**Unit-4**

**DEADLOCKS**

* **Deadlock-definition:** A set of processes is deadlocked when every process in the set is waiting for a resource that is currently allocated to another process in the set and which can only be released when that other waiting process makes progress.
* **various deadlock handling mechanisms:**

There are three ways of handling deadlocks

* 1. Deadlock prevention or avoidance - Do not allow the system to get into a deadlocked state.
  2. Deadlock detection and recovery - Abort a process or pre-empt some resources when deadlocks are detected.
  3. Ignore the problem all together - If deadlocks only occur once a year or so, it may be better to simply let them happen and reboot as necessary than to incur the constant overhead and system performance penalties associated with deadlock prevention or detection. This is the approach that both Windows and UNIX take.
* In order to avoid deadlocks, the system must have additional information about all processes. In particular, the system must know what resources a process will or may request in the future. (Ranging from a simple worst-case maximum to a complete resource request and release plan for each process, depending on the particular algorithm. )
* Deadlock detection is fairly straightforward, but deadlock recovery requires either aborting processes or pre-empting resources, neither of which is an attractive alternative.
* If deadlocks are neither prevented nor detected, then when a deadlock occurs the system will gradually slow down, as more and more processes become stuck waiting for resources currently held by the deadlock and by other waiting processes. Unfortunately this slowdown can be indistinguishable from a general system slowdown when a real-time process has heavy computing needs.

**DEADLOCK PREVENTION**

* Deadlocks can be prevented by preventing at least one of the four required conditions:

**7.4.1 Mutual Exclusion**

* Shared resources such as read-only files do not lead to deadlocks.
* Unfortunately some resources, such as printers and tape drives, require exclusive access by a single process.

**7.4.2 Hold and Wait**

* To prevent this condition processes must be prevented from holding one or more resources while simultaneously waiting for one or more others. There are several possibilities for this:
  + Require that all processes request all resources at one time. This can be wasteful of system resources if a process needs one resource early in its execution and doesn't need some other resource until much later.
  + Require that processes holding resources must release them before requesting new resources, and then re-acquire the released resources along with the new ones in a single new request. This can be a problem if a process has partially completed an operation using a resource and then fails to get it re-allocated after releasing it.
  + Either of the methods described above can lead to starvation if a process requires one or more popular resources.

**7.4.3 No Preemption**

* Preemption of process resource allocations can prevent this condition of deadlocks, when it is possible.
  + One approach is that if a process is forced to wait when requesting a new resource, then all other resources previously held by this process are implicitly released, ( preempted ), forcing this process to re-acquire the old resources along with the new resources in a single request, similar to the previous discussion.
  + Another approach is that when a resource is requested and not available, then the system looks to see what other processes currently have those resources *and* are themselves blocked waiting for some other resource. If such a process is found, then some of their resources may get preempted and added to the list of resources for which the process is waiting.
  + Either of these approaches may be applicable for resources whose states are easily saved and restored, such as registers and memory, but are generally not applicable to other devices such as printers and tape drives.

**7.4.4 Circular Wait**

* One way to avoid circular wait is to number all resources, and to require that processes request resources only in strictly increasing ( or decreasing ) order.
* In other words, in order to request resource Rj, a process must first release all Ri such that i >= j.
* One big challenge in this scheme is determining the relative ordering of the different resources

**Bankers algorithm for DEADLOCK AVOIDANCE with an example.**

The Banker's Algorithm gets its name because it is a method that bankers could use to assure that when they lend out resources they will still be able to satisfy all their clients.

* When a process starts up, it must state in advance the maximum allocation of resources it may request, up to the amount available on the system.
* When a request is made, the scheduler determines whether granting the request would leave the system in a safe state. If not, then the process must wait until the request can be granted safely.
* The banker's algorithm relies on several key data structures: ( where n is the number of processes and m is the number of resource categories. )
  + Available[ m ] indicates how many resources are currently available of each type.
  + Max[ n ][ m ] indicates the maximum demand of each process of each resource.
  + Allocation[ n ][ m ] indicates the number of each resource category allocated to each process.
  + Need[ n ][ m ] indicates the remaining resources needed of each type for each process. ( Note that Need[ i ][ j ] = Max[ i ][ j ] - Allocation[ i ][ j ] for all i, j. )
* For simplification of discussions, we make the following notations / observations:
  + One row of the Need vector, Need[ i ], can be treated as a vector corresponding to the needs of process i, and similarly for Allocation and Max.
  + A vector X is considered to be <= a vector Y if X[ i ] <= Y[ i ] for all i.

**7.5.3.1 Safety Algorithm**

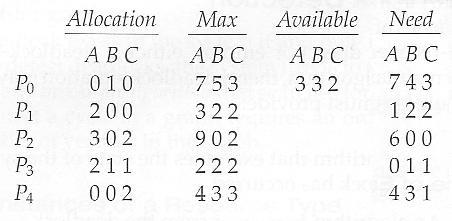
* In order to apply the Banker's algorithm, we first need an algorithm for determining whether or not a particular state is safe.
* This algorithm determines if the current state of a system is safe, according to the following steps:
  1. Let Work and Finish be vectors of length m and n respectively.
     + Work is a working copy of the available resources, which will be modified during the analysis.
     + Finish is a vector of booleans indicating whether a particular process can finish. ( or has finished so far in the analysis. )
     + Initialize Work to Available, and Finish to false for all elements.
  2. Find an i such that both (A) Finish[ i ] == false, and (B) Need[ i ] < Work. This process has not finished, but could with the given available working set. If no such i exists, go to step 4.
  3. Set Work = Work + Allocation[ i ], and set Finish[ i ] to true. This corresponds to process i finishing up and releasing its resources back into the work pool. Then loop back to step 2.
  4. If finish[ i ] == true for all i, then the state is a safe state, because a safe sequence has been found.
* JTB's Modification:
  1. In step 1. instead of making Finish an array of booleans initialized to false, make it an array of ints initialized to 0. Also initialize an int s = 0 as a step counter.
  2. In step 2, look for Finish[ i ] == 0.
  3. In step 3, set Finish[ i ] to ++s. S is counting the number of finished processes.
  4. For step 4, the test can be either Finish[ i ] > 0 for all i, or s >= n. The benefit of this method is that if a safe state exists, then Finish[ ] indicates one safe sequence ( of possibly many. ) )

