

UNIT V

FILE AND SECONDARY STORAGE MANAGEMENT

I/O Hardware: I/O devices, Device controllers, Direct memory access, Principles of I/O Software **Disk Management:** Disk structure, **Disk scheduling** - FCFS, SSTF, SCAN, C-SCAN, **Disk reliability**, Disk formatting, Boot-block, Bad blocks

File Management: Concept of File, Access methods, File types, File operation, Directory structure, File System structure, **Allocation methods** (contiguous, linked, indexed), **Free-space management** (bit vector, linked list, grouping), **directory implementation** (linear list, hash table), efficiency and performance.

Overview

- Management of **I/O devices** is a very important part of the operating system - entire I/O subsystems are devoted to its operation. (Consider the range of devices on a modern computer, from mouse, keyboards, disk drives, display adapters, USB devices, network connections, audio I/O, printers, special devices for the handicapped, and many special-purpose peripherals.)
- **Device drivers** are **modules** that can be **plugged into an OS to handle a particular device**.

I/O Hardware:

- **I/O devices** can be roughly categorized as **storage, communications, user-interface**, and other
- Devices **communicate** with the computer via signals sent **over wires or through the air**.
- Devices connect with the computer via **ports**, e.g. a **serial or parallel port**.
- A common set of wires connecting multiple devices is termed a **bus**.
 - Buses include rigid protocols for the types of messages that can be sent across the bus and the procedures for resolving contention issues.
 - Figure below illustrates three of the **four bus types** commonly found in a modern PC:
 1. The **PCI bus** connects **high-speed high-bandwidth devices to the memory subsystem** (and the CPU)
 2. The **expansion bus** connects **slower low-bandwidth devices**, which typically deliver data one character at a time (with buffering)
 3. The **SCSI bus** connects a number of **SCSI devices** to a common SCSI controller.
 4. A **daisy-chain bus**, is when a string of devices is connected to each other like beads on a chain, and only one of the devices is directly connected to the host.

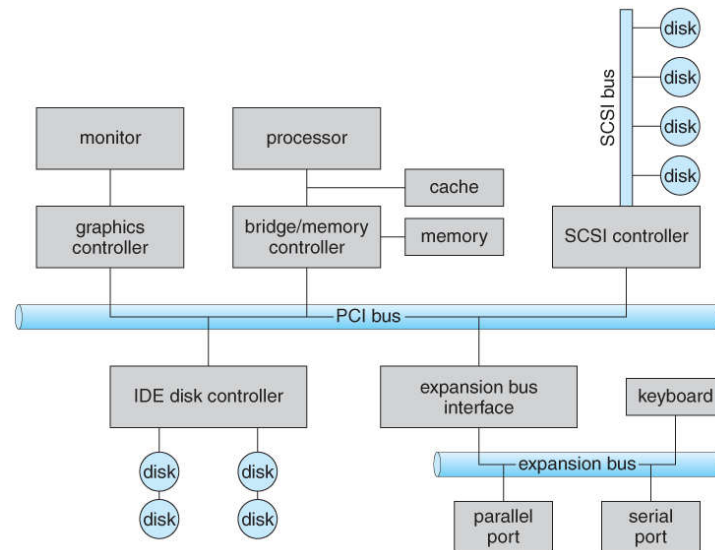


Figure - A typical PC bus structure.

- One way of communicating with devices is through **registers associated with each port**.
- Registers may be **one to four bytes in size**, and may typically include the following four:
 1. The **data-in register** is read by the host **to get input from the device**.
 2. The **data-out register** is written by the host **to send output**.
 3. The **status register** has bits read by the host to **ascertain the status of the device, such as idle, ready for input, busy, error**, transaction complete, etc.
 4. The **control register** has bits written by the host **to issue commands or to change settings of the device** such as parity checking, word length, or full- versus half-duplex operation.
- Figure below shows some of the most common I/O port address ranges.

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

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

- Another **technique for communicating with devices** is **memory-mapped I/O**.
 - In this case a certain **portion of the processor's address space is mapped to the device**, and communications occur by **reading and writing directly to/from those memory areas**.
 - Memory-mapped I/O is suitable for devices which must move **large quantities of data** quickly, such as **graphics cards**.
 - Memory-mapped I/O can be used either instead of or more often in combination with traditional registers. For example, graphics cards still use registers for control information such as setting the video mode.

