# MODULE 4

## Mass-Storage Structure

### Overview of Mass-Storage Structure

### 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.

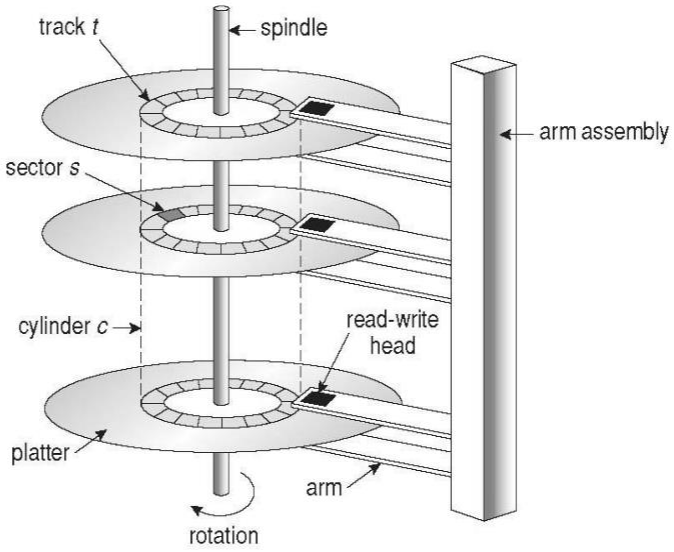



Fig: Moving-head disk mechanism

- 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 seeks 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.

## 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 use 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 file system meta-data, e.g. directory and anode 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.



## Magnetic Tapes

- 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.

## Disk Structure

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



## Disk Attachment

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

### 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.
- 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.)

## Network-Attached Storage

- Network attached storage connects storage devices to computers using a remote procedure call, RPC, interface, typically with something like NFS file system 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.

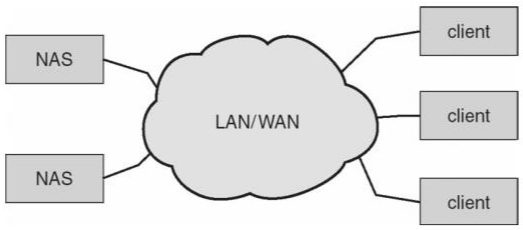


Fig: Network-attached storage.

## 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.

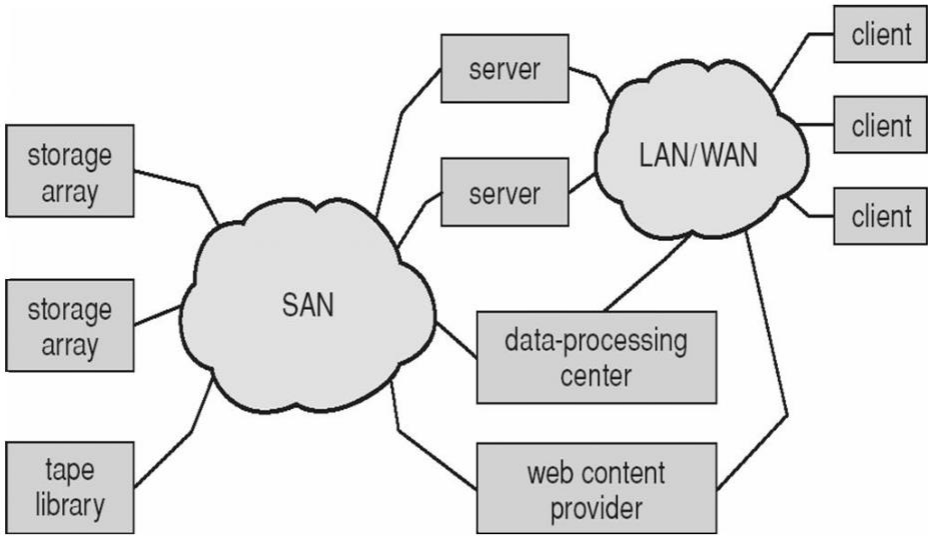



Fig: Storage-area network.

# 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.

## 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:

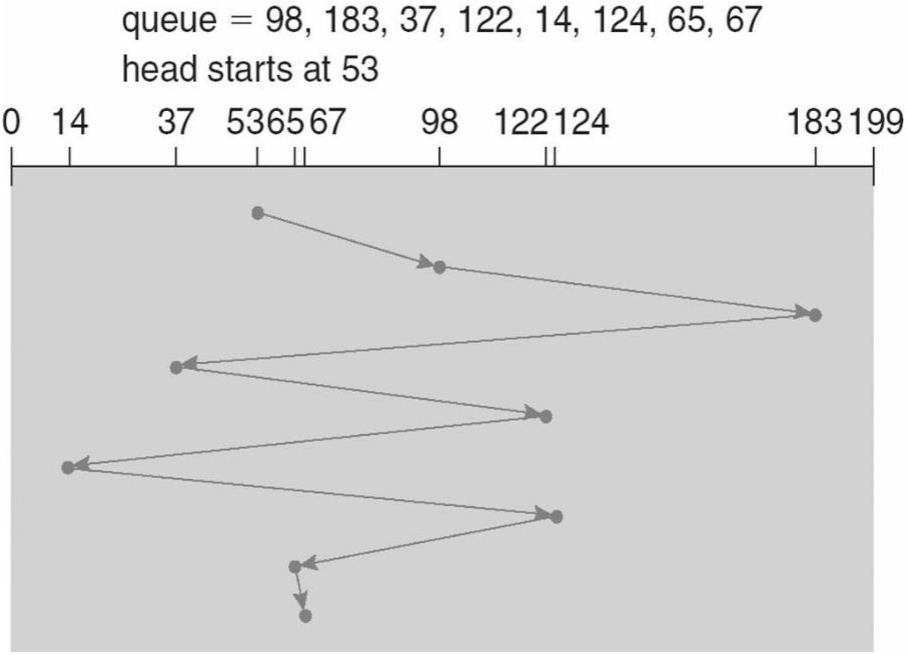



Fig: FCFS disk scheduling.

## 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.

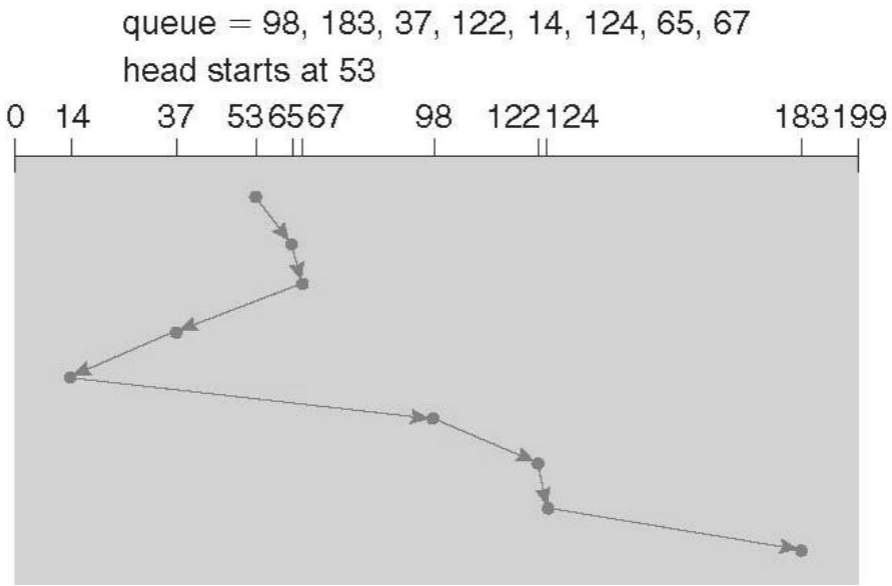



Fig: SSTF disk scheduling.

## 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.

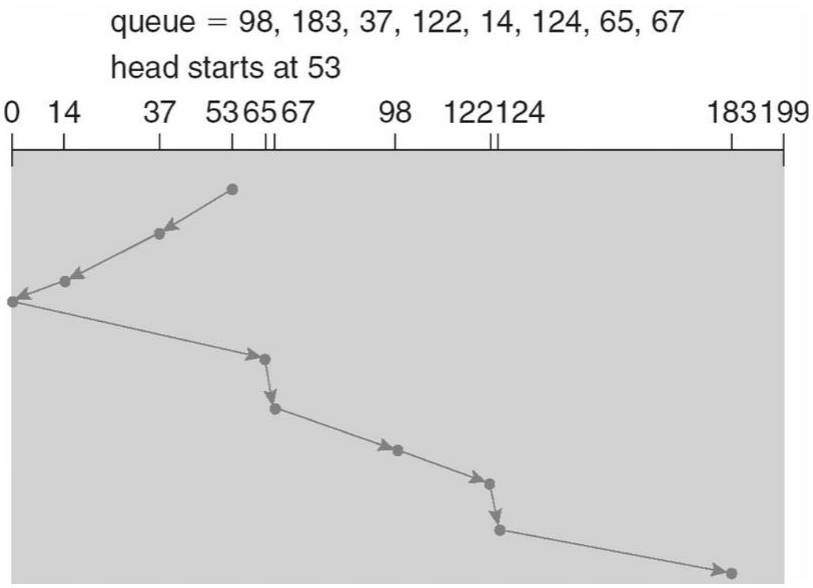



Fig: SCAN disk scheduling.

- 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.

## 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:

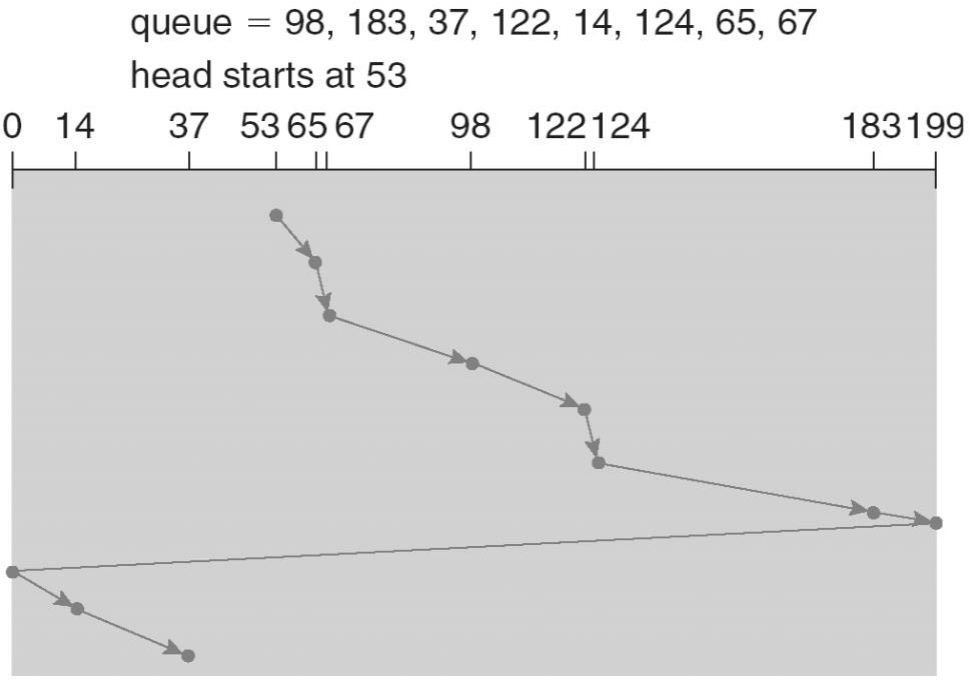



Fig: C-SCAN disk scheduling.