**7.5.3.2 Resource-Request Algorithm ( The Bankers Algorithm )**

* Now that we have a tool for determining if a particular state is safe or not, we are now ready to look at the Banker's algorithm itself.
* This algorithm determines if a new request is safe, and grants it only if it is safe to do so.
* When a request is made ( that does not exceed currently available resources ), pretend it has been granted, and then see if the resulting state is a safe one. If so, grant the request, and if not, deny the request, as follows:
  1. Let Request[ n ][ m ] indicate the number of resources of each type currently requested by processes. If Request[ i ] > Need[ i ] for any process i, raise an error condition.
  2. If Request[ i ] > Available for any process i, then that process must wait for resources to become available. Otherwise the process can continue to step 3.
  3. Check to see if the request can be granted safely, by pretending it has been granted and then seeing if the resulting state is safe. If so, grant the request, and if not, then the process must wait until its request can be granted safely.The procedure for granting a request ( or pretending to for testing purposes ) is:
     + Available = Available - Request
     + Allocation = Allocation + Request
     + Need = Need - Request

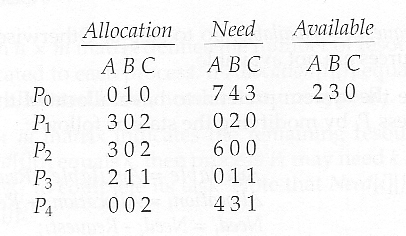
**An Illustrative Example**

* Consider the following situation:



* And now consider what happens if process P1 requests 1 instance of A and 2 instances of C.

( Request[ 1 ] = ( 1, 0, 2 ) )

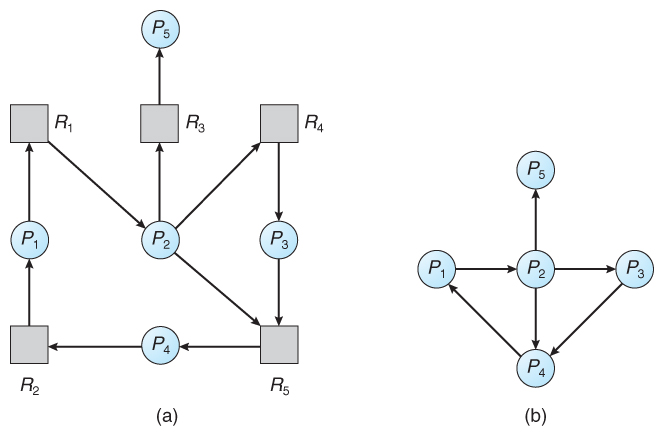


### DEADLOCK DETECTION

* If deadlocks are not avoided, then another approach is to detect when they have occurred and recover somehow.
* In addition to the performance hit of constantly checking for deadlocks, a policy / algorithm must be in place for recovering from deadlocks, and there is potential for lost work when processes must be aborted or have their resources preempted.

#### **7.6.1 Single Instance of Each Resource Type**

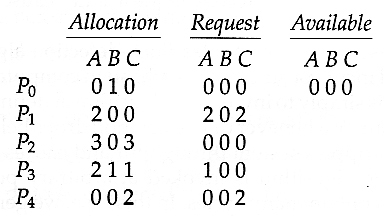
* If each resource category has a single instance, then we can use a variation of the resource-allocation graph known as a ***wait-for graph***.
* A wait-for graph can be constructed from a resource-allocation graph by eliminating the resources and collapsing the associated edges, as shown in the figure below.
* An arc from Pi to Pj in a wait-for graph indicates that process Pi is waiting for a resource that process Pj is currently holding.

  
**Figure 7.9 - (a) Resource allocation graph. (b) Corresponding wait-for graph**

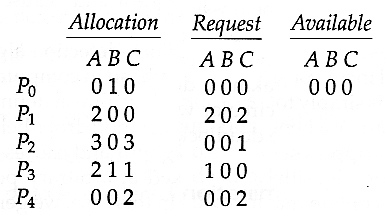
* As before, cycles in the wait-for graph indicate deadlocks.
* This algorithm must maintain the wait-for graph, and periodically search it for cycles.

#### **7.6.2 Several Instances of a Resource Type**

* The detection algorithm outlined here is essentially the same as the Banker's algorithm, with two subtle differences:
  + In step 1, the Banker's Algorithm sets Finish[ i ] to false for all i. The algorithm presented here sets Finish[ i ] to false only if Allocation[ i ] is not zero. If the currently allocated resources for this process are zero, the algorithm sets Finish[ i ] to true. This is essentially assuming that IF all of the other processes can finish, then this process can finish also. Furthermore, this algorithm is specifically looking for which processes are involved in a deadlock situation, and a process that does not have any resources allocated cannot be involved in a deadlock, and so can be removed from any further consideration.
  + Steps 2 and 3 are unchanged
  + In step 4, the basic Banker's Algorithm says that if Finish[ i ] == true for all i, that there is no deadlock. This algorithm is more specific, by stating that if Finish[ i ] == false for any process Pi, then that process is specifically involved in the deadlock which has been detected.
* ( Note: An alternative method was presented above, in which Finish held integers instead of booleans. This vector would be initialized to all zeros, and then filled with increasing integers as processes are detected which can finish. If any processes are left at zero when the algorithm completes, then there is a deadlock, and if not, then the integers in finish describe a safe sequence. To modify this algorithm to match this section of the text, processes with allocation = zero could be filled in with N, N - 1, N - 2, etc. in step 1, and any processes left with Finish = 0 in step 4 are the deadlocked processes. )
* Consider, for example, the following state, and determine if it is currently deadlocked:



* Now suppose that process P2 makes a request for an additional instance of type C, yielding the state shown below. Is the system now deadlocked?