Polling

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

Interrupts

- Interrupts allow **devices to notify the CPU** when they have **data to transfer or when an operation is complete**, allowing the CPU to perform other duties when no I/O transfers need its immediate attention.
- The CPU has an **interrupt-request line** that is sensed after every instruction.
 - A device's controller **raises** an interrupt by asserting a signal on the interrupt request line.
 - The CPU then performs a state save, and transfers control to the **interrupt handler** routine at a fixed address in memory. (The CPU **catches** the interrupt and **dispatches** the interrupt handler.)
 - The interrupt handler determines the cause of the interrupt, performs the necessary processing, performs a state restore, and executes a **return from**

interrupt instruction to return control to the CPU. (The interrupt handler **clears** the interrupt by servicing the device.)

- Figure below illustrates the interrupt-driven I/O procedure:

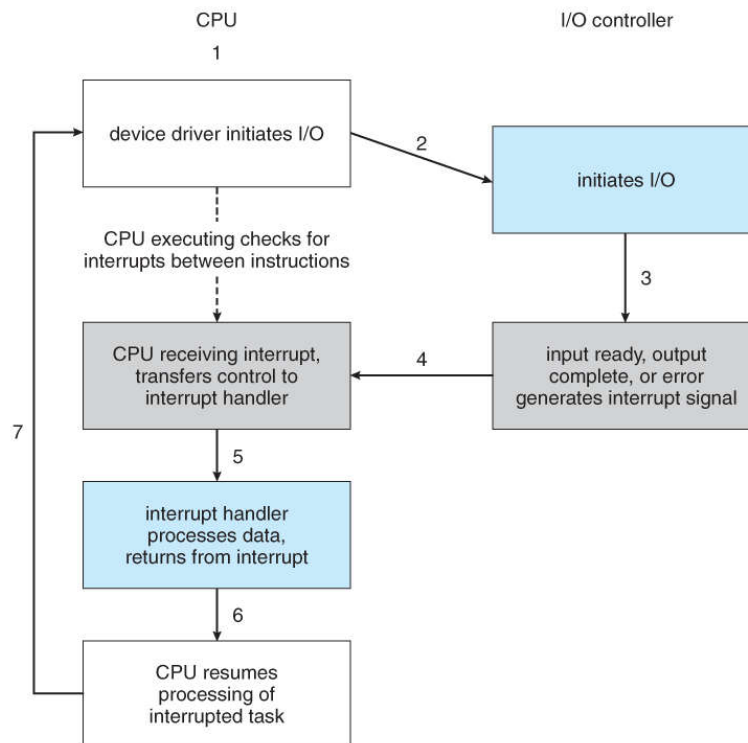


Figure - Interrupt-driven I/O cycle.

Most CPUs now have two interrupt-request lines: One that is **non-maskable** for critical error conditions and one that is **maskable**, that the CPU can temporarily ignore during critical processing.

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

At boot time the system determines which devices are present, and loads the appropriate handler addresses into the interrupt table.

During operation, devices signal errors or the completion of commands via interrupts.

Exceptions, such as dividing by zero, invalid memory accesses, or attempts to access kernel mode instructions can be signaled via interrupts.

Direct Memory Access (DMA)

For devices that **transfer large quantities of data** (such as disk controllers), it is **wasteful to tie up the CPU transferring data in and out of registers** one byte at a time.

Instead this work can be off-loaded to a special processor, known as the **Direct Memory Access, DMA, Controller**.

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

A simple DMA controller is a standard component in modern PCs, and many bus-mastering I/O cards contain their own DMA hardware.

Handshaking between DMA controllers and their devices is accomplished through two wires called the **DMA-request and DMA-acknowledge wires**.

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

DMA can be done in terms of either physical addresses or virtual addresses that are mapped to physical addresses. The latter approach is known as Direct Virtual Memory Access, DVMA, and allows direct data transfer from one memory-mapped device to another without using the main memory chips.

Direct DMA access by user processes can speed up operations, but is generally forbidden by modern systems for security and protection reasons.

Figure below illustrates the DMA process.

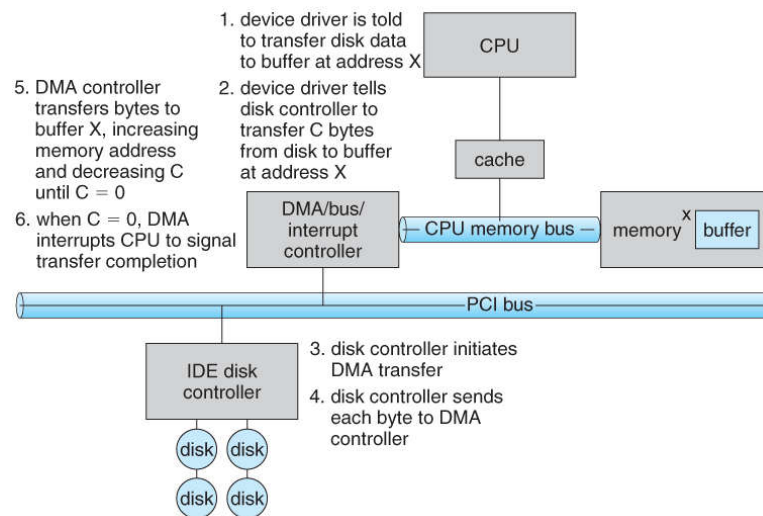


Figure - Steps in a DMA transfer.