## 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:

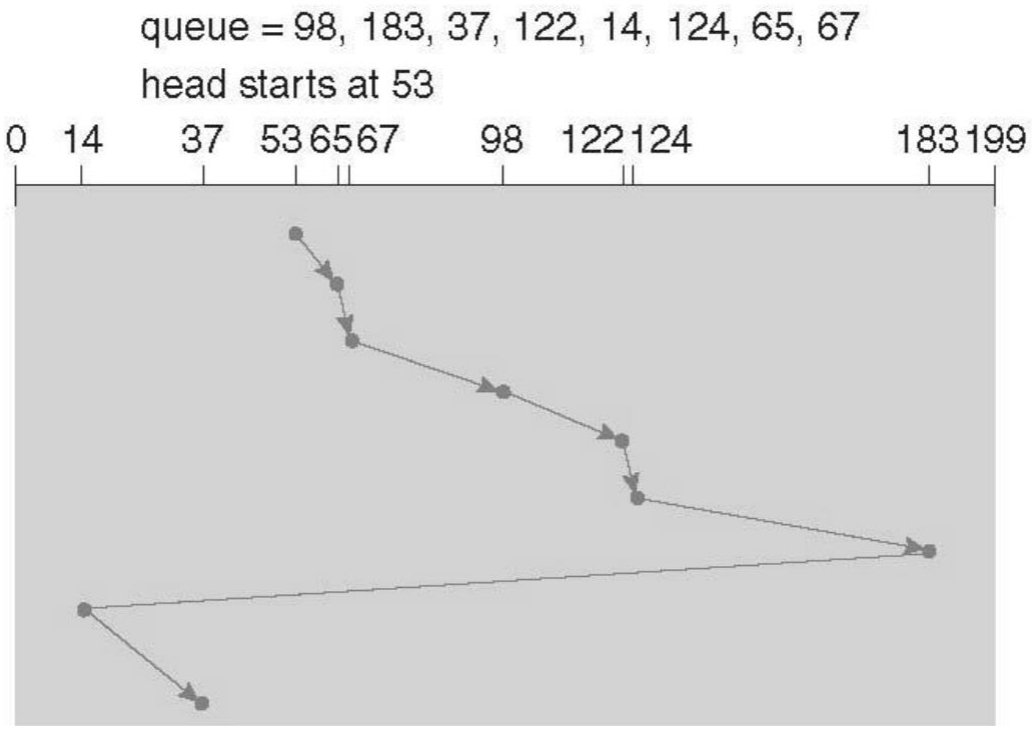



Fig: C-LOOK disk scheduling.

### 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 file system 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 they 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.



## Disk Management

### Disk Formatting

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

- ECC calculation is performed with every disk read or write, and if damage is detected but the data is recoverable, then a soft error has occurred. Soft errors are generally handled by the on-board disk controller, and never seen by the OS. (See below.)
- Once the disk is low-level formatted, the next step is to partition the drive into one or more separate partitions. This step must be completed even if the disk is to be used as a single large partition, so that the partition table can be written to the beginning of the disk.
- After partitioning, then the file systems must be logically formatted, which involves laying down the master directory information (FAT table or inode structure), initializing free lists, and creating at least the root directory of the file system. (Disk partitions which are to be used as raw devices are not logically formatted. This saves the overhead and disk space of the file system structure, but requires that the application program manage its own disk storage requirements. )

## Boot Block

- Computer ROM contains a bootstrap program (OS independent) with just enough code to find the first sector on the first hard drive on the first controller, load that sector into memory, and transfer control over to it. ( The ROM bootstrap program may look in floppy and/or CD drives before accessing the hard drive, and is smart enough to recognize whether it has found valid boot code or not. )
- The first sector on the hard drive is known as the Master Boot Record, MBR, and contains a very small amount of code in addition to the partition table. The partition table documents how the disk is partitioned into logical disks, and indicates specifically which partition is the active or boot partition.

- The boot program then looks to the active partition to find an operating system, possibly loading up a slightly larger / more advanced boot program along the way.
- In a dual-boot ( or larger multi-boot ) system, the user may be given a choice of which operating system to boot, with a default action to be taken in the event of no response within some time frame.
- Once the kernel is found by the boot program, it is loaded into memory and then control is transferred over to the OS. The kernel will normally continue the boot process by initializing all important kernel data structures, launching important system services (e.g. network daemons, sched, init, etc.), and finally providing one or more login prompts. Boot options at this stage may include single-user a.k.a. maintenance or safe modes, in which very few system services are started - These modes are designed for system administrators to repair problems or otherwise maintain the system.

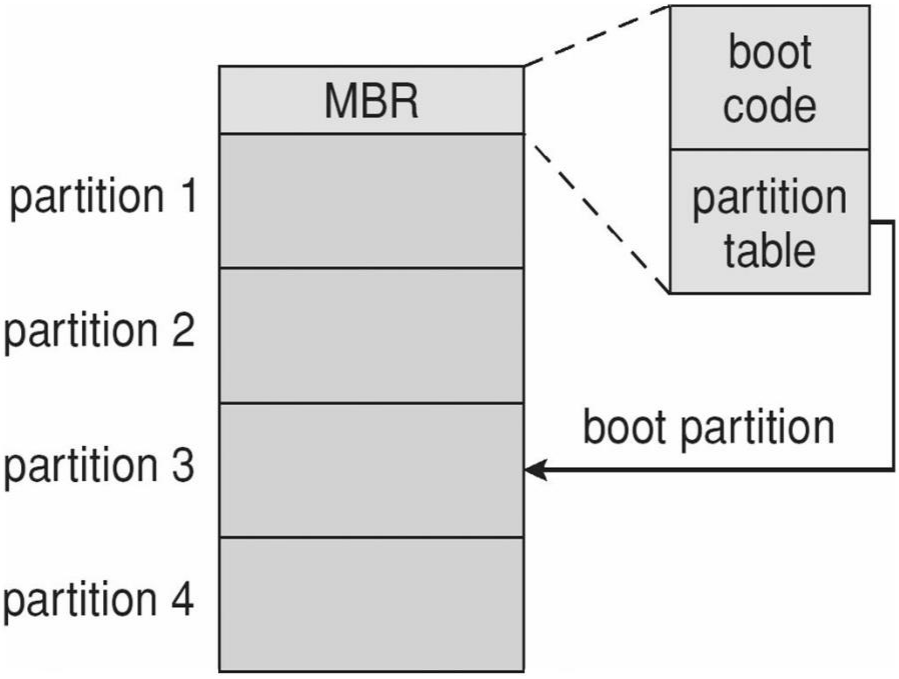



Fig: Booting from disk in Windows 2000.

## Bad Blocks

- No disk can be manufactured to 100% perfection, and all physical objects wear out over time. For these reasons all disks are shipped with a few bad blocks, and additional blocks can be expected to go bad slowly over time. If a large number of blocks go bad then the entire disk will need to be replaced, but a few here and there can be handled through other means.
- In the old days, bad blocks had to be checked for manually. Formatting of the disk or running certain disk-analysis tools would identify bad blocks, and attempt to read the data off of them one last time through repeated tries. Then the bad blocks would be mapped out and taken out of future service. Sometimes the data could be recovered, and sometimes it was lost forever. (Disk analysis tools could be either destructive or non-destructive.)
- Modern disk controllers make much better use of the error-correcting codes, so that bad blocks can be detected earlier and the data usually recovered. (Recall that blocks are tested with every write as well as with every read, so often errors can be detected before the write operation is complete, and the data simply written to a different sector instead.)
- Note that re-mapping of sectors from their normal linear progression can throw off the disk scheduling optimization of the OS, especially if the replacement sector is physically far away from the sector it is replacing. For this reason most disks normally keep a few spare sectors on each cylinder, as well as at least one spare cylinder. Whenever possible a bad sector will be mapped to another sector on the same cylinder, or at least a cylinder as close as possible. Sector slipping may also be performed, in which all sectors between the

bad sector and the replacement sector are moved down by one, so that the linear progression of sector numbers can be maintained.

- If the data on a bad block cannot be recovered, then a hard error has occurred. which requires replacing the file(s) from backups, or rebuilding them from scratch.

## Swap-Space Management

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

### Swap-Space Use

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

### Swap-Space Location

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

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



### Swap-Space Management: An Example

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

free slots and non-zeros refer to how many processes have a mapping to that particular block (>1 for shared pages only.)

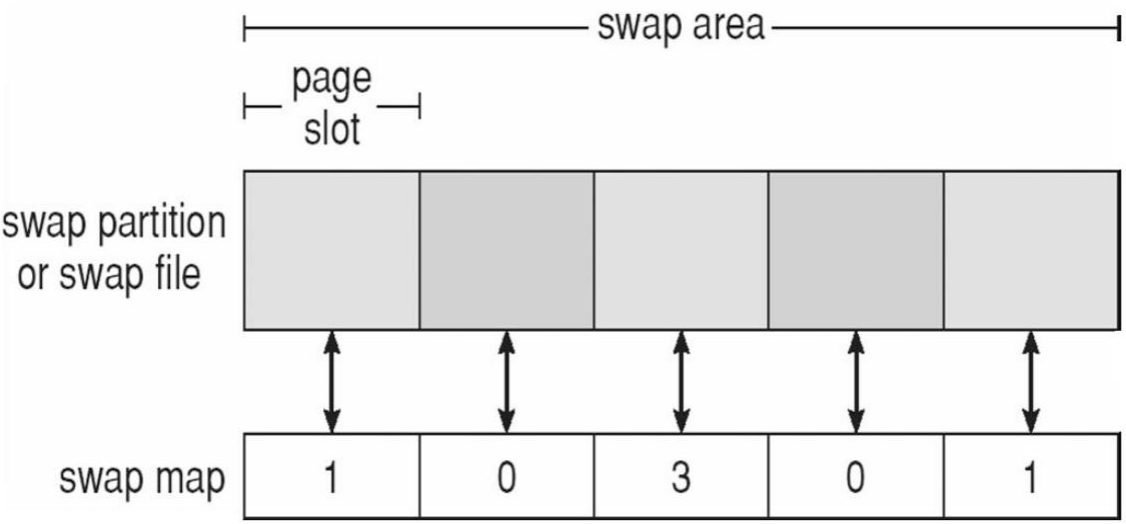



Fig: The data structures for swapping on Linux systems.

# RAID Structure

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

## Improvement of Reliability via Redundancy

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

- Note that a power failure during a write operation could cause both disks to contain corrupt data, if both disks were writing simultaneously at the time of the power failure. One solution is to write to the two disks in series, so that they will not both become corrupted (at least not in the same way) by a power failure. And alternate solution involves non-volatile RAM as a write cache, which is not lost in the event of a power failure and which is protected by error-correcting codes.

## Improvement in Performance via Parallelism

- There is also a performance benefit to mirroring, particularly with respect to reads. Since every block of data is duplicated on multiple disks, read operations can be satisfied from any available copy, and multiple disks can be reading different data blocks simultaneously in parallel. (Writes could possibly be sped up as well through careful scheduling algorithms, but it would be complicated in practice.)
- Another way of improving disk access time is with striping, which basically means spreading data out across multiple disks that can be accessed simultaneously.
  - With **bit-level** striping the bits of each byte are striped across multiple disks. For example if 8 disks were involved, then each 8-bit byte would be read in parallel by 8 heads on separate disks. A single disk read would access 8 * 512 bytes = 4K worth of data in the time normally required to read 512 bytes. Similarly if 4 disks were involved, then two bits of each byte could be stored on each disk, for 2K worth of disk access per read or write operation.
  - **Block-level** striping spreads a file system across multiple disks on a block-by-block basis, so if block N were located on disk 0, then block N + 1 would be on disk 1, and so on. This is particularly useful when file systems are accessed in clusters of physical blocks. Other striping possibilities exist, with block-level striping being the most common.



## RAID Levels

- Mirroring provides reliability but is expensive; Striping improves performance, but does not improve reliability. Accordingly there are a number of different schemes that combine the principals of mirroring and striping in different ways, in order to balance reliability versus performance versus cost. These are described by different RAID levels, as follows: (In the diagram that follows, "C" indicates a copy, and "P" indicates parity, i.e. checksum bits.)
  - **Raid Level 0** - This level includes striping only, with no mirroring.
  - **Raid Level 1** - This level includes mirroring only, no striping.
  - **Raid Level 2** - This level stores error-correcting codes on additional disks, allowing for any damaged data to be reconstructed by subtraction from the remaining undamaged data. Note that this scheme requires only three extra disks to protect 4 disks worth of data, as opposed to full mirroring. (The number of

disks required is a function of the error-correcting algorithms, and the means by which the particular bad bit(s) is (are) identified.)

- **Raid Level 3** - This level is similar to level 2, except that it takes advantage of the fact that each disk is still doing its own error-detection, so that when an error occurs, there is no question about which disk in the array has the bad data. As a result a single parity bit is all that is needed to recover the lost data from an array of disks. Level 3 also includes striping, which improves performance. The downside with the parity approach is that every disk must take part in every disk access, and the parity bits must be constantly calculated and checked, reducing performance. Hardware-level parity calculations and NVRAM cache can help with both of those issues. In practice level 3 is greatly preferred over level 2.
- **Raid Level 4** - This level is similar to level 3, employing block-level striping instead of bit-level striping. The benefits are that multiple blocks can be read independently, and changes to a block only require writing two blocks (data and parity) rather than involving all disks. Note that new disks can be added seamlessly to the system provided they are initialized to all zeros, as this does not affect the parity results.
- **Raid Level 5** - This level is similar to level 4, except the parity blocks are distributed over all disks, thereby more evenly balancing the load on the system. For any given block on the disk(s), one of the disks will hold the parity information for that block and the other N-1 disks will hold the data. Note that the same disk cannot hold both data and parity for the same block, as both would be lost in the event of a disk crash.
- **Raid Level 6** - This level extends raid level 5 by storing multiple bits of error-recovery codes, (such as the Reed-Solomon codes), for each bit position of data, rather than a single parity bit. In the example shown below 2 bits of ECC are stored for every 4 bits of data, allowing data recovery in the face of up to two simultaneous disk failures. Note that this still involves only 50% increase in storage needs, as opposed to 100% for simple mirroring which could only tolerate a single disk failure.