#### 7.6.3 Detection-Algorithm Usage

* When should the deadlock detection be done? Frequently, or infrequently?
* The answer may depend on how frequently deadlocks are expected to occur, as well as the possible consequences of not catching them immediately. ( If deadlocks are not removed immediately when they occur, then more and more processes can "back up" behind the deadlock, making the eventual task of unblocking the system more difficult and possibly damaging to more processes. )
* There are two obvious approaches, each with trade-offs:
  1. Do deadlock detection after every resource allocation which cannot be immediately granted. This has the advantage of detecting the deadlock right away, while the minimum number of processes are involved in the deadlock. ( One might consider that the process whose request triggered the deadlock condition is the "cause" of the deadlock, but realistically all of the processes in the cycle are equally responsible for the resulting deadlock. ) The down side of this approach is the extensive overhead and performance hit caused by checking for deadlocks so frequently.
  2. Do deadlock detection only when there is some clue that a deadlock may have occurred, such as when CPU utilization reduces to 40% or some other magic number. The advantage is that deadlock detection is done much less frequently, but the down side is that it becomes impossible to detect the processes involved in the original deadlock, and so deadlock recovery can be more complicated and damaging to more processes.
  3. ( As I write this, a third alternative comes to mind: Keep a historical log of resource allocations, since that last known time of no deadlocks. Do deadlock checks periodically ( once an hour or when CPU usage is low?), and then use the historical log to trace through and determine when the deadlock occurred and what processes caused the initial deadlock. Unfortunately I'm not certain that breaking the original deadlock would then free up the resulting log jam. )

### 7.7 RECOVERY From DEADLOCK

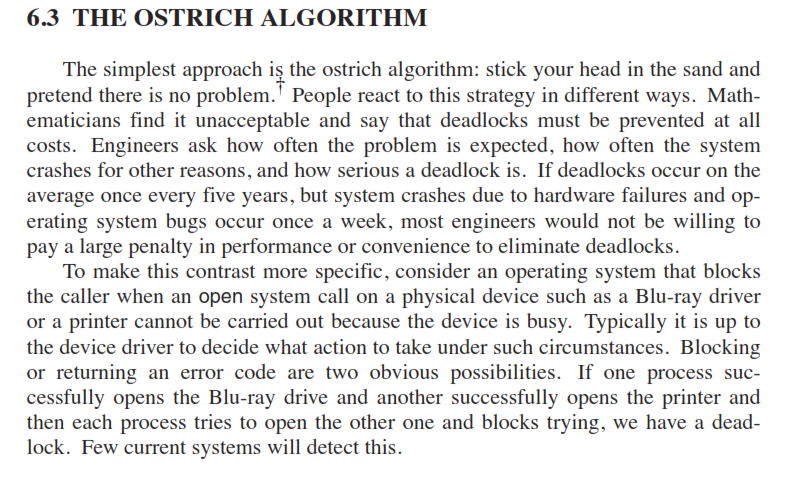
* There are three basic approaches to recovery from deadlock:
  1. Inform the system operator, and allow him/her to take manual intervention.
  2. Terminate one or more processes involved in the deadlock
  3. Preempt resources.

#### 7.7.1 Process Termination

* Two basic approaches, both of which recover resources allocated to terminated processes:
  + Terminate all processes involved in the deadlock. This definitely solves the deadlock, but at the expense of terminating more processes than would be absolutely necessary.
  + Terminate processes one by one until the deadlock is broken. This is more conservative, but requires doing deadlock detection after each step.
* In the latter case there are many factors that can go into deciding which processes to terminate next:
  + Process priorities.
  + How long the process has been running, and how close it is to finishing.
  + How many and what type of resources is the process holding. ( Are they easy to preempt and restore? )
  + How many more resources does the process need to complete.
  + How many processes will need to be terminated
  + Whether the process is interactive or batch.
  + ( Whether or not the process has made non-restorable changes to any resource. )

#### 7.7.2 Resource Preemption

* When preempting resources to relieve deadlock, there are three important issues to be addressed:
  1. **Selecting a victim**- Deciding which resources to preempt from which processes involves many of the same decision criteria outlined above.
  2. **Rollback** - Ideally one would like to roll back a preempted process to a safe state prior to the point at which that resource was originally allocated to the process. Unfortunately it can be difficult or impossible to determine what such a safe state is, and so the only safe rollback is to roll back all the way back to the beginning. ( I.e. abort the process and make it start over. )
  3. **Starvation**- How do you guarantee that a process won't starve because its resources are constantly being preempted? One option would be to use a priority system, and increase the priority of a process every time its resources get preempted. Eventually it should get a high enough priority that it won't get preempted any more.



**UNIT-4 [part-2]**

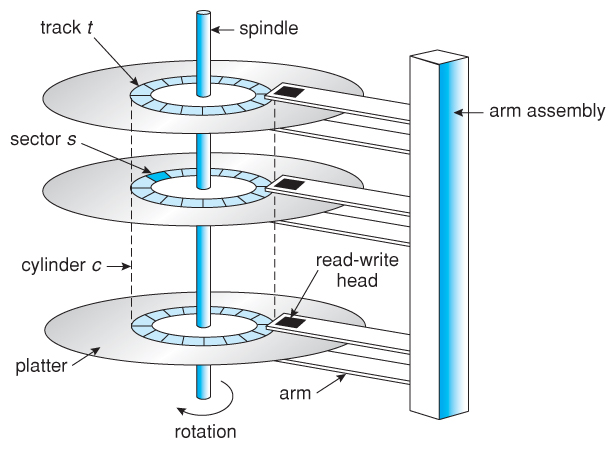
[Secondary-Storage Structure: Overview of disk structure, and attachment, Disk scheduling,

RAID structure, Stable storage implementation.]