Application I/O Interface

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

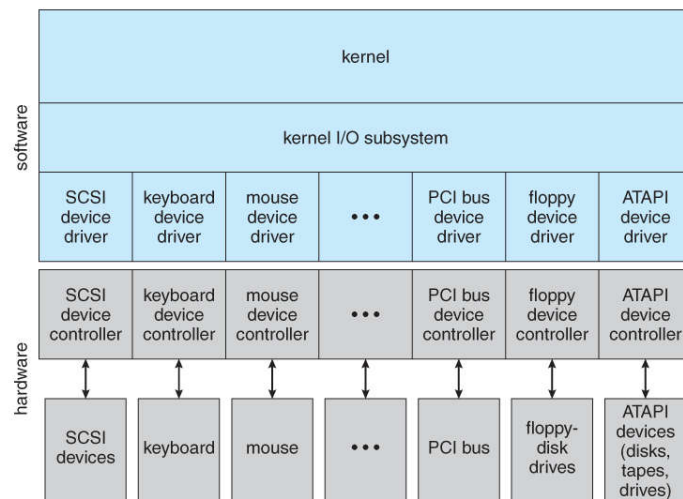


Figure - A kernel I/O structure.

Disk Management

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 is managed internally to the disk controller.
 3. Modern hard drives can have thousands of cylinders, and hundreds of sectors per track on their outermost tracks. These numbers exceed the range of HSC numbers for many (older) operating systems, and therefore disks can be configured for any convenient combination of HSC values that falls within the total number of sectors physically on the drive.
- There is a limit to how closely packed individual bits can be placed on a physical media, but that limit is growing increasingly more packed as technological advances are made.
- Modern disks pack many more sectors into outer cylinders than inner ones, using one of two approaches:
 - With **Constant Linear Velocity, CLV**, the density of bits is uniform from cylinder to cylinder. Because there are more sectors in outer cylinders, the disk spins slower when reading those cylinders, causing the rate of bits passing under the read-write head to remain constant. This is the approach used by modern CDs and DVDs.
 - With **Constant Angular Velocity, CAV**, the disk rotates at a constant angular speed, with the bit density decreasing on outer cylinders. (These disks would have a constant number of sectors per track on all cylinders.)

Disk Scheduling

- 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:

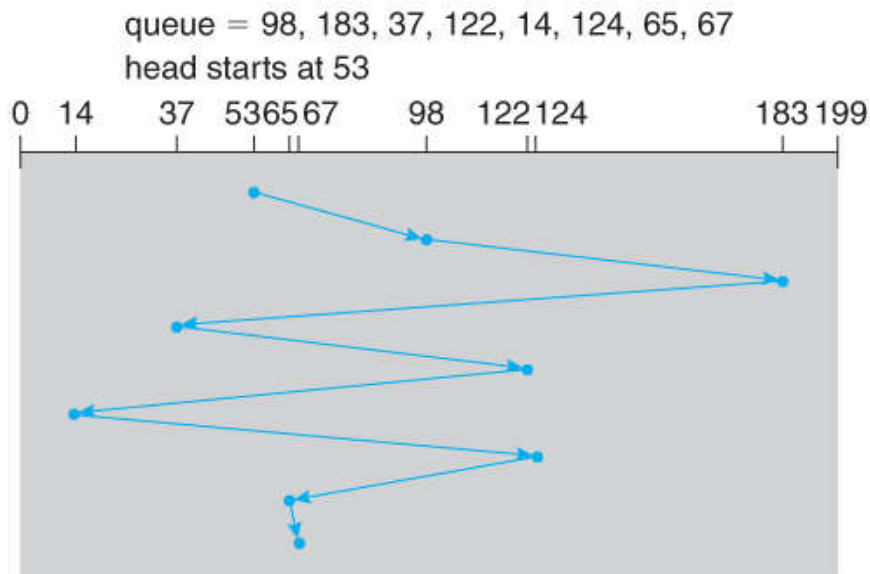


Figure - FCFS disk scheduling.