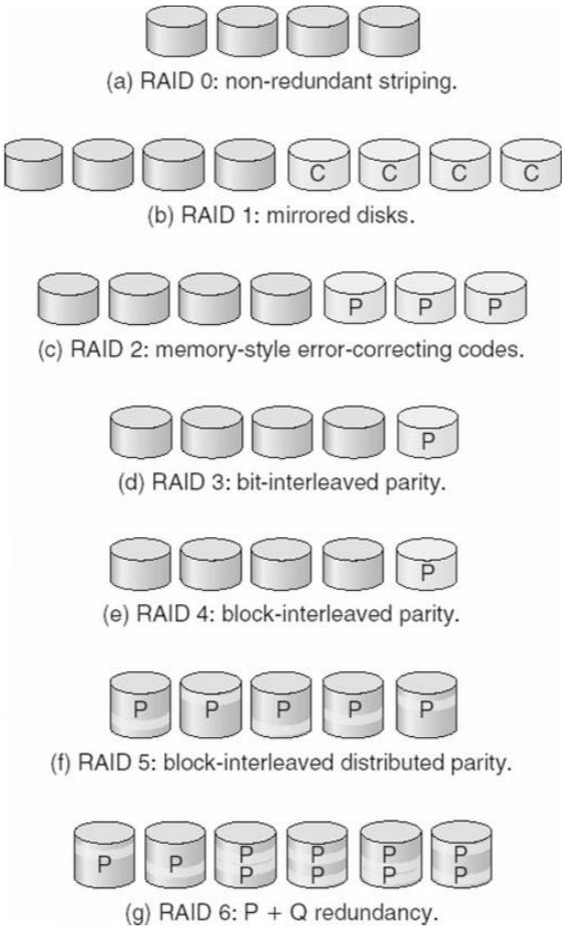



Fig: RAID levels.

- 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.

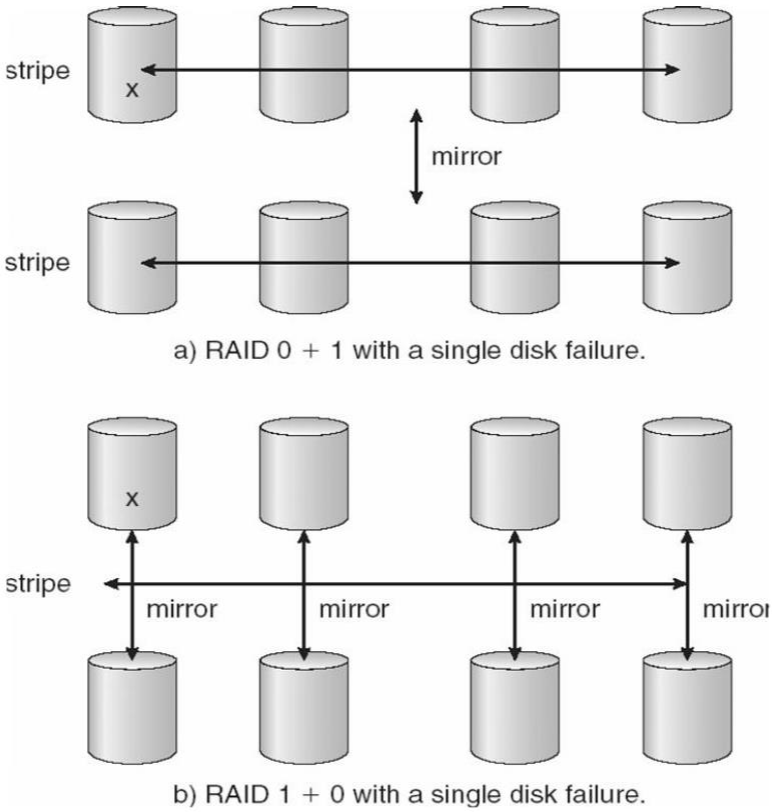



Fig: RAID 0 + 1 and 1 + 0

## Selecting a RAID Level

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

## Extensions

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

## Problems with RAID

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

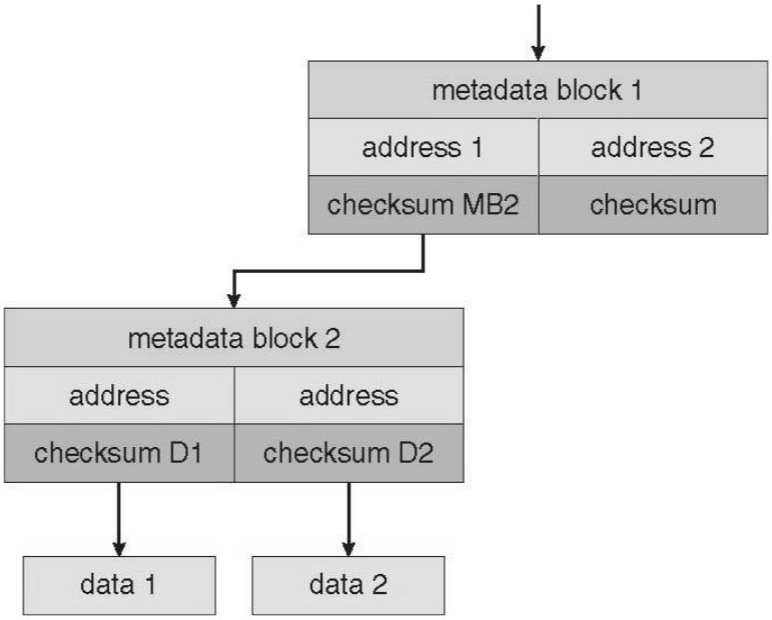



Fig: ZFS checksums all metadata and data.

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

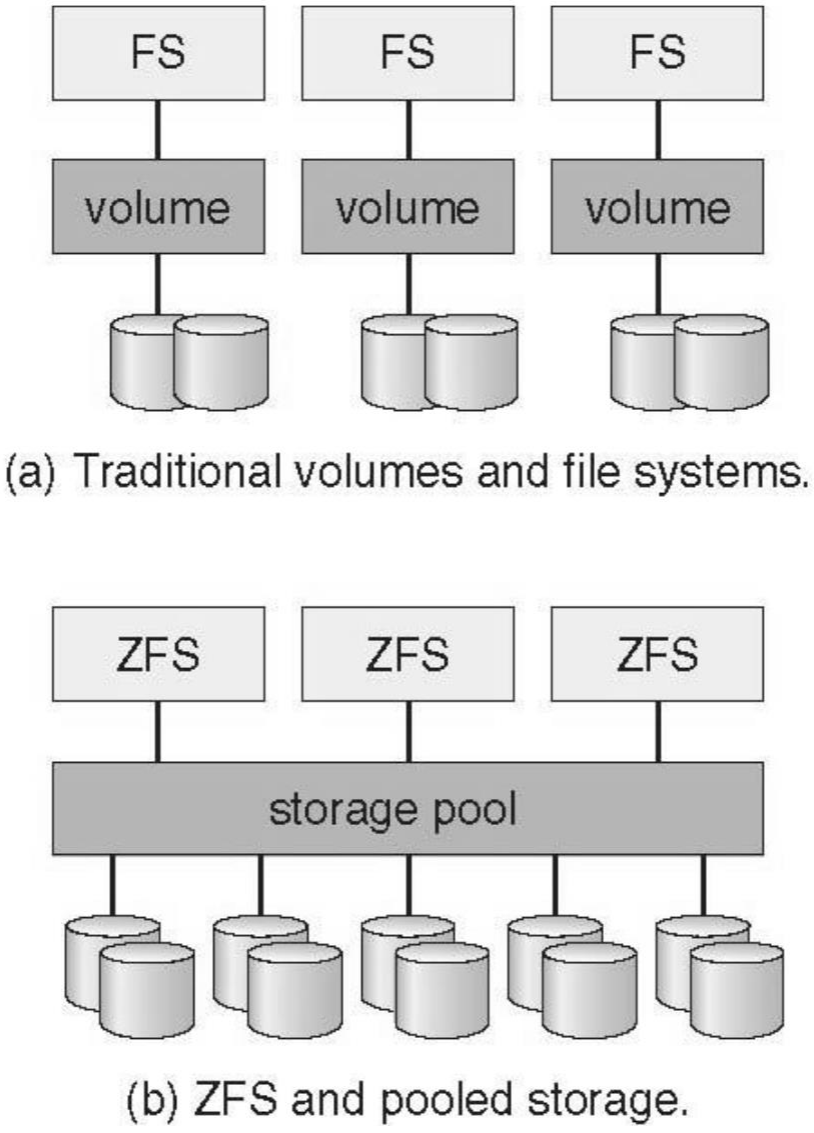



Fig: (a) Traditional volumes and file systems. (b) a ZFS pool and file systems.

## 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:
    - The data is successfully and completely written.
    - The data is partially written, but not completely. The last block written may be garbled.
    - 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:
    - Write the data to the first physical block.
    - After step 1 had completed, then write the data to the second physical block.
    - Declare the operation complete only after both physical writes have completed successfully.
- During recovery the pair of blocks is examined.
    - If both blocks are identical and there is no sign of damage, then no further action is necessary.
    - 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.)
    - 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.

### 4.2 FILE SYSTEM INTERFACE

## File Concept



### File Attributes

- Different OSes keep track of different file attributes, including:
    - Name - Some systems give special significance to names, and particularly extensions ( .exe, .txt, etc. ), and some do not. Some extensions may be of significance to the OS ( .exe ), and others only to certain applications ( .jpg )
    - Identifier ( e.g. inode number )
    - Type - Text, executable, other binary,

    etc.
    - Location - on the hard drive.
    - Size
    - Protection
    - Time & Date
    - User ID

### File Operations

- The file ADT supports many common operations:
    - Creating a file

- Writing a file
- Reading a file
- Repositioning within a file
- Deleting a file
- Truncating a file.

- Most OSes require that files be opened before access and closed after all access is complete. Normally the programmer must open and close files explicitly, but some rare systems open the file automatically at first access. Information about currently open files is stored in an open file table, containing for example:
  - **File pointer** - records the current position in the file, for the next read or write access.
  - **File-open count** - How many times has the current file been opened ( simultaneously by different processes ) and not yet closed? When this counter reaches zero the file can be removed from the table.
  - **Disk location** of the file.
  - **Access rights**
- Some systems provide support for **file locking**.
  - A **shared lock** is for reading only.
  - A **exclusive lock** is for writing as well as reading.
- An **advisory lock** is informational only, and not enforced. ( A "Keep Out" sign, which may be ignored. )
- A **mandatory lock** is enforced. ( A truly locked door. )
- UNIX used advisory locks, and Windows uses mandatory locks.

**File Types**Windows ( and some other systems ) use special file extensions to indicate the type of each file:

| file type | usual extension | function |
|---|---|---|
| executable | exe, com, bin or none | ready-to-run machine-language program |
| object | obj, o | compiled, machine language, not linked |
| source code | c, cc, java, pas, asm, a | source code in various languages |
| batch | bat, sh | commands to the command interpreter |
| text | txt, doc | textual data, documents |
| word processor | wp, tex, rtf, doc | various word-processor formats |
| library | lib, a, so, dll | libraries of routines for programmers |
| print or view | ps, pdf, jpg | ASCII or binary file in a format for printing or viewing |
| archive | arc, zip, tar | related files grouped into one file, sometimes compressed, for archiving or storage |
| multimedia | mpeg, mov, rm, mp3, avi | binary file containing audio or A/V information |

Fig: Common File Types

- Macintosh stores a creator attribute for each file, according to the program that first created it with the create( ) system call.
- UNIX stores magic numbers at the beginning of certain files. ( Experiment with the "file" command, especially in directories such as /bin and /dev )

### File Structure

- Some files contain an internal structure, which may or may not be known to the OS.
- For the OS to support particular file formats increases the size and complexity of the OS.
- UNIX treats all files as sequences of bytes, with no further consideration of the internal structure. ( With the exception of executable binary programs, which it must know how to load and find the first executable statement, etc. )
- Macintosh files have two forks - a resource fork, and a data fork. The resource fork contains information relating to the UI, such as icons and button images, and can be modified independently of the data fork, which contains the code or data as appropriate.