**Overview of Mass-Storage Structure**

**10.1.1 Magnetic Disks**

* Traditional magnetic disks have the following basic structure:
  + One or more ***platters***in the form of disks covered with magnetic media. ***Hard disk*** platters are made of rigid metal, while "***floppy***" disks are made of more flexible plastic.
  + Each platter has two working ***surfaces.***Older hard disk drives would sometimes not use the very top or bottom surface of a stack of platters, as these surfaces were more susceptible to potential damage.
  + Each working surface is divided into a number of concentric rings called ***tracks.***The collection of all tracks that are the same distance from the edge of the platter, ( i.e. all tracks immediately above one another in the following diagram ) is called a ***cylinder***.
  + Each track is further divided into ***sectors,*** traditionally containing 512 bytes of data each, although some modern disks occasionally use larger sector sizes. ( Sectors also include a header and a trailer, including checksum information among other things. Larger sector sizes reduce the fraction of the disk consumed by headers and trailers, but increase internal fragmentation and the amount of disk that must be marked bad in the case of errors. )
  + The data on a hard drive is read by read-write ***heads.***The standard configuration ( shown below ) uses one head per surface, each on a separate ***arm***, and controlled by a common ***arm assembly*** which moves all heads simultaneously from one cylinder to another. ( Other configurations, including independent read-write heads, may speed up disk access, but involve serious technical difficulties. )
  + The storage capacity of a traditional disk drive is equal to the number of heads ( i.e. the number of working surfaces ), times the number of tracks per surface, times the number of sectors per track, times the number of bytes per sector. A particular physical block of data is specified by providing the head-sector-cylinder number at which it is located.

  
**Figure 10.1 - Moving-head disk mechanism.**

* In operation the disk rotates at high speed, such as 7200 rpm ( 120 revolutions per second. ) The rate at which data can be transferred from the disk to the computer is composed of several steps:
  + The ***positioning time***, a.k.a. the ***seek time***or***random access time*** is the time required to move the heads from one cylinder to another, and for the heads to settle down after the move. This is typically the slowest step in the process and the predominant bottleneck to overall transfer rates.
  + The ***rotational latency*** is the amount of time required for the desired sector to rotate around and come under the read-write head.This can range anywhere from zero to one full revolution, and on the average will equal one-half revolution. This is another physical step and is usually the second slowest step behind seek time. ( For a disk rotating at 7200 rpm, the average rotational latency would be 1/2 revolution / 120 revolutions per second, or just over 4 milliseconds, a long time by computer standards.
  + The ***transfer rate***, which is the time required to move the data electronically from the disk to the computer. ( Some authors may also use the term transfer rate to refer to the overall transfer rate, including seek time and rotational latency as well as the electronic data transfer rate. )
* Disk heads "fly" over the surface on a very thin cushion of air. If they should accidentally contact the disk, then a ***head crash*** occurs, which may or may not permanently damage the disk or even destroy it completely. For this reason it is normal to ***park*** the disk heads when turning a computer off, which means to move the heads off the disk or to an area of the disk where there is no data stored.
* Floppy disks are normally ***removable***. Hard drives can also be removable, and some are even ***hot-swappable***, meaning they can be removed while the computer is running, and a new hard drive inserted in their place.
* Disk drives are connected to the computer via a cable known as the ***I/O Bus.*** Some of the common interface formats include Enhanced Integrated Drive Electronics, EIDE; Advanced Technology Attachment, ATA; Serial ATA, SATA, Universal Serial Bus, USB; Fiber Channel, FC, and Small Computer Systems Interface, SCSI.
* The ***host controller*** is at the computer end of the I/O bus, and the ***disk controller*** is built into the disk itself. The CPU issues commands to the host controller via I/O ports. Data is transferred between the magnetic surface and onboard ***cache*** by the disk controller, and then the data is transferred from that cache to the host controller and the motherboard memory at electronic speeds.

**10.1.2 Solid-State Disks - New**

* As technologies improve and economics change, old technologies are often used in different ways. One example of this is the increasing used of ***solid state disks, or SSDs.***
* SSDs use memory technology as a small fast hard disk. Specific implementations may use either flash memory or DRAM chips protected by a battery to sustain the information through power cycles.
* Because SSDs have no moving parts they are much faster than traditional hard drives, and certain problems such as the scheduling of disk accesses simply do not apply.
* However SSDs also have their weaknesses: They are more expensive than hard drives, generally not as large, and may have shorter life spans.
* SSDs are especially useful as a high-speed cache of hard-disk information that must be accessed quickly. One example is to store filesystem meta-data, e.g. directory and inode information, that must be accessed quickly and often. Another variation is a boot disk containing the OS and some application executables, but no vital user data. SSDs are also used in laptops to make them smaller, faster, and lighter.
* Because SSDs are so much faster than traditional hard disks, the throughput of the bus can become a limiting factor, causing some SSDs to be connected directly to the system PCI bus for example.

**10.1.3 Magnetic Tapes - was 12.1.2**

* Magnetic tapes were once used for common secondary storage before the days of hard disk drives, but today are used primarily for backups.
* Accessing a particular spot on a magnetic tape can be slow, but once reading or writing commences, access speeds are comparable to disk drives.
* Capacities of tape drives can range from 20 to 200 GB, and compression can double that capacity.