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.

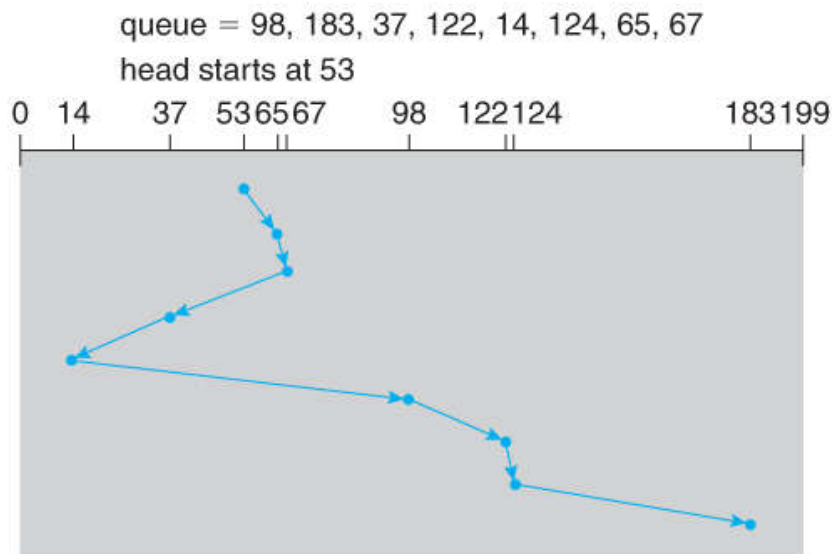


Figure - SSTF disk scheduling.

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.

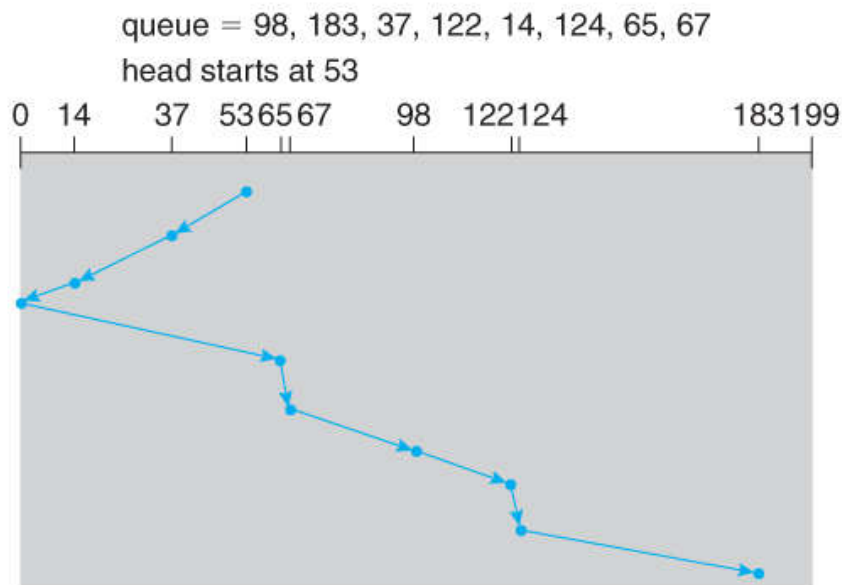


Figure - SCAN disk scheduling.

- 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:

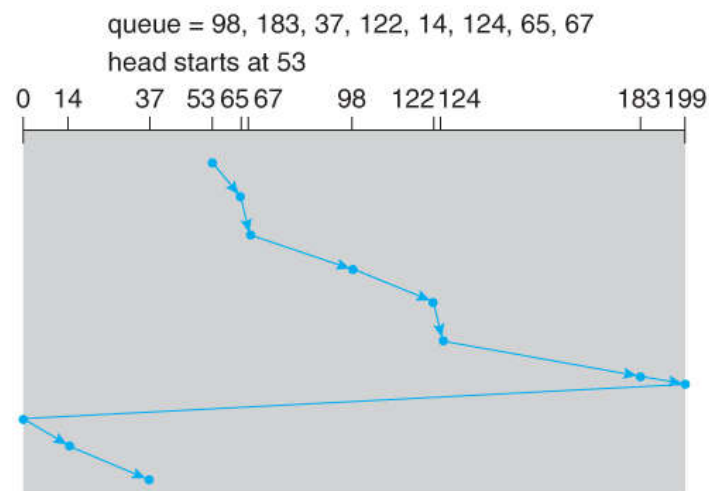


Figure - C-SCAN disk scheduling.