### Internal File Structure

- Disk files are accessed in units of physical blocks, typically 512 bytes or some power-of-two multiple thereof. ( Larger physical disks use larger block sizes, to keep the range of block numbers within the range of a 32-bit integer. )
- Internally files are organized in units of logical units, which may be as small as a single byte, or may be a larger size corresponding to some data record or structure size.
- The number of logical units which fit into one physical block determines its packing, and has an impact on the amount of internal fragmentation ( wasted space ) that occurs.
- As a general rule, half a physical block is wasted for each file, and the larger the block sizes the more space is lost to internal fragmentation.

## Access Methods

### Sequential Access

- A sequential access file emulates magnetic tape operation, and generally supports a few operations:
  - **read next** - read a record and advance the tape to the next position.
  - **write next** - write a record and advance the tape to the next position. o **rewind**
  - **skip n records** - May or may not be supported. N may be limited to positive numbers, or may be limited to +/- 1.


Fig: Sequential File Access

## Direct Access

- Jump to any record and read that record. Operations supported include:
  - read n - read record number n. ( Note an argument is now required. )
  - write n - write record number n. ( Note an argument is now required. )
  - jump to record n - could be 0 or the end of file.
  - Query current record - used to return back to this record later.
  - Sequential access can be easily emulated using direct access. The inverse is complicated and inefficient.

| sequential access | implementation for direct access |
|---|---|
| *reset* | *cp* = 0; |
| *read next* | *read cp*;<br>*cp* = *cp* + 1; |
| *write next* | *write cp*;<br>*cp* = *cp* + 1; |

Fig: Simulation of sequential access on a direct-access file

## Other Access Methods

An indexed access scheme can be easily built on top of a direct access system. Very large files may require a multi-tiered indexing scheme, i.e. indexes of indexes.

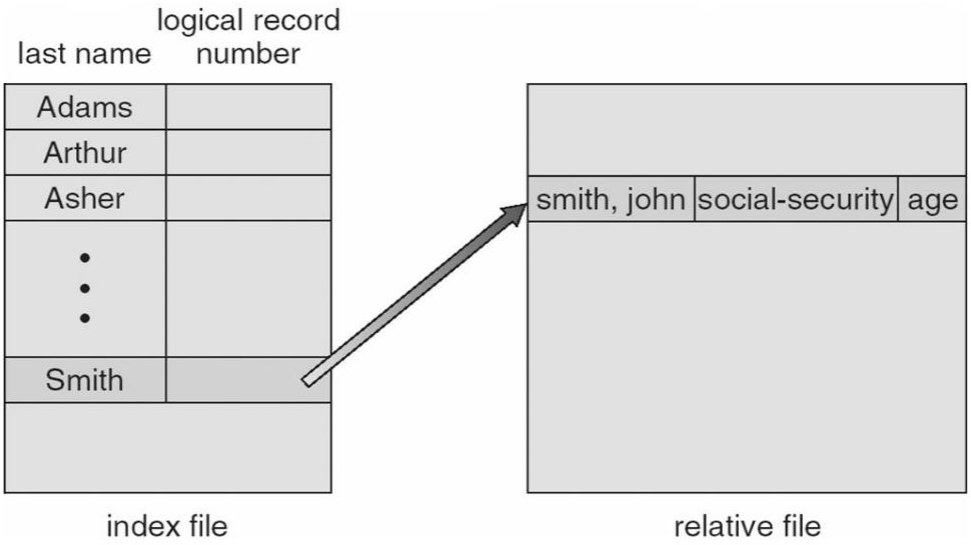



Fig: Example of index and relative files.

# Directory Structure

## Storage Structure

- A disk can be used in its entirety for a file system.
- Alternatively a physical disk can be broken up into multiple partitions, slices, or mini-disks, each of which becomes a virtual disk and can have its own file system. ( or be used for raw storage, swap space, etc. )
- Or, multiple physical disks can be combined into one volume, i.e. a larger virtual disk, with its own file system spanning the physical disks.

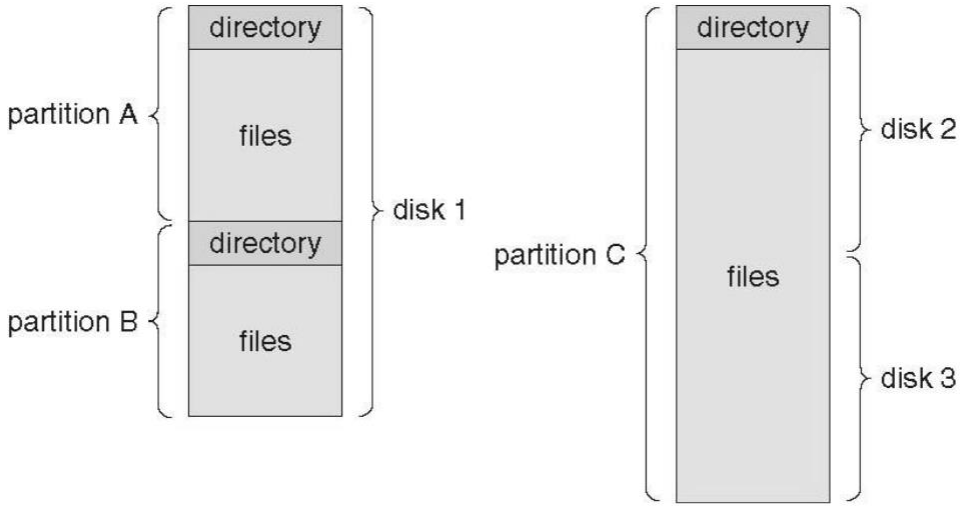


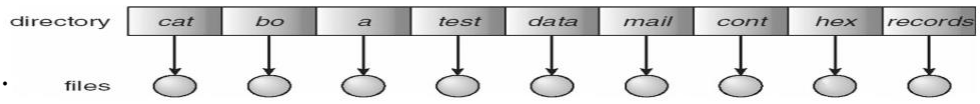

Fig: A typical file-system organization

## Directory Overview

- Directory operations to be supported include:
  - Search for a file
  - Create a file - add to the directory
  - Delete a file - erase from the directory
  - List a directory - possibly ordered in different ways.
  - Rename a file - may change sorting order
  - **Traverse the file**

## system. Single-Level Directory

Simple to implement, but each file must have a unique name.




**Draw Backs:**

- Naming Problem
- Grouping Problem

**Two-Level Directory**

- Each user gets their own directory space.
- File names only need to be unique within a given user's directory.
- A master file directory is used to keep track of each user's directory, and must be maintained when users are added to or removed from the system.
- A separate directory is generally needed for system ( executable ) files.
- Systems may or may not allow users to access other directories besides their own
    - If access to other directories is allowed, then provision must be made to specify the directory being accessed.
    - If access is denied, then special consideration must be made for users to run programs located in system directories. A search path is the list of directories in which to search for executable programs, and can be set uniquely for each user.

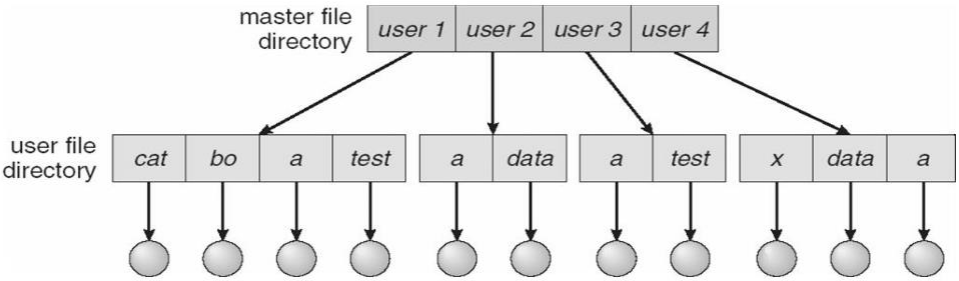



**Fig: Two-level directory structure.**

## Tree-Structured Directories

- An obvious extension to the two-tiered directory structure, and the one with which we are all most familiar.
- Each user / process has the concept of a current directory from which all ( relative ) searches take place.
- Files may be accessed using either absolute pathnames ( relative to the root of the tree ) or relative pathnames ( relative to the current directory. )
- Directories are stored the same as any other file in the system, except there is a bit that identifies them as directories, and they have some special structure that the OS understands.
- One question for consideration is whether or not to allow the removal of directories that are not empty - Windows requires that directories be emptied first, and UNIX provides an option for deleting entire sub-trees.

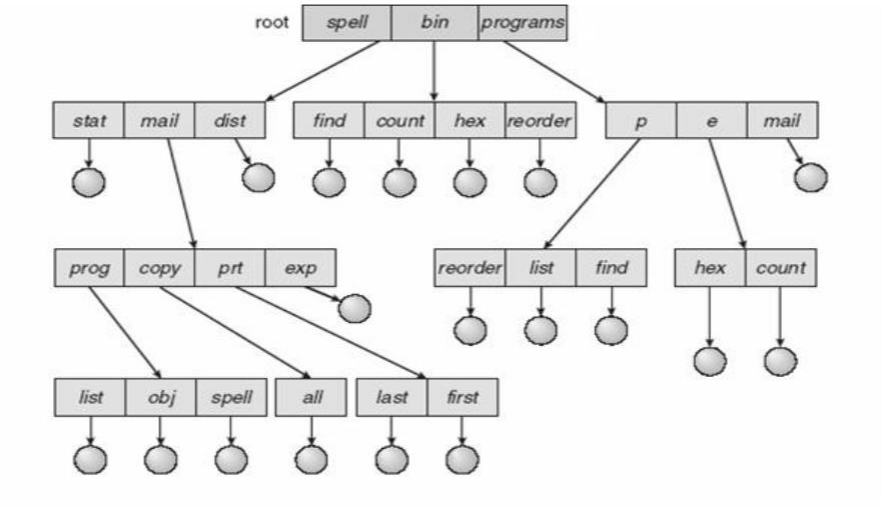



Fig: Tree Structured Directory Structure

## Acyclic-Graph Directories

- When the same files need to be accessed in more than one place in the directory structure ( e.g. because they are being shared by more than one user / process ), it can be useful to provide an acyclic-graph structure. ( Note the directed arcs from parent to child. )
- UNIX provides two types of links for implementing the acyclic-graph structure. ( See "man ln" for more details. )
    - A **hard link** ( usually just called a link ) involves multiple directory entries that both refer to the same file. Hard links are only valid for ordinary files in the same file system.
    - A **symbolic link** that involves a special file, containing information about where to find the linked file. Symbolic links may be used to link directories and/or files in other file systems, as well as ordinary files in the current file system.
- Windows only supports symbolic links, termed shortcuts.
- Hard links require a reference count, or link count for each file, keeping track of how many directory entries are currently referring to this file. Whenever one of the references is removed the link count is reduced, and when it reaches zero, the disk space can be reclaimed.
- For symbolic links there is some question as to what to do with the symbolic links when the original file is moved or deleted:
    - One option is to find all the symbolic links and adjust them also.
    - Another is to leave the symbolic links dangling, and discover that they are no longer valid the next time they are used.
    - What if the original file is removed, and replaced with another file having the same name before the symbolic link is next used?