**10.2 Over view of Disk Structure**

* The traditional head-sector-cylinder, HSC numbers are mapped to linear block addresses by numbering the first sector on the first head on the outermost track as sector 0. Numbering proceeds with the rest of the sectors on that same track, and then the rest of the tracks on the same cylinder before proceeding through the rest of the cylinders to the center of the disk. In modern practice these linear block addresses are used in place of the HSC numbers for a variety of reasons:
  1. The linear length of tracks near the outer edge of the disk is much longer than for those tracks located near the center, and therefore it is possible to squeeze many more sectors onto outer tracks than onto inner ones.
  2. All disks have some bad sectors, and therefore disks maintain a few spare sectors that can be used in place of the bad ones. The mapping of spare sectors to bad sectors in managed internally to the disk controller.
  3. Modern hard drives can have thousands of cylinders, and hundreds of sectors per track on their outermost tracks. These numbers exceed the range of HSC numbers for many ( older ) operating systems, and therefore disks can be configured for any convenient combination of HSC values that falls within the total number of sectors physically on the drive.
* There is a limit to how closely packed individual bits can be placed on a physical media, but that limit is growing increasingly more packed as technological advances are made.
* Modern disks pack many more sectors into outer cylinders than inner ones, using one of two approaches:
  1. With ***Constant Linear Velocity, CLV,***the density of bits is uniform from cylinder to cylinder. Because there are more sectors in outer cylinders, the disk spins slower when reading those cylinders, causing the rate of bits passing under the read-write head to remain constant. This is the approach used by modern CDs and DVDs.
  2. With ***Constant Angular Velocity, CAV,***the disk rotates at a constant angular speed, with the bit density decreasing on outer cylinders. ( These disks would have a constant number of sectors per track on all cylinders. )

**10.3 Disk Attachment**

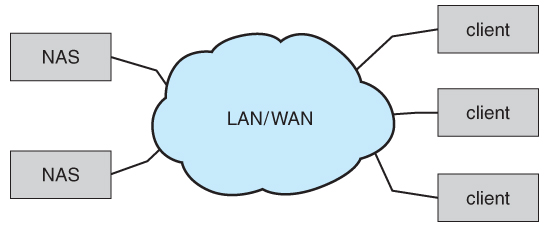
Disk drives can be attached either directly to a particular host ( a local disk ) or to a network.

**10.3.1 Host-Attached Storage**

* Local disks are accessed through I/O Ports as described earlier.
* The most common interfaces are IDE or ATA, each of which allow up to two drives per host controller.
* SATA is similar with simpler cabling.
* High end workstations or other systems in need of larger number of disks typically use SCSI disks:
  + The SCSI standard supports up to 16 *targets* on each SCSI bus, one of which is generally the host adapter and the other 15 of which can be disk or tape drives.
  + A SCSI target is usually a single drive, but the standard also supports up to 8 *units* within each target. These would generally be used for accessing individual disks within a RAID array. ( See below. )
  + The SCSI standard also supports multiple host adapters in a single computer, i.e. multiple SCSI busses.
  + Modern advancements in SCSI include "fast" and "wide" versions, as well as SCSI-2.
  + SCSI cables may be either 50 or 68 conductors. SCSI devices may be external as well as internal.
  + See [wikipedia](http://en.wikipedia.org/wiki/SCSI) for more information on the SCSI interface.
* FC is a high-speed serial architecture that can operate over optical fiber or four-conductor copper wires, and has two variants:
  + A large switched fabric having a 24-bit address space. This variant allows for multiple devices and multiple hosts to interconnect, forming the basis for the *storage-area networks, SANs,*to be discussed in a future section.
  + The *arbitrated loop, FC-AL,*that can address up to 126 devices ( drives and controllers. )

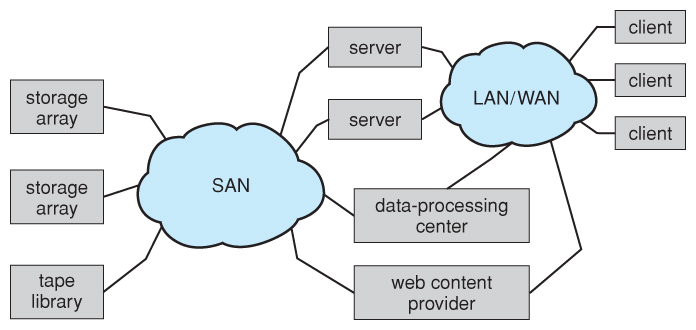
**10.3.2 Network-Attached Storage**

* Network attached storage connects storage devices to computers using a remote procedure call, RPC, interface, typically with something like NFS filesystem mounts. This is convenient for allowing several computers in a group common access and naming conventions for shared storage.
* NAS can be implemented using SCSI cabling, or*ISCSI*uses Internet protocols and standard network connections, allowing long-distance remote access to shared files.
* NAS allows computers to easily share data storage, but tends to be less efficient than standard host-attached storage.

  
Figure 10.2 - Network-attached storage.

**10.3.3 Storage-Area Network**

* A *Storage-Area Network, SAN,*connects computers and storage devices in a network, using storage protocols instead of network protocols.
* One advantage of this is that storage access does not tie up regular networking bandwidth.
* SAN is very flexible and dynamic, allowing hosts and devices to attach and detach on the fly.
* SAN is also controllable, allowing restricted access to certain hosts and devices.

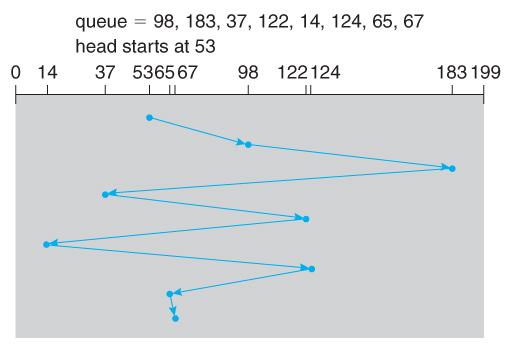
  
Figure 10.3 - Storage-area network.