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:

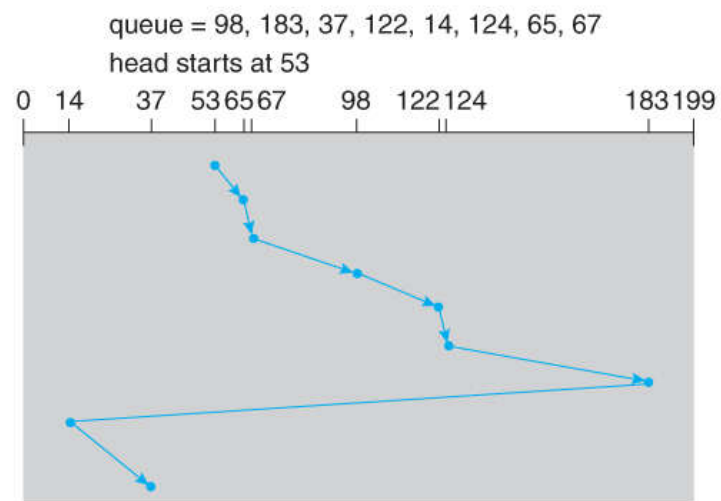


Figure - C-LOOK disk scheduling.

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 in ode 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.

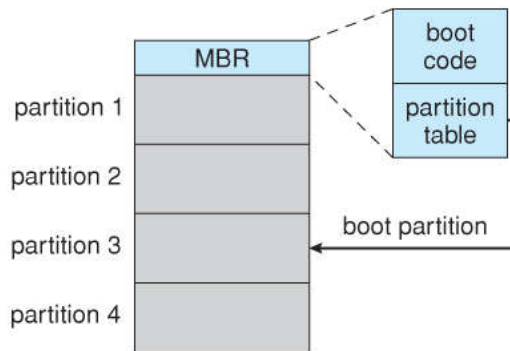


Figure - Booting from disk in Windows 2000.

Bad Blocks

- No disk can be manufactured to 100% perfection, and all physical objects wear out over time. For these reasons all disks are shipped with a few bad blocks, and additional blocks can be expected to go bad slowly over time. If a large number of blocks go bad then the entire disk will need to be replaced, but a few here and there can be handled through other means.
- In the old days, bad blocks had to be checked for manually. Formatting of the disk or running certain disk-analysis tools would identify bad blocks, and attempt to read the data off of them one last time through repeated tries. Then the bad blocks would be mapped out and taken out of future service. Sometimes the data could be recovered, and sometimes it was lost forever. (Disk analysis tools could be either destructive or non-destructive.)
- Modern disk controllers make much better use of the error-correcting codes, so that bad blocks can be detected earlier and the data usually recovered. (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.

Disk reliability

RAID Structure

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

Improvement of Reliability via Redundancy

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

Improvement in Performance via Parallelism

- There is also a performance benefit to mirroring, particularly with respect to reads. Since every block of data is duplicated on multiple disks, read operations can be satisfied from any available copy, and multiple disks can be reading different data blocks simultaneously in parallel. (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

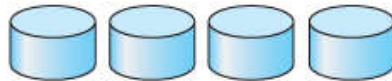
were involved, then two bits of each byte could be stored on each disk, for 2K worth of disk access per read or write operation.

- **Block-level striping** spreads a filesystem across multiple disks on a block-by-block basis, so if block N were located on disk 0, then block N + 1 would be on disk 1, and so on. This is particularly useful when filesystems are accessed in *clusters* of physical blocks. Other striping possibilities exist, with block-level striping being the most common.

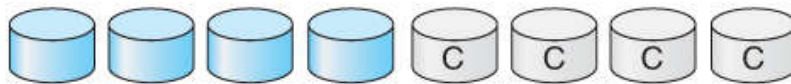
RAID Levels

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

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



(a) RAID 0: non-redundant striping.



(b) RAID 1: mirrored disks.



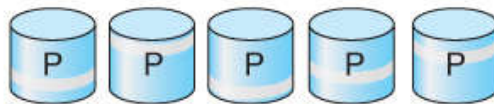
(c) RAID 2: memory-style error-correcting codes.



(d) RAID 3: bit-interleaved parity.



(e) RAID 4: block-interleaved parity.



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



(g) RAID 6: P + Q redundancy.

Figure - RAID levels.

File Management

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.).
 - **Identifier** (e.g. inode number)
 - **Type** - Text, executable, 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. 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 *mandatory lock* is enforced.
 - UNIX uses advisory locks and Windows uses mandatory locks.

File Types

- Windows 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, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

Figure - Common file types.