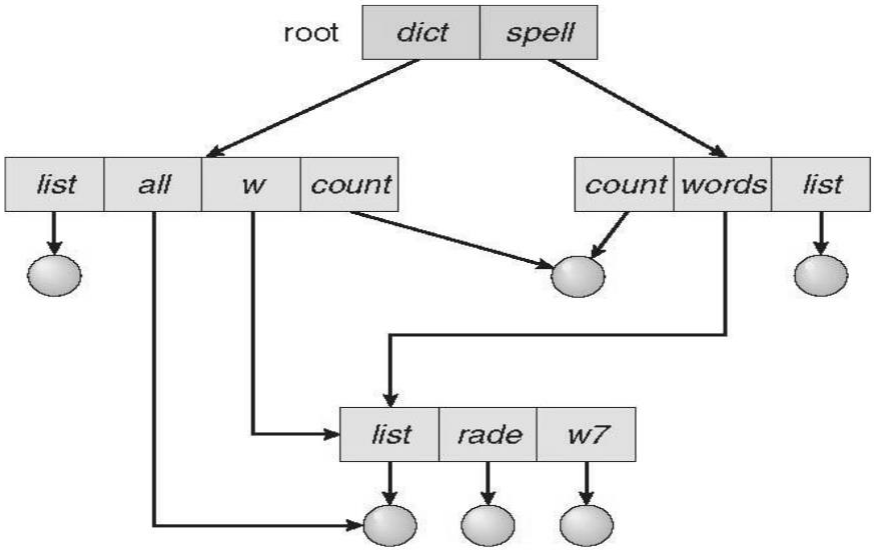



Fig: A cyclic-graph Directory Structure

## General Graph Directory

- If cycles are allowed in the graphs, then several problems can arise:
    - Search algorithms can go into infinite loops. One solution is to not follow links in search algorithms. ( Or not to follow symbolic links, and to only allow symbolic links to refer to directories. )
    - Sub-trees can become disconnected from the rest of the tree and still not have their reference counts reduced to zero. Periodic garbage collection is required to detect and resolve this problem. ( chkdsk in DOS and fsck in UNIX search for these problems, among others, even though cycles are not supposed to be allowed in either system. Disconnected disk blocks that are not marked as free are added back to the file systems with made-up file names, and can usually be safely deleted. )

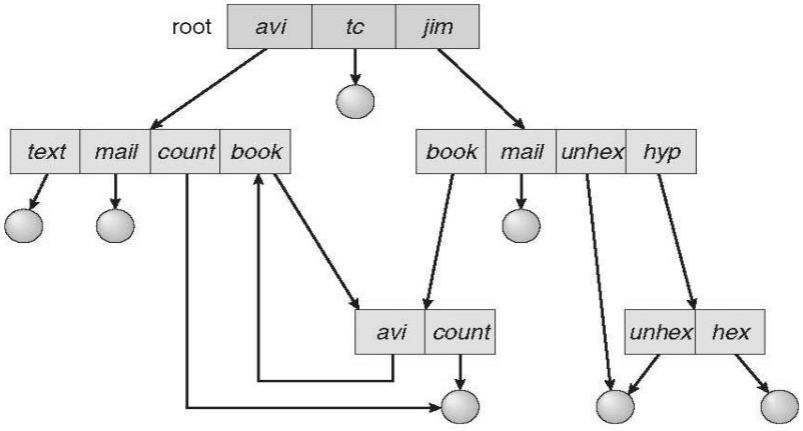



Fig: General Graph Directory

# File-System Mounting

- The basic idea behind mounting file systems is to combine multiple file systems into one large tree structure.
- The mount command is given a file system to mount and a mount point ( directory ) on which to attach it.
- Once a file system is mounted onto a mount point, any further references to that directory actually refer to the root of the mounted file system.
- Any files ( or sub-directories ) that had been stored in the mount point directory prior to mounting the new file system are now hidden by the mounted file system, and are no longer available. For this reason some systems only allow mounting onto empty directories.
- File systems can only be mounted by root, unless root has previously configured certain file systems to be mountable onto certain pre-determined mount points. ( E.g. root may allow users to mount floppy file systems to /mnt or something like it. ) Anyone can run the mount command to see what file systems are currently mounted.
- File systems may be mounted read-only, or have other restrictions imposed.

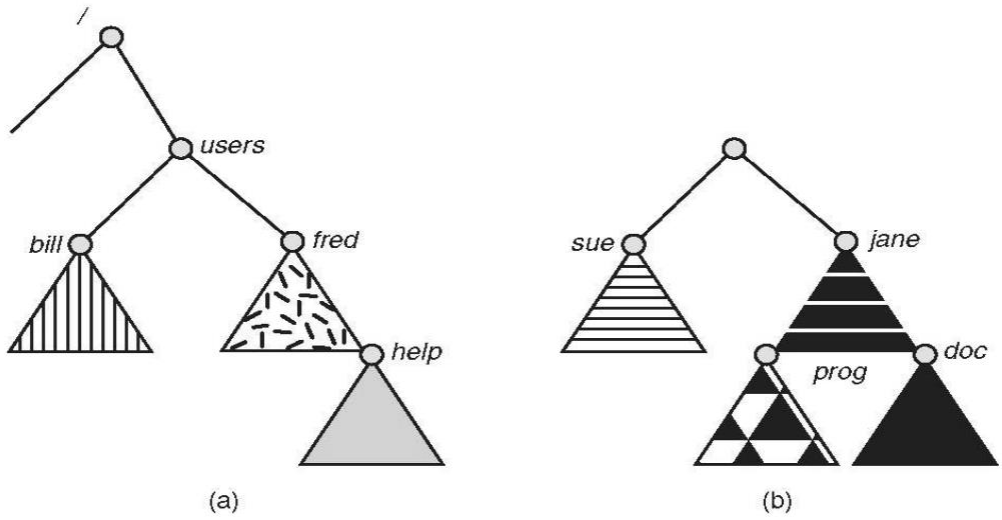



Figure :File system. (a) Existing system. (b) Unmounted volume.

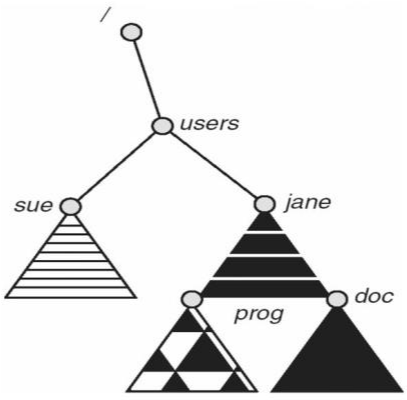


Figure :Mount point.

- The traditional Windows OS runs an extended two-tier directory structure, where the first tier of the structure separates volumes by drive letters, and a tree structure is implemented below that level.
- Macintosh runs a similar system, where each new volume that is found is automatically mounted and added to the desktop when it is found.
- More recent Windows systems allow file systems to be mounted to any directory in the file system, much like UNIX.

# File Sharing

## Multiple Users

- On a multi-user system, more information needs to be stored for each file: ○ The owner ( user ) who owns the file, and who can control its access.
  - The group of other user IDs that may have some special access to the file.
  - What access rights are afforded to the owner ( User ), the Group, and to the rest of the world ( the universe, a.k.a. Others. )
  - Some systems have more complicated access control, allowing or denying specific accesses to specifically named users or groups.

## Remote File Systems

- The advent of the Internet introduces issues for accessing files stored on remote computers
  - The original method was ftp, allowing individual files to be transported across systems as needed. Ftp can be either account and password controlled, or anonymous, not requiring any user name or password.
  - Various forms of distributed file systems allow remote file systems to be mounted onto a local directory structure, and accessed using normal file access commands. ( The actual files are still transported across the network as needed, possibly using ftp as the underlying transport mechanism. )
  - The WWW has made it easy once again to access files on remote systems without mounting their file systems, generally using ( anonymous ) ftp as the underlying file transport mechanism.



### The Client-Server Model

- When one computer system remotely mounts a file system that is physically located on another system, the system which physically owns the files acts as a server, and the system which mounts them is the client.

- User IDs and group IDs must be consistent across both systems for the system to work properly. ( I.e. this is most applicable across multiple computers managed by the same organization, shared by a common group of users. )
- The same computer can be both a client and a server. ( E.g. cross-linked file systems. )
- There are a number of security concerns involved in this model:
    - Servers commonly restrict mount permission to certain trusted systems only. Spoofing ( a computer pretending to be a different computer ) is a potential security risk.
    - Servers may restrict remote access to read-only.
    - Servers restrict which file systems may be remotely mounted. Generally the information within those subsystems is limited, relatively public, and protected by frequent backups.
- The NFS ( Network File System ) is a classic example of such a system.

**Distributed Information Systems**

- The Domain Name System, DNS, provides for a unique naming system across all of the Internet.
- Domain names are maintained by the Network Information System, NIS, which unfortunately has several security issues. NIS+ is a more secure version, but has not yet gained the same widespread acceptance as NIS.
- Microsoft's Common Internet File System, CIFS, establishes a network login for each user on a networked system with shared file access. Older Windows systems used domains, and newer systems ( XP, 2000 ), use active directories. User names must match across the network for this system to be valid.
- A newer approach is the Lightweight Directory-Access Protocol, LDAP, which provides a secure single sign-on for all users to access all resources on a network. This is a secure system which is gaining in popularity, and which has the maintenance advantage of combining authorization information in one central location.



**Failure Modes**

- When a local disk file is unavailable, the result is generally known immediately, and is generally non-recoverable. The only reasonable response is for the response to fail.
- However when a remote file is unavailable, there are many possible reasons, and whether or not it is unrecoverable is not readily apparent. Hence most remote access systems allow for blocking or delayed response, in the hopes that the remote system ( or the network ) will come back up eventually.

## Consistency Semantics

- Consistency Semantics deals with the consistency between the views of shared files on a networked system. When one user changes the file, when do other users see the changes?

- At first glance this appears to have all of the synchronization issues. Unfortunately the long delays involved in network operations prohibit the use of atomic operations.

**UNIX Semantics**

- The UNIX file system uses the following semantics:
- Writes to an open file are immediately visible to any other user who has the file open.
- One implementation uses a shared location pointer, which is adjusted for all sharing users.
- The file is associated with a single exclusive physical resource, which may delay some accesses.

**Session Semantics**

- The Andrew File System, AFS uses the following semantics:
  - Writes to an open file are not immediately visible to other users.
  - When a file is closed, any changes made become available only to users who open the file at a later time.
- According to these semantics, a file can be associated with multiple ( possibly different ) views. Almost no constraints are imposed on scheduling accesses. No user is delayed in reading or writing their personal copy of the file.
- AFS file systems may be accessible by systems around the world. Access control is maintained through ( somewhat ) complicated access control lists, which may grant access to the entire world ( literally ) or to specifically named users accessing the files from specifically named remote environments.



**Immutable-Shared-Files Semantics**

Under this system, when a file is declared as shared by its creator, it becomes immutable and the name cannot be re-used for any other resource. Hence it becomes read-only, and shared access is simple.

# Protection

- Files must be kept safe for reliability ( against accidental damage ), and protection ( against deliberate malicious access. ) The former is usually managed with backup copies. This section discusses the latter.
- One simple protection scheme is to remove all access to a file. However this makes the file unusable, so some sort of controlled access must be arranged.

## Types of Access

The following low-level operations are often controlled:

- Read - View the contents of the file

- Write - Change the contents of the file.
- Execute - Load the file onto the CPU and follow the instructions contained therein.
- Append - Add to the end of an existing file.
- Delete - Remove a file from the system.
- List -View the name and other attributes of files on the system.

Higher-level operations, such as copy, can generally be performed through combinations of the above.

## Access Control

- One approach is to have complicated Access Control Lists, ACL, which specify exactly what access is allowed or denied for specific users or groups.
  - The AFS uses this system for distributed access.
  - Control is very finely adjustable, but may be complicated, particularly when the specific users involved are unknown. ( AFS allows some wild cards, so for example all users on a certain remote system may be trusted, or a given username may be trusted when accessing from any remote system. )
- UNIX uses a set of 9 access control bits, in three groups of three. These correspond to R, W, and X permissions for each of the Owner, Group, and Others. ( See "man chmod" for full details. ) The RWX bits control the following privileges for ordinary files and directories:
- In addition there are some special bits that can also be applied:
  - The set user ID ( SUID ) bit and/or the set group ID ( SGID ) bits applied to executable files temporarily change the identity of whoever runs the program to match that of the owner / group of the executable program. This allows users running specific programs to have access to files ( while running that program ) to which they would normally be unable to access. Setting of these two bits is usually restricted to root, and must be done with caution, as it introduces a potential security leak.
  - The sticky bit on a directory modifies write permission, allowing users to only delete files for which they are the owner. This allows everyone to create files in /tmp, for example, but to only delete files which they have created, and not anyone else's.
  - The SUID, SGID, and sticky bits are indicated with an S, S, and T in the positions for execute permission for the user, group, and others, respectively. If the letter is lower case, ( s, s, t ), then the corresponding execute permission is not also given. If it is upper case, ( S, S, T ), then the corresponding execute permission IS given.
  - The numeric form of chmod is needed to set these advanced bits.

```
-rw-rw-r--    1 pbg    staff      31200   Sep 3 08:30     intro.ps
drwx------    5 pbg    staff        512   Jul 8 09.33     private/
drwxrwxr-x    2 pbg    staff        512   Jul 8 09:35     doc/
drwxrwx---    2 pbg    student      512   Aug 3 14:13     student-proj/
-rw-r--r--    1 pbg    staff       9423   Feb 24 2003     program.c
-rwxr-xr-x    1 pbg    staff      20471   Feb 24 2003     program
drwx--x--x    4 pbg    faculty      512   Jul 31 10:31    lib/
drwx------    3 pbg    staff       1024   Aug 29 06:52    mail/
drwxrwxrwx    3 pbg    staff        512   Jul 8 09:35     test/
```

Fig: Sample permissions in a UNIX system.