**10.4 Disk Scheduling**

* As mentioned earlier, disk transfer speeds are limited primarily by ***seek times*** and ***rotational latency.*** When multiple requests are to be processed there is also some inherent delay in waiting for other requests to be processed.
* ***Bandwidth*** is measured by the amount of data transferred divided by the total amount of time from the first request being made to the last transfer being completed, ( for a series of disk requests. )
* Both bandwidth and access time can be improved by processing requests in a good order.
* Disk requests include the disk address, memory address, number of sectors to transfer, and whether the request is for reading or writing.

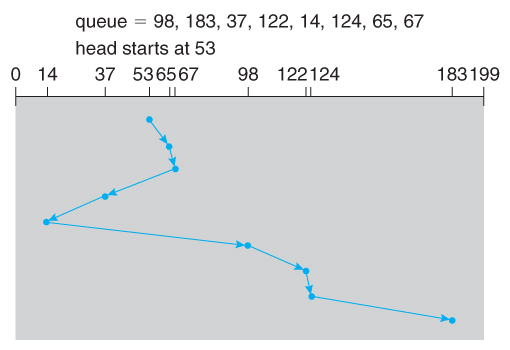
**10.4.1 FCFS Scheduling**

* ***First-Come First-Serve*** is simple and intrinsically fair, but not very efficient. Consider in the following sequence the wild swing from cylinder 122 to 14 and then back to 124:

  
**Figure 10.4 - FCFS disk scheduling.**

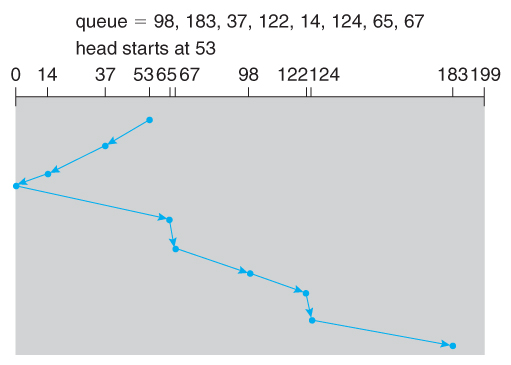
**10.4.2 SSTF Scheduling**

* ***Shortest Seek Time First*** scheduling is more efficient, but may lead to starvation if a constant stream of requests arrives for the same general area of the disk.
* SSTF reduces the total head movement to 236 cylinders, down from 640 required for the same set of requests under FCFS. Note, however that the distance could be reduced still further to 208 by starting with 37 and then 14 first before processing the rest of the requests.

  
**Figure 10.5 - SSTF disk scheduling.**

**10.4.3 SCAN Scheduling**

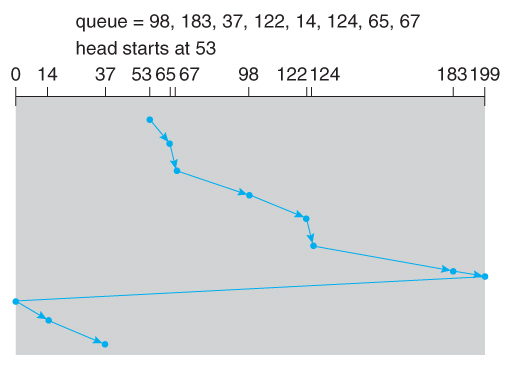
* The ***SCAN*** algorithm, a.k.a. the ***elevator***algorithm moves back and forth from one end of the disk to the other, similarly to an elevator processing requests in a tall building.

  
**Figure 10.6 - SCAN disk scheduling.**

* Under the SCAN algorithm, If a request arrives just ahead of the moving head then it will be processed right away, but if it arrives just after the head has passed, then it will have to wait for the head to pass going the other way on the return trip. This leads to a fairly wide variation in access times which can be improved upon.
* Consider, for example, when the head reaches the high end of the disk: Requests with high cylinder numbers just missed the passing head, which means they are all fairly recent requests, whereas requests with low numbers may have been waiting for a much longer time. Making the return scan from high to low then ends up accessing recent requests first and making older requests wait that much longer.

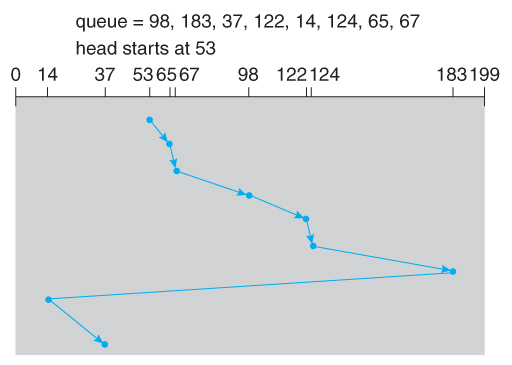
**10.4.4 C-SCAN Scheduling**

* The ***Circular-SCAN*** algorithm improves upon SCAN by treating all requests in a circular queue fashion - Once the head reaches the end of the disk, it returns to the other end without processing any requests, and then starts again from the beginning of the disk:

  
**Figure 10.7 - C-SCAN disk scheduling.**

**12.4.5 LOOK Scheduling**

* ***LOOK*** scheduling improves upon SCAN by looking ahead at the queue of pending requests, and not moving the heads any farther towards the end of the disk than is necessary. The following diagram illustrates the circular form of LOOK:

  
**Figure 10.8 - C-LOOK disk scheduling.**

**10.4.6 Selection of a Disk-Scheduling Algorithm**

* With very low loads all algorithms are equal, since there will normally only be one request to process at a time.
* For slightly larger loads, SSTF offers better performance than FCFS, but may lead to starvation when loads become heavy enough.
* For busier systems, SCAN and LOOK algorithms eliminate starvation problems.
* The actual optimal algorithm may be something even more complex than those discussed here, but the incremental improvements are generally not worth the additional overhead.
* Some improvement to overall filesystem access times can be made by intelligent placement of directory and/or inode information. If those structures are placed in the middle of the disk instead of at the beginning of the disk, then the maximum distance from those structures to data blocks is reduced to only one-half of the disk size. If those structures can be further distributed and furthermore have their data blocks stored as close as possible to the corresponding directory structures, then that reduces still further the overall time to find the disk block numbers and then access the corresponding data blocks.
* On modern disks the rotational latency can be almost as significant as the seek time, however it is not within the OSes control to account for that, because modern disks do not reveal their internal sector mapping schemes, ( particularly when bad blocks have been remapped to spare sectors. )
  + Some disk manufacturers provide for disk scheduling algorithms directly on their disk controllers, ( which do know the actual geometry of the disk as well as any remapping ), so that if a series of requests are sent from the computer to the controller then those requests can be processed in an optimal order.