File Structure

Internal File Structure

- Disk files are accessed in units of physical blocks, typically 512 bytes or some power-of-two multiple thereof.
- 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 (File access methods)

1. 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.
 - **rewind**
 - **skip n records** - May or may not be supported. N may be limited to positive numbers, or may be limited to +/- 1.

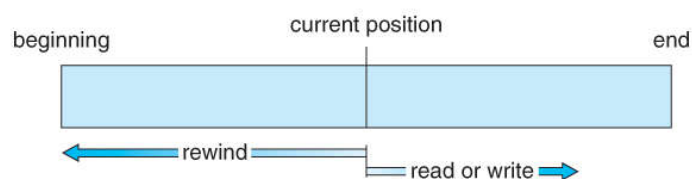


Figure - Sequential-access file.

2. Direct Access

- Jump to any record and read that record. Operations supported include:
 - **read n** - read record number n.
 - **write n** - write record number n.
 - **jump to record n** - could be 0 or the end of file.
 - **Query current record** - used to return back to this record later.

3. Indexed Access Method

- 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.

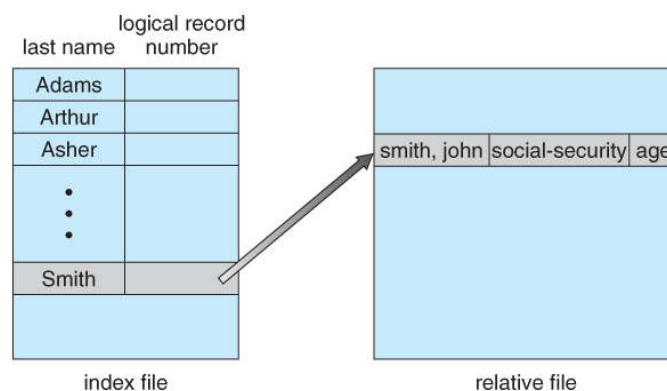


Figure - Example of index and relative files.

Directory Structure

Storage Structure

- A disk can be used in its entirety for a file system.
- A physical disk can be broken up into multiple *partitions*, each of which becomes a virtual disk and can have its own filesystem.
- Or, multiple physical disks can be combined into one *volume*, i.e. a larger virtual disk, with its own filesystem spanning the physical disks.

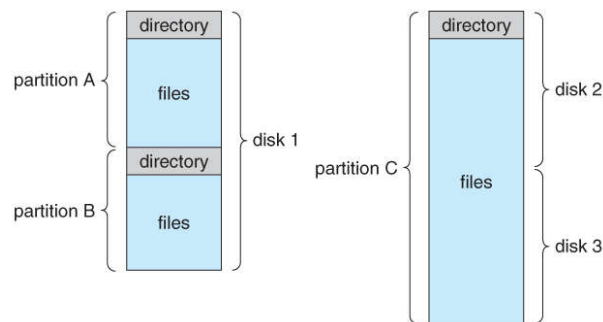


Figure - A typical file-system organization.

Directory Structure

- 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.

1. Single-Level Directory

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

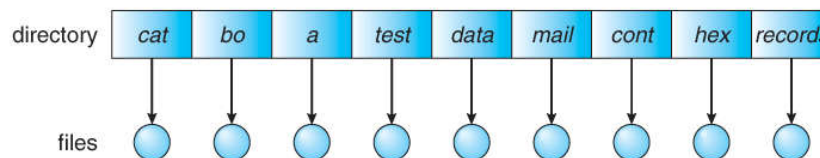


Figure - Single-level directory.

2. 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 users 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.

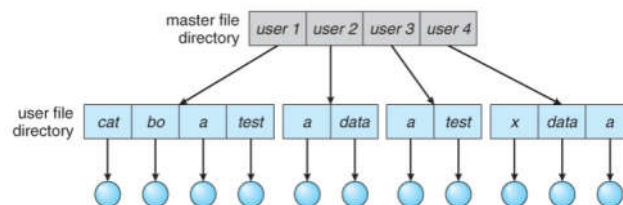


Figure - Two-level directory structure.

3. Tree-Structured Directories

- An extension to the two-tiered directory structure
- 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.