- Windows adjusts files access through a simple GUI:

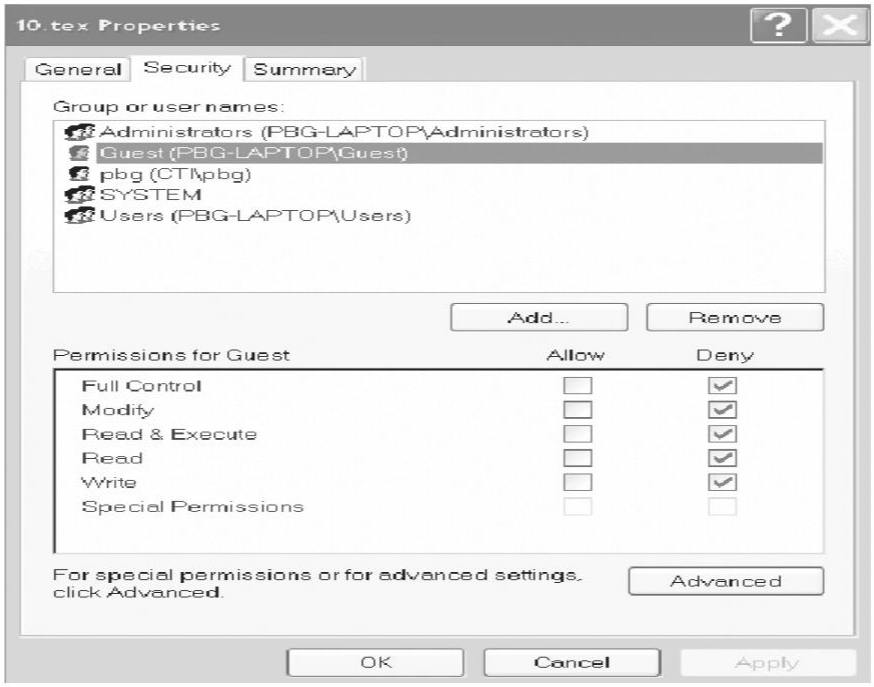



Figure - Windows 7 access-control list management.

## Other Protection Approaches and Issues

- Some systems can apply passwords, either to individual files, or to specific sub-directories, or to the entire system. There is a trade-off between the number of passwords that must be maintained ( and remembered by the users ) and the amount of information that is vulnerable to a lost or forgotten password.
- Older systems which did not originally have multi-user file access permissions ( DOS and older versions of Mac ) must now be retrofitted if they are to share files on a network.

- Access to a file requires access to all the files along its path as well. In a cyclic directory structure, users may have different access to the same file accessed through different paths.
- Sometimes just the knowledge of the existence of a file of a certain name is a security ( or privacy ) concern. Hence the distinction between the R and X bits on UNIX directories.

## 4.3 File-System Implementation

## File-System Structure

- Hard disks have two important properties that make them suitable for secondary storage of files in file systems: (1) Blocks of data can be rewritten in place, and
  (2) they are direct access, allowing any block of data to be accessed with only ( relatively ) minor movements of the disk heads and rotational latency.
- Disks are usually accessed in physical blocks, rather than a byte at a time. Block sizes may range from 512 bytes to 4K or larger.
- File systems organize storage on disk drives, and can be viewed as a layered design:
  - At the lowest layer are the physical devices, consisting of the magnetic media, motors & controls, and the electronics connected to them and controlling them. Modern disk put more and more of the electronic controls directly on the disk drive itself, leaving relatively little work for the disk controller card to perform.
  - I/O Control consists of device drivers, special software programs ( often written in assembly ) which communicate with the devices by reading and writing special codes directly to and from memory addresses corresponding to the controller card's registers. Each controller card ( device ) on a system has a different set of addresses ( registers, a.k.a. ports ) that it listens to, and a unique set of command codes and results codes that it understands.
  - The basic file system level works directly with the device drivers in terms of retrieving and storing raw blocks of data, without any consideration for what is in each block. Depending on the system, blocks may be referred to with a single block number, ( e.g. block # 234234 ), or with head-sector-cylinder combinations.
  - The file organization module knows about files and their logical blocks, and how they map to physical blocks on the disk. In addition to translating from logical to physical blocks, the file organization module also maintains the list of free blocks, and allocates free blocks to files as needed.
  - The logical file system deals with all of the meta data associated with a file ( UID, GID, mode, dates, etc ), i.e. everything about the file except the data itself. This level manages the directory structure and the mapping of file names to file control

blocks, FCBs, which contain all of the meta data as well as block number information for finding the data on the disk.

- The layered approach to file systems means that much of the code can be used uniformly for a wide variety of different file systems, and only certain layers need to be file system specific. Common file systems in use include the UNIX file system, UFS, the Berkeley Fast File System, FFS, Windows systems FAT, FAT32, NTFS, CD-ROM systems ISO 9660, and for Linux the extended file systems ext2 and ext3 ( among 40 others supported)

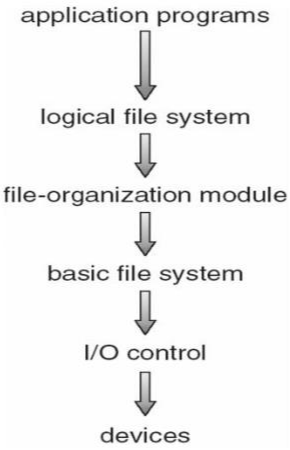



Fig: Layered file system

# File-System Implementation

## Overview

- File systems store several important data structures on the disk:
  - A boot-control block, ( per volume ) a.k.a. the boot block in UNIX or the partition boot sector in Windows contains information about how to boot the system off of this disk. This will generally be the first sector of the volume if there is a bootable system loaded on that volume, or the block will be left vacant otherwise.
  - A volume control block, ( per volume ) a.k.a. the master file table in UNIX or the superblock in Windows, which contains information such as the partition table, number of blocks on each file system, and pointers to free blocks and free FCB blocks.
  - A directory structure ( per file system ), containing file names and pointers to corresponding FCBs. UNIX uses inode numbers, and NTFS uses a master file table.
  - The File Control Block, FCB, ( per file ) containing details about ownership, size, permissions, dates, etc. UNIX stores this information in inodes, and NTFS in the master file table as a relational database structure.

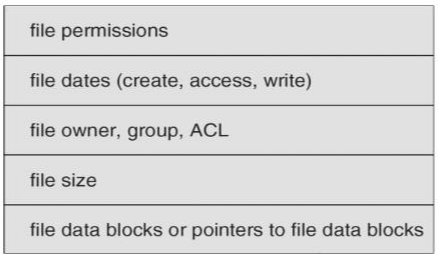



Fig: A typical file-control block.

- There are also several key data structures stored in memory: o An in-memory mount table.
  - o An in-memory directory cache of recently accessed directory information.
  - o A **system-wide open file table**, containing a copy of the FCB for every currently open file in the system, as well as some other related information.
  - o A **per-process open file table**, containing a pointer to the system open file table as well as some other information. ( For example the current file position pointer may be either here or in the system file table, depending on the implementation and whether the file is being shared or not. )
- Figure below illustrates some of the interactions of file system components when files are created and/or used:
  - o When a new file is created, a new FCB is allocated and filled out with important information regarding the new file. The appropriate directory is modified with the new file name and FCB information.
  - o When a file is accessed during a program, the open( ) system call reads in the FCB information from disk, and stores it in the system-wide open file table. An entry is added to the per-process open file table referencing the system-wide table, and an index into the per-process table is returned by the open( ) system call. UNIX refers to this index as a file descriptor, and Windows refers to it as a file handle.
  - o If another process already has a file open when a new request comes in for the same file, and it is sharable, then a counter in the system-wide table is incremented and the per-process table is adjusted to point to the existing entry in the system-wide table.
  - o When a file is closed, the per-process table entry is freed, and the counter in the system-wide table is decremented. If that counter reaches zero, then the system wide table is also freed. Any data currently stored in memory cache for this file is written out to disk if necessary.

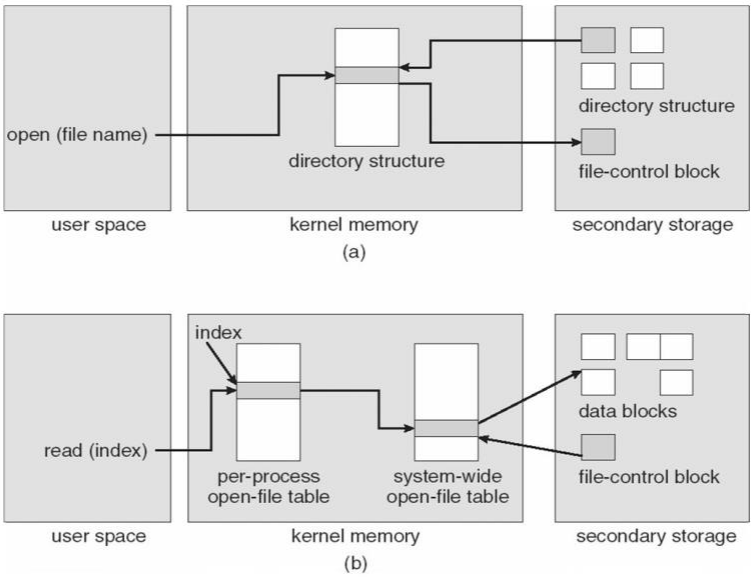



Fig: In-memory file-system structures. (a) File open. (b) File read.

## Partitions and Mounting

- Physical disks are commonly divided into smaller units called partitions. They can also be combined into larger units, but that is most commonly done for RAID installations and is left for later chapters.
- Partitions can either be used as raw devices ( with no structure imposed upon them ), or they can be formatted to hold a file system ( i.e. populated with FCBs and initial directory structures as appropriate. ) Raw partitions are generally used for swap space, and may also be used for certain programs such as databases that choose to manage their own disk storage system. Partitions containing file systems can generally only be accessed using the file system structure by ordinary users, but can often be accessed as a raw device also by root.
- The boot block is accessed as part of a raw partition, by the boot program prior to any operating system being loaded. Modern boot programs understand multiple OSes and file system formats, and can give the user a choice of which of several available systems to boot.
- The **root partition** contains the OS kernel and at least the key portions of the OS needed to complete the boot process. At boot time the root partition is mounted, and control is transferred from the boot program to the kernel found there. ( Older systems required that the root partition lie completely within the first 1024 cylinders of the disk, because that was as far as the boot program could reach. Once the kernel had control, then it could access partitions beyond the 1024 cylinder boundary. )
- Continuing with the boot process, additional file systems get mounted, adding their information into the appropriate mount table structure. As a part of the mounting process the file systems may be checked for errors or inconsistencies, either because they are flagged as not having been closed properly the last time they were used, or just for general principals. File systems may be mounted either automatically or manually. In

UNIX a mount point is indicated by setting a flag in the in-memory copy of the inode, so all future references to that inode get re-directed to the root directory of the mounted file system.

### Virtual File Systems

- Virtual File Systems, VFS, provide a common interface to multiple different file system types. In addition, it provides for a unique identifier ( vnode ) for files across the entire space, including across all file systems of different types. ( UNIX inodes are unique only across a single file system, and certainly do not carry across networked file systems. )
- The VFS in Linux is based upon four key object types:
    - The inode object, representing an individual file
    - The file object, representing an open file.
    - The superblock object, representing a file system.
    - The dentry object, representing a directory entry.
- Linux VFS provides a set of common functionalities for each file system, using function pointers accessed through a table. The same functionality is accessed through the same table position for all file system types, though the actual functions pointed to by the pointers may be file system-specific. See /usr/include/linux/fs.h for full details. Common operations provided include open( ), read( ), write( ), and mmap( ).

