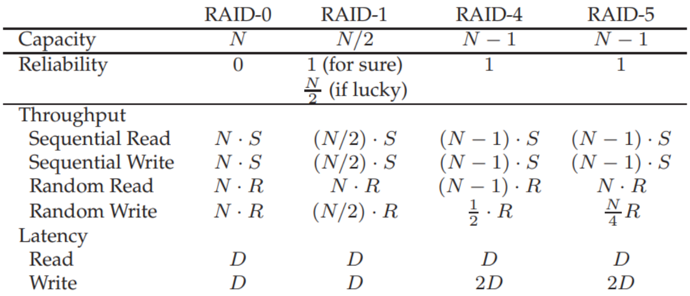
To avoid parity disk bottleneck in RAID 5 the parity blocks are rotated over all the disks; the rest is very similar to RAID 4. For example, the effective capacity and failure tolerance of the two levels are identical. So are sequential read and write performance. The latency of a single request (whether a read or a write) is also the same as RAID-4.

DISK 0	DISK 1	DISK 2	DISK 3	DISK 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

Random read performance is a little better as we could use all the disks. Random write performance is much better as we could use disks parallely. we can generally assume that that given a large number of random requests, we will be able to keep all the disks about evenly busy. If that is the case, then our total bandwidth for small writes will be (N/4) x R MB/s. The factor of four loss is due to the fact that each RAID-5 write still generates 4 total I/O operations, which is simply the cost of using parity-based RAID.

RAID : Comparison

There are many possible RAID levels to choose from, and the exact RAID level to use depends heavily on what is important to the end-user. For example, mirrored RAID is simple, reliable, and generally provides good performance but at a high capacity cost. RAID-5, in contrast, is reliable and better from a capacity standpoint, but performs quite poorly when there are small writes in the workload. Picking a RAID and setting its parameters (chunk size, number of disks, etc.) properly for a particular workload is challenging, and remains more of an art than a science.



	RAID-0	RAID-1	RAID-4	RAID-5
Capacity	N	$N/2$	$N-1$	$N-1$
Reliability	0	1 (for sure) $\frac{N}{2}$ (if lucky)	1	1
Throughput				
Sequential Read	$N \cdot S$	$(N/2) \cdot S$	$(N-1) \cdot S$	$(N-1) \cdot S$
Sequential Write	$N \cdot S$	$(N/2) \cdot S$	$(N-1) \cdot S$	$(N-1) \cdot S$
Random Read	$N \cdot R$	$N \cdot R$	$(N-1) \cdot R$	$N \cdot R$
Random Write	$N \cdot R$	$(N/2) \cdot R$	$\frac{1}{2} \cdot R$	$\frac{N}{4} R$
Latency				
Read	D	D	D	D
Write	D	D	$2D$	$2D$