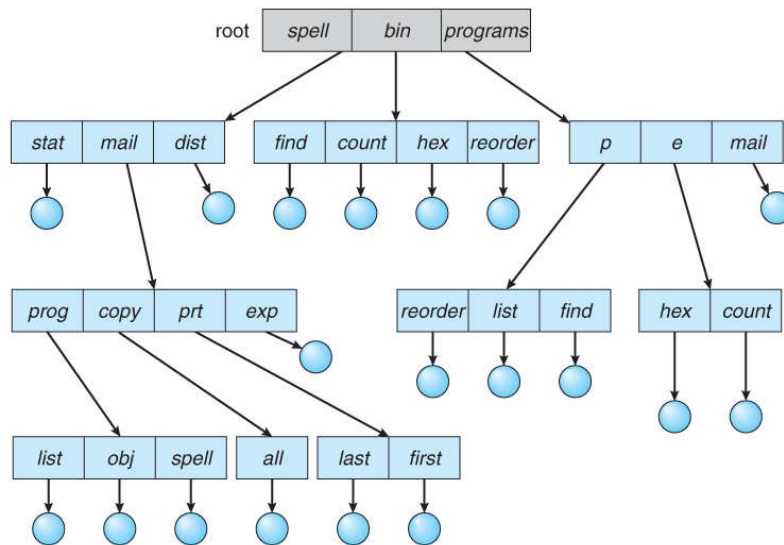


Figure - Tree-structured directory structure.

4. 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.
 - A **hard link** involves multiple directory entries that both refer to the same file. Hard links are only valid for ordinary files in the same filesystem.
 - 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 filesystems, as well as ordinary files in the current filesystem.
- 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?

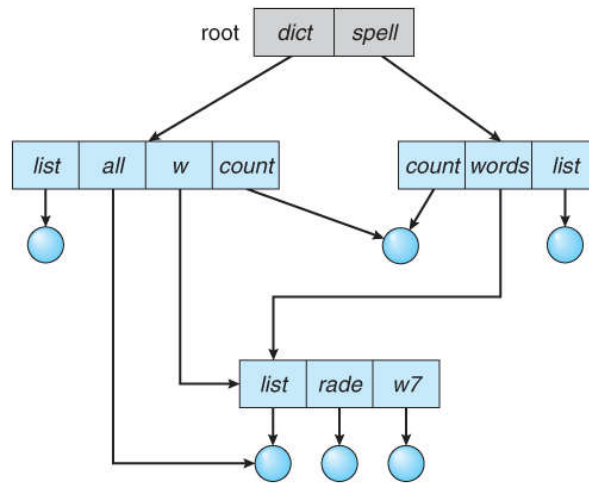


Figure - Acyclic-graph directory structure.

5. 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.
 - 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.

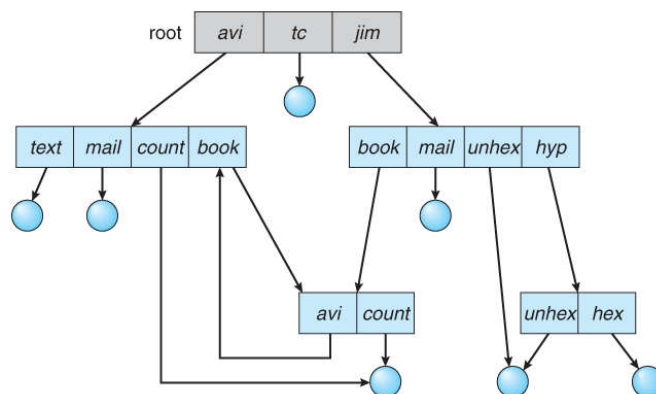


Figure - General graph directory.

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 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 filesystem 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

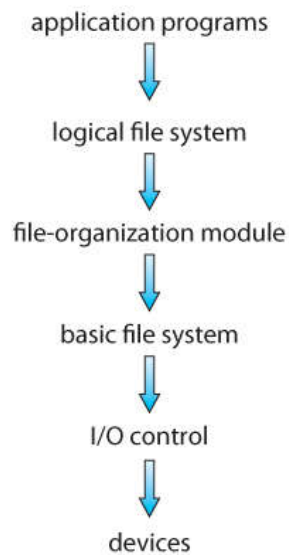


Figure - Layered file system.

File Allocation Methods

- There are three major methods of storing files on disks: contiguous, linked, and indexed.

1. 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.