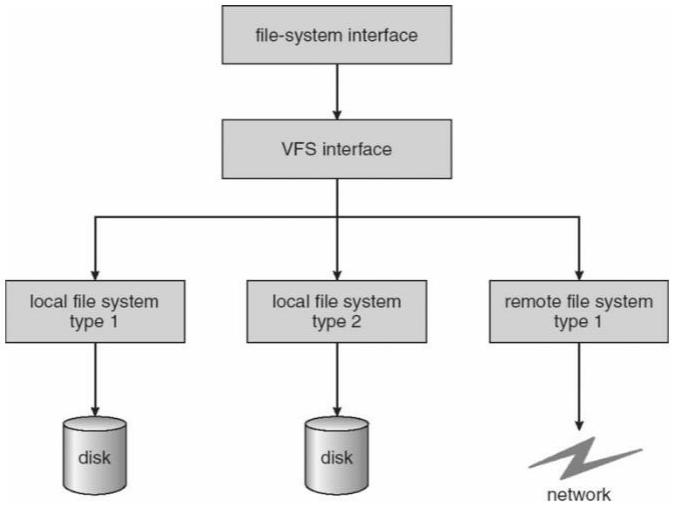



Fig: Schematic view of a virtual file system

## Directory Implementation

Directories need to be fast to search, insert, and delete, with a minimum of wasted disk space.

### Linear List

- A linear list is the simplest and easiest directory structure to set up, but it does have some drawbacks.
- Finding a file ( or verifying one does not already exist upon creation ) requires a linear search.
- Deletions can be done by moving all entries, flagging an entry as deleted, or by moving the last entry into the newly vacant position.
- Sorting the list makes searches faster, at the expense of more complex insertions and deletions.
- A linked list makes insertions and deletions into a sorted list easier, with overhead for the links.
- More complex data structures, such as B-trees, could also be considered.

### Hash Table

- A hash table can also be used to speed up searches.
- Hash tables are generally implemented in addition to a linear or other structure

## Allocation Methods

There are three major methods of storing files on disks:

- contiguous,
- linked, and
- indexed.



### Contiguous Allocation

- Contiguous Allocation requires that all blocks of a file be kept together contiguously.
- Performance is very fast, because reading successive blocks of the same file generally requires no movement of the disk heads, or at most one small step to the next adjacent cylinder.
- Storage allocation involves the same issues discussed earlier for the allocation of contiguous blocks of memory ( first fit, best fit, fragmentation problems, etc. ) The distinction is that the high time penalty required for moving the disk heads from spot to spot may now justify the benefits of keeping files contiguously when possible.
- ( Even file systems that do not by default store files contiguously can benefit from certain utilities that compact the disk and make all files contiguous in the process. )
- Problems can arise when files grow, or if the exact size of a file is unknown at creation time:
    - Over-estimation of the file's final size increases external fragmentation and wastes disk space.
    - Under-estimation may require that a file be moved or a process aborted if the file grows beyond its originally allocated space.

- If a file grows slowly over a long time period and the total final space must be allocated initially, then a lot of space becomes unusable before the file fills the space.

- A variation is to allocate file space in large contiguous chunks, called extents. When a file outgrows its original extent, then an additional one is allocated. ( For example an extent may be the size of a complete track or even cylinder, aligned on an appropriate track or cylinder boundary. ) The high-performance files system Veritas uses extents to optimize performance.

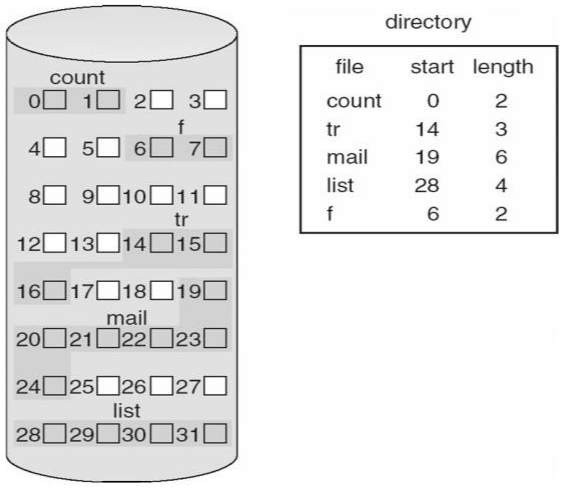



Fig: Contiguous allocation of disk space.

## Linked Allocation

- Disk files can be stored as linked lists, with the expense of the storage space consumed by each link. ( E.g. a block may be 508 bytes instead of 512. )
- Linked allocation involves no external fragmentation, does not require pre-known file sizes, and allows files to grow dynamically at any time.
- Unfortunately linked allocation is only efficient for sequential access files, as random access requires starting at the beginning of the list for each new location access.
- Allocating clusters of blocks reduces the space wasted by pointers, at the cost of internal fragmentation.
- Another big problem with linked allocation is reliability if a pointer is lost or damaged. Doubly linked lists provide some protection, at the cost of additional overhead and wasted space.

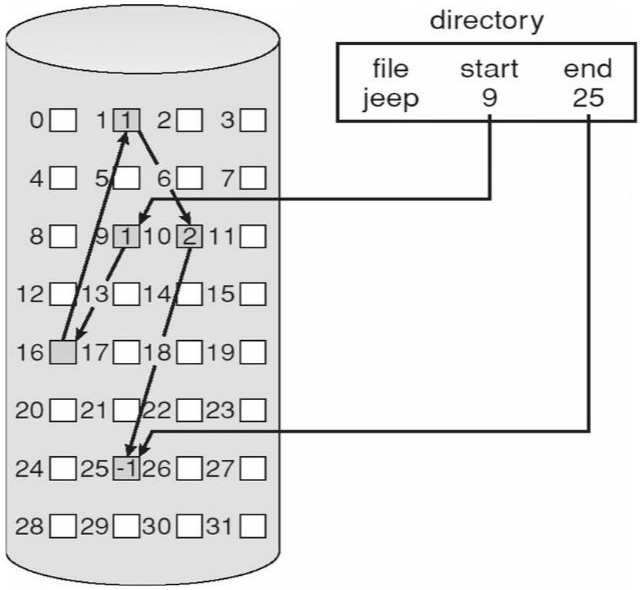



Fig: Linked allocation of disk space.

- The File Allocation Table, FAT, used by DOS is a variation of linked allocation, where all the links are stored in a separate table at the beginning of the disk. The benefit of this approach is that the FAT table can be cached in memory, greatly improving random access speeds.

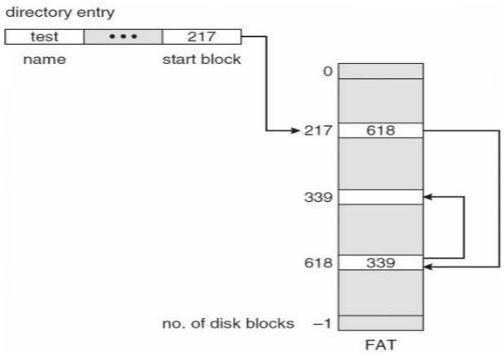



Fig: File-allocation table

## Indexed Allocation

- Indexed Allocation combines all of the indexes for accessing each file into a common block ( for that file ), as opposed to spreading them all over the disk or storing them in a FAT table.

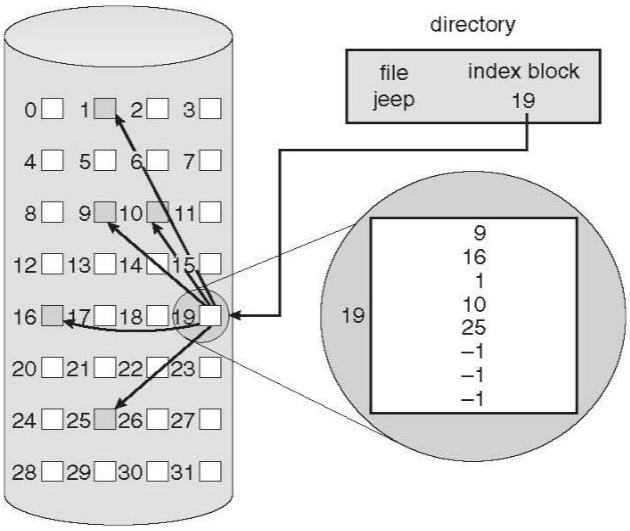



Fig: Indexed allocation of disk space

- Some disk space is wasted ( relative to linked lists or FAT tables ) because an entire index block must be allocated for each file, regardless of how many data blocks the file contains. This leads to questions of how big the index block should be, and how it should be implemented. There are several approaches:
    - **Linked Scheme** - An index block is one disk block, which can be read and written in a single disk operation. The first index block contains some header information, the first N block addresses, and if necessary a pointer to additional linked index blocks.
    - **Multi-Level Index** - The first index block contains a set of pointers to secondary index blocks, which in turn contain pointers to the actual data blocks.
    - **Combined Scheme** - This is the scheme used in UNIX inodes, in which the first 12 or so data block pointers are stored directly in the inode, and then singly, doubly, and triply indirect pointers provide access to more data blocks as needed. ( See below. ) The advantage of this scheme is that for small files ( which many are ), the data blocks are readily accessible ( up to 48K with 4K block sizes ); files up to about 4144K ( using 4K blocks ) are accessible with only a single indirect block ( which can be cached ), and huge files are still accessible using a relatively small number of disk accesses ( larger in theory than can be addressed by a 32-bit address, which is why some systems have moved to 64-bit file pointers. )

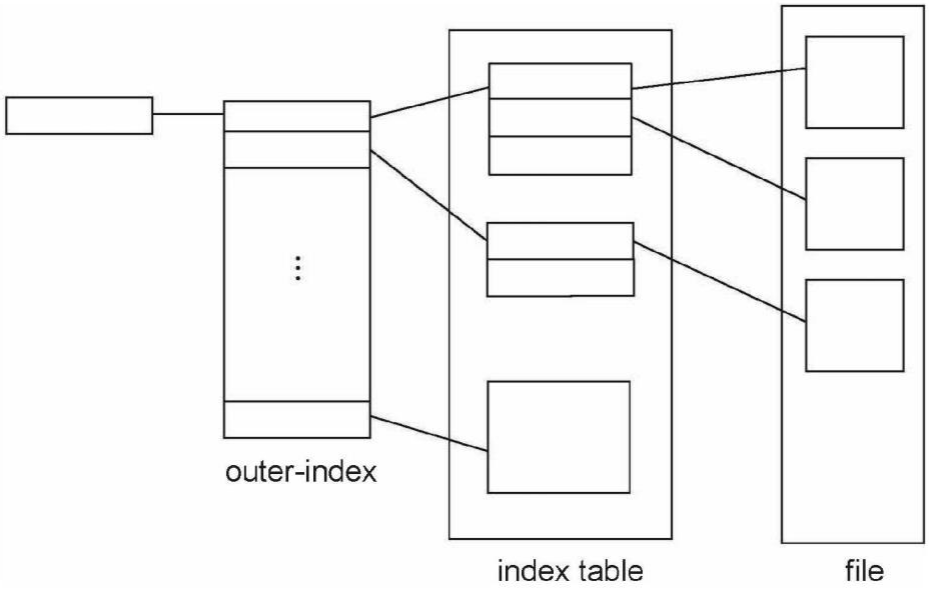



Fig: The UNIX inode

### Performance

- The optimal allocation method is different for sequential access files than for random access files, and is also different for small files than for large files.
- Some systems support more than one allocation method, which may require specifying how the file is to be used ( sequential or random access ) at the time it is allocated. Such systems also provide conversion utilities.
- Some systems have been known to use contiguous access for small files, and automatically switch to an indexed scheme when file sizes surpass a certain threshold.
- And of course some systems adjust their allocation schemes ( e.g. block sizes ) to best match the characteristics of the hardware for optimum performance.

## Free-Space Management

Another important aspect of disk management is keeping track of and allocating free space.

### Bit Vector

- One simple approach is to use a bit vector, in which each bit represents a disk block, set to 1 if free or 0 if allocated.
- Fast algorithms exist for quickly finding contiguous blocks of a given size
- The down side is that a 40GB disk requires over 5MB just to store the bitmap. ( For example. )

### Linked List

- A linked list can also be used to keep track of all free blocks.

- Traversing the list and/or finding a contiguous block of a given size are not easy, but fortunately are not frequently needed operations. Generally the system just adds and removes single blocks from the beginning of the list.
- The FAT table keeps track of the free list as just one more linked list on the table.