### Unfortunately there are some considerations that the OS must take into account that are beyond the abilities of the on-board disk-scheduling algorithms, such as priorities of some requests over others, or the need to process certain requests in a particular order. For this reason OSes may elect to spoon-feed requests to the disk controller one at a time in certain situations.

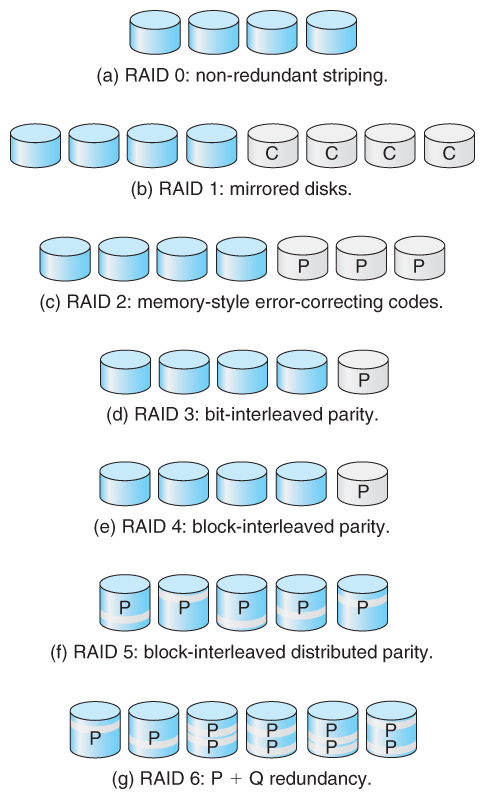
**RAID**

**RAID definition.**

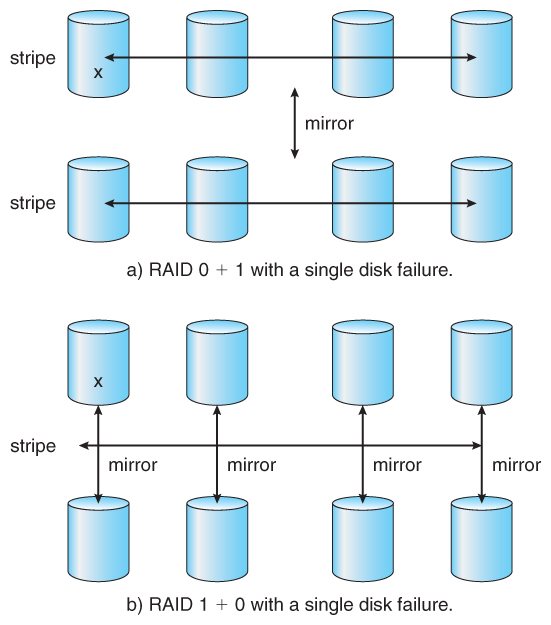
* The general idea behind RAID is to employ a group of hard drives together with some form of duplication, either to increase reliability or to speed up operations, ( or sometimes both. )
* *RAID* originally stood for *Redundant Array of Inexpensive Disks,* and was designed to use a bunch of cheap small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

**RAID Levels**

* Mirroring provides reliability but is expensive; Striping improves performance, but does not improve reliability. Accordingly there are a number of different schemes that combine the principals of mirroring and striping in different ways, in order to balance reliability versus performance versus cost. These are described by different *RAID levels*, as follows: ( In the diagram that follows, "C" indicates a copy, and "P" indicates parity, i.e. checksum bits. )
  1. ***Raid Level 0*** *-*This level includes striping only, with no mirroring.
  2. ***Raid Level 1*** *-*This level includes mirroring only, no striping.
  3. ***Raid Level 2*** *-*This level stores error-correcting codes on additional disks, allowing for any damaged data to be reconstructed by subtraction from the remaining undamaged data. Note that this scheme requires only three extra disks to protect 4 disks worth of data, as opposed to full mirroring. ( The number of disks required is a function of the error-correcting algorithms, and the means by which the particular bad bit(s) is(are) identified. )
  4. ***Raid Level 3*** *-*This level is similar to level 2, except that it takes advantage of the fact that each disk is still doing its own error-detection, so that when an error occurs, there is no question about which disk in the array has the bad data. As a result a single parity bit is all that is needed to recover the lost data from an array of disks. Level 3 also includes striping, which improves performance. The downside with the parity approach is that every disk must take part in every disk access, and the parity bits must be constantly calculated and checked, reducing performance. Hardware-level parity calculations and NVRAM cache can help with both of those issues. In practice level 3 is greatly preferred over level 2.
  5. ***Raid Level 4*** *-*This level is similar to level 3, employing block-level striping instead of bit-level striping. The benefits are that multiple blocks can be read independently, and changes to a block only require writing two blocks ( data and parity ) rather than involving all disks. Note that new disks can be added seamlessly to the system provided they are initialized to all zeros, as this does not affect the parity results.
  6. ***Raid Level 5*** *-*This level is similar to level 4, except the parity blocks are distributed over all disks, thereby more evenly balancing the load on the system. For any given block on the disk(s), one of the disks will hold the parity information for that block and the other N-1 disks will hold the data. Note that the same disk cannot hold both data and parity for the same block, as both would be lost in the event of a disk crash.
  7. ***Raid Level 6*** *-*This level extends raid level 5 by storing multiple bits of error-recovery codes, ( such as the [*Reed-Solomon codes*](http://en.wikipedia.org/wiki/Reed-Solomon_coding) ), for each bit position of data, rather than a single parity bit. In the example shown below 2 bits of ECC are stored for every 4 bits of data, allowing data recovery in the face of up to two simultaneous disk failures. Note that this still involves only 50% increase in storage needs, as opposed to 100% for simple mirroring which could only tolerate a single disk failure.