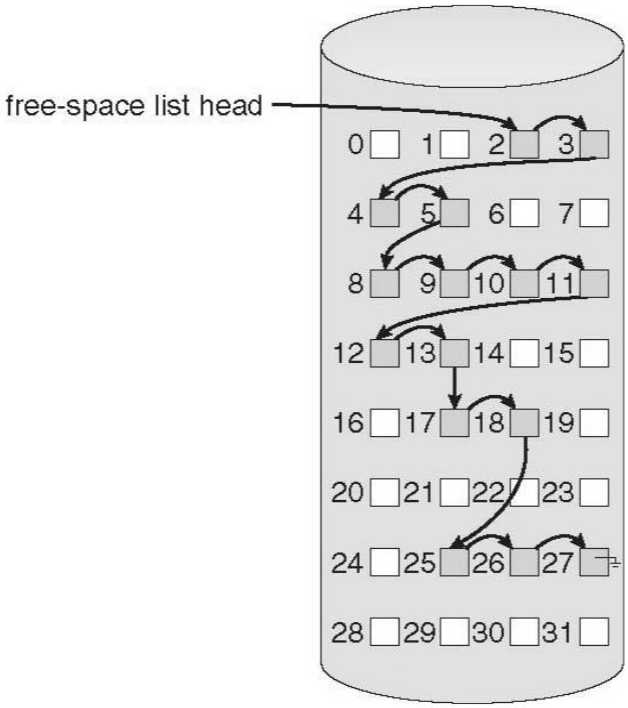



Fig: Linked free-space list on disk.

## Grouping

A variation on linked list free lists is to use links of blocks of indices of free blocks. If a block holds up to N addresses, then the first block in the linked-list contains up to N-1 addresses of free blocks and a pointer to the next block of free addresses.

## Counting

When there are multiple contiguous blocks of free space then the system can keep track of the starting address of the group and the number of contiguous free blocks. As long as the average length of a contiguous group of free blocks is greater than two this offers a savings in space needed for the free list. ( Similar to compression techniques used for graphics images when a group of pixels all the same color is encountered. )

## Space Maps

- Sun's ZFS file system was designed for HUGE numbers and sizes of files, directories, and even file systems.

- The resulting data structures could be VERY inefficient if not implemented carefully. For example, freeing up a 1 GB file on a 1 TB file system could involve updating thousands of blocks of free list bit maps if the file was spread across the disk.
- ZFS uses a combination of techniques, starting with dividing the disk up into ( hundreds of ) metaslabs of a manageable size, each having their own space map.
- Free blocks are managed using the counting technique, but rather than write the information to a table, it is recorded in a log-structured transaction record. Adjacent free blocks are also coalesced into a larger single free block.
- An in-memory space map is constructed using a balanced tree data structure, constructed from the log data.
- The combination of the in-memory tree and the on-disk log provide for very fast and efficient management of these very large files and free blocks.

# Efficiency and Performance

## Efficiency

- UNIX pre-allocates inodes, which occupies space even before any files are created.
- UNIX also distributes inodes across the disk, and tries to store data files near their inode, to reduce the distance of disk seeks between the inodes and the data.
- Some systems use variable size clusters depending on the file size.
- The more data that is stored in a directory ( e.g. last access time ), the more often the directory blocks have to be re-written.
- As technology advances, addressing schemes have had to grow as well.
    - Sun's ZFS file system uses 128-bit pointers, which should theoretically never need to be expanded. ( The mass required to store 2^128 bytes with atomic storage would be at least 272 trillion kilograms! )
- Kernel table sizes used to be fixed, and could only be changed by rebuilding the kernels. Modern tables are dynamically allocated, but that requires more complicated algorithms for accessing them.



## Performance

- Disk controllers generally include on-board caching. When a seek is requested, the heads are moved into place, and then an entire track is read, starting from whatever sector is currently under the heads ( reducing latency. ) The requested sector is returned and the unrequested portion of the track is cached in the disk's electronics.
- Some OSes cache disk blocks they expect to need again in a buffer cache.
- A page cache connected to the virtual memory system is actually more efficient as memory addresses do not need to be converted to disk block addresses and back again.
- Some systems ( Solaris, Linux, Windows 2000, NT, XP ) use page caching for both process pages and file data in a unified virtual memory.

- Figures below show the advantages of the unified buffer cache found in some versions of UNIX and Linux - Data does not need to be stored twice, and problems of inconsistent buffer information are avoided.

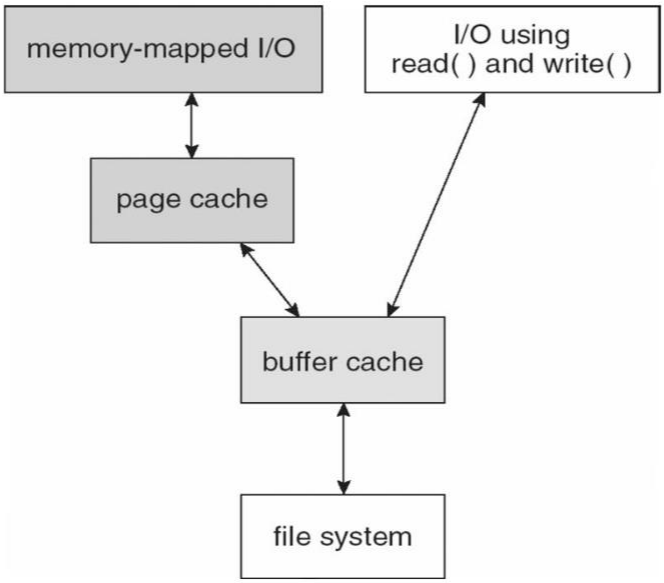



Figure : I/O without a unified buffer cache.

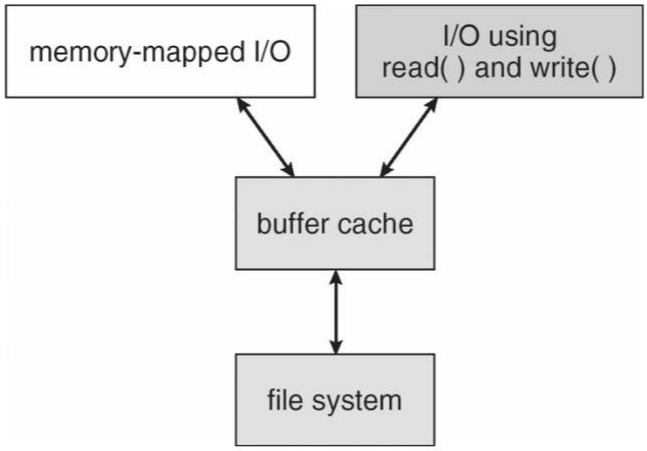



Figure 12.12 - I/O using a unified buffer cache.

- Page replacement strategies can be complicated with a unified cache, as one needs to decide whether to replace process or file pages, and how many pages to guarantee to each category of pages. Solaris, for example, has gone through many variations, resulting in priority paging giving process pages priority over file I/O pages, and setting limits so that neither can knock the other completely out of memory.
- Another issue affecting performance is the question of whether to implement synchronous writes or asynchronous writes. Synchronous writes occur in the order in which the disk subsystem receives them, without caching; Asynchronous writes are

cached, allowing the disk subsystem to schedule writes in a more efficient order ( See Chapter 12. ) Metadata writes are often done synchronously. Some systems support flags to the open call requiring that writes be synchronous, for example for the benefit of database systems that require their writes be performed in a required order.

- The type of file access can also have an impact on optimal page replacement policies. For example, LRU is not necessarily a good policy for sequential access files. For these types of files progression normally goes in a forward direction only, and the most recently used page will not be needed again until after the file has been rewound and re-read from the beginning, ( if it is ever needed at all. ) On the other hand, we can expect to need the next page in the file fairly soon. For this reason sequential access files often take advantage of two special policies:
    - **Free-behind** frees up a page as soon as the next page in the file is requested, with the assumption that we are now done with the old page and won't need it again for a long time.
    - **Read-ahead** reads the requested page and several subsequent pages at the same time, with the assumption that those pages will be needed in the near future. This is similar to the track caching that is already performed by the disk controller, except it saves the future latency of transferring data from the disk controller memory into motherboard main memory.
- The caching system and asynchronous writes speed up disk writes considerably, because the disk subsystem can schedule physical writes to the disk to minimize head movement and disk seek times. Reads, on the other hand, must be done more synchronously in spite of the caching system, with the result that disk writes can counter-intuitively be much faster on average than disk reads.



## Recovery

### Consistency Checking

- The storing of certain data structures ( e.g. directories and inodes ) in memory and the caching of disk operations can speed up performance, but what happens in the result of a system crash? All volatile memory structures are lost, and the information stored on the hard drive may be left in an inconsistent state.
- A Consistency Checker ( fsck in UNIX, chkdsk or scandisk in Windows ) is often run at boot time or mount time, particularly if a file system was not closed down properly. Some of the problems that these tools look for include:
    - Disk blocks allocated to files and also listed on the free list.
    - Disk blocks neither allocated to files nor on the free list. ○

    Disk blocks allocated to more than one file.
    - The number of disk blocks allocated to a file inconsistent with the file's stated size.
    - Properly allocated files / inodes which do not appear in any directory entry.

    - Link counts for an inode not matching the number of references to that inode in the directory structure.
    - Two or more identical file names in the same directory.
    - Illegally linked directories, e.g. cyclical relationships where those are not allowed, or files/directories that are not accessible from the root of the directory tree.
    - Consistency checkers will often collect questionable disk blocks into new files with names such as chk00001.dat. These files may contain valuable information that would otherwise be lost, but in most cases they can be safely deleted, ( returning those disk blocks to the free list. )
- UNIX caches directory information for reads, but any changes that affect space allocation or metadata changes are written synchronously, before any of the corresponding data blocks are written to.

## Log-Structured File Systems

- Log-based transaction-oriented ( a.k.a. journaling ) file systems borrow techniques developed for databases, guaranteeing that any given transaction either completes successfully or can be rolled back to a safe state before the transaction commenced:
    - All metadata changes are written sequentially to a log.
    - A set of changes for performing a specific task ( e.g. moving a file ) is a transaction.
    - As changes are written to the log they are said to be committed, allowing the system to return to its work.
    - In the meantime, the changes from the log are carried out on the actual file system, and a pointer keeps track of which changes in the log have been completed and which have not yet been completed.
    - When all changes corresponding to a particular transaction have been completed, that transaction can be safely removed from the log.
    - At any given time, the log will contain information pertaining to uncompleted transactions only, e.g. actions that were committed but for which the entire transaction has not yet been completed.
        - From the log, the remaining transactions can be completed,
        - or if the transaction was aborted, then the partially completed changes can be undone.



## Other Solutions

- Sun's ZFS and Network Appliance's WAFL file systems take a different approach to file system consistency.
- No blocks of data are ever over-written in place. Rather the new data is written into fresh new blocks, and after the transaction is complete, the metadata ( data block pointers ) is updated to point to the new blocks.
    - The old blocks can then be freed up for future use.

    - Alternatively, if the old blocks and old metadata are saved, then a snapshot of the system in its original state is preserved. This approach is taken by WAFL.
- ZFS combines this with check-summing of all metadata and data blocks, and RAID, to ensure that no inconsistencies are possible, and therefore ZFS does not incorporate a consistency checker.

## Backup and Restore

- In order to recover lost data in the event of a disk crash, it is important to conduct backups regularly.
- Files should be copied to some removable medium, such as magnetic tapes, CDs, DVDs, or external removable hard drives.
- A full backup copies every file on a file system.
- Incremental backups copy only files which have changed since some previous time.
- A combination of full and incremental backups can offer a compromise between full recoverability, the number and size of backup tapes needed, and the number of tapes that need to be used to do a full restore. For example, one strategy might be:
    - At the beginning of the month do a full backup.
    - At the end of the first and again at the end of the second week, backup all files which have changed since the beginning of the month.
    - At the end of the third week, backup all files that have changed since the end of the second week.
    - Every day of the month not listed above, do an incremental backup of all files that have changed since the most recent of the weekly backups described above.
- Backup tapes are often reused, particularly for daily backups, but there are limits to how many times the same tape can be used.
- Every so often a full backup should be made that is kept "forever" and not overwritten.
- Backup tapes should be tested, to ensure that they are readable!
- For optimal security, backup tapes should be kept off-premises, so that a fire or burglary cannot destroy both the system and the backups. There are companies ( e.g. Iron Mountain ) that specialize in the secure off-site storage of critical backup information.
- Keep your backup tapes secure - The easiest way for a thief to steal all your data is to simply pocket your backup tapes!
- Storing important files on more than one computer can be an alternate though less reliable form of backup.
- Note that incremental backups can also help users to get back a previous version of a file that they have since changed in some way.
- Beware that backups can help forensic investigators recover e-mails and other files that users had though they had deleted!