  
Figure 10.11 - RAID levels.

* There are also two RAID levels which combine RAID levels 0 and 1 ( striping and mirroring ) in different combinations, designed to provide both performance and reliability at the expense of increased cost.
  + RAID level 0 + 1 disks are first striped, and then the striped disks mirrored to another set. This level generally provides better performance than RAID level 5.
  + RAID level 1 + 0 mirrors disks in pairs, and then stripes the mirrored pairs. The storage capacity, performance, etc. are all the same, but there is an advantage to this approach in the event of multiple disk failures, as illustrated below:.
    - In diagram (a) below, the 8 disks have been divided into two sets of four, each of which is striped, and then one stripe set is used to mirror the other set.
      * If a single disk fails, it wipes out the entire stripe set, but the system can keep on functioning using the remaining set.
      * However if a second disk from the other stripe set now fails, then the entire system is lost, as a result of two disk failures.
    - In diagram (b), the same 8 disks are divided into four sets of two, each of which is mirrored, and then the file system is striped across the four sets of mirrored disks.
      * If a single disk fails, then that mirror set is reduced to a single disk, but the system rolls on, and the other three mirror sets continue mirroring.
      * Now if a second disk fails, ( that is not the mirror of the already failed disk ), then another one of the mirror sets is reduced to a single disk, but the system can continue without data loss.
      * In fact the second arrangement could handle as many as four simultaneously failed disks, as long as no two of them were from the same mirror pair.

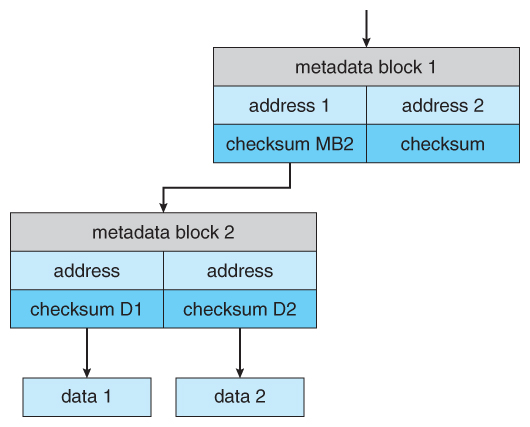
  
Figure 10.12 - RAID 0 + 1 and 1 + 0

10.7.4 Selecting a RAID Level

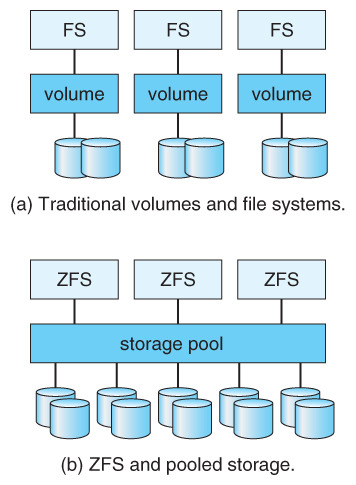
* Trade-offs in selecting the optimal RAID level for a particular application include cost, volume of data, need for reliability, need for performance, and rebuild time, the latter of which can affect the likelihood that a second disk will fail while the first failed disk is being rebuilt.
* Other decisions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

**10.7.5 Problems with RAID**

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

  
Figure 10.13 - ZFS checksums all metadata and data.

* Another problem with traditional filesystems is that the sizes are fixed, and relatively difficult to change. Where RAID sets are involved it becomes even harder to adjust filesystem sizes, because a filesystem cannot span across multiple filesystems.
* ZFS solves these problems by pooling RAID sets, and by dynamically allocating space to filesystems as needed. Filesystem sizes can be limited by quotas, and space can also be reserved to guarantee that a filesystem will be able to grow later, but these parameters can be changed at any time by the filesystem's owner. Otherwise filesystems grow and shrink dynamically as needed.

**  
Figure 10.14 - (a) Traditional volumes and file systems. (b) a ZFS pool and file systems.**

### STABLE-STORAGE IMPLEMENTATION

* The concept of stable storage ( first presented in chapter 6 ) involves a storage medium in which data is ***never*** lost, even in the face of equipment failure in the middle of a write operation.
* To implement this requires two ( or more ) copies of the data, with separate failure modes.
* An attempted disk write results in one of three possible outcomes:
  1. The data is successfully and completely written.
  2. The data is partially written, but not completely. The last block written may be garbled.
  3. No writing takes place at all.
* Whenever an equipment failure occurs during a write, the system must detect it, and return the system back to a consistent state. To do this requires two physical blocks for every logical block, and the following procedure:
  1. Write the data to the first physical block.
  2. After step 1 had completed, then write the data to the second physical block.
  3. Declare the operation complete only after both physical writes have completed successfully.
* During recovery the pair of blocks is examined.
  1. If both blocks are identical and there is no sign of damage, then no further action is necessary.
  2. If one block contains a detectable error but the other does not, then the damaged block is replaced with the good copy. ( This will either undo the operation or complete the operation, depending on which block is damaged and which is undamaged. )
  3. If neither block shows damage but the data in the blocks differ, then replace the data in the first block with the data in the second block. ( Undo the operation. )
* Because the sequence of operations described above is slow, stable storage usually includes NVRAM as a cache, and declares a write operation complete once it has been written to the NVRAM.