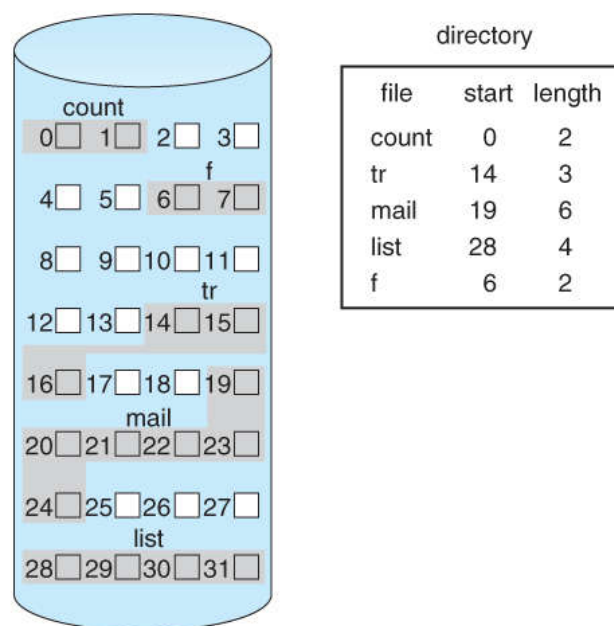


Figure - Contiguous allocation of disk space.

2. 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.

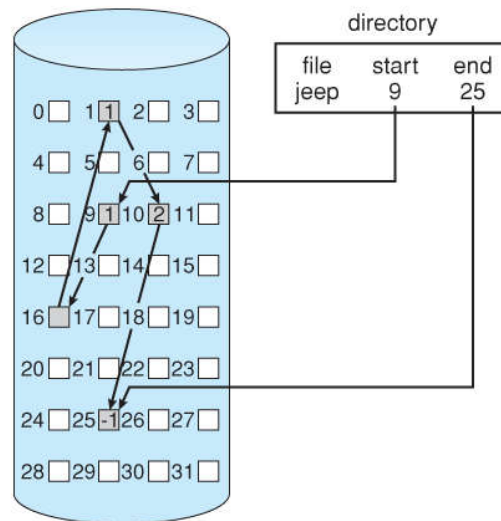


Figure - Linked allocation of disk space.

- 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.

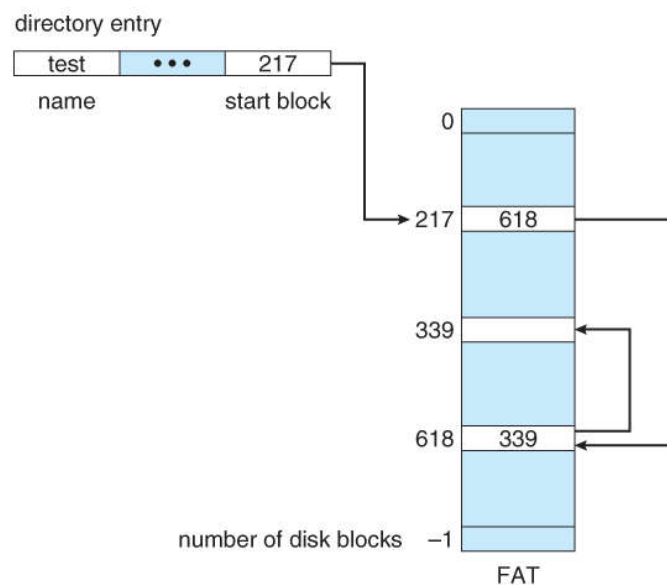


Figure - File-allocation table.

3. 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.

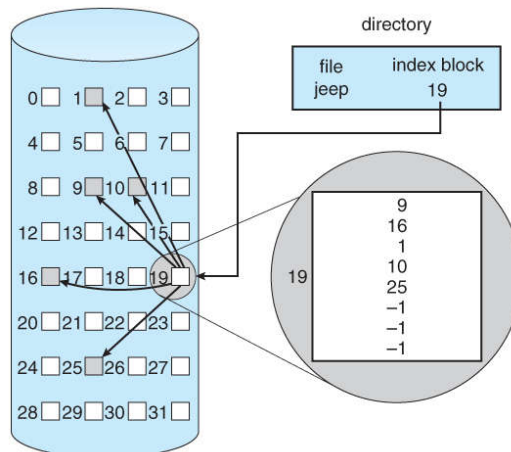


Figure - Indexed allocation of disk space.

- 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. 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.)

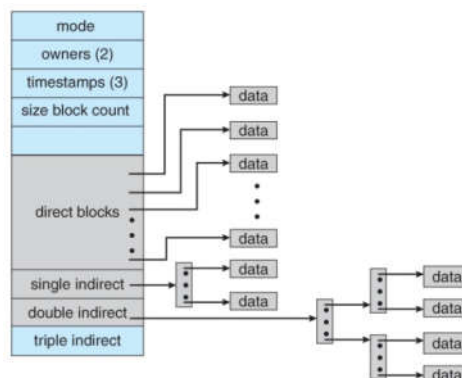


Figure - The UNIX inode.

Free-Space Management

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

1. 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.)

2. 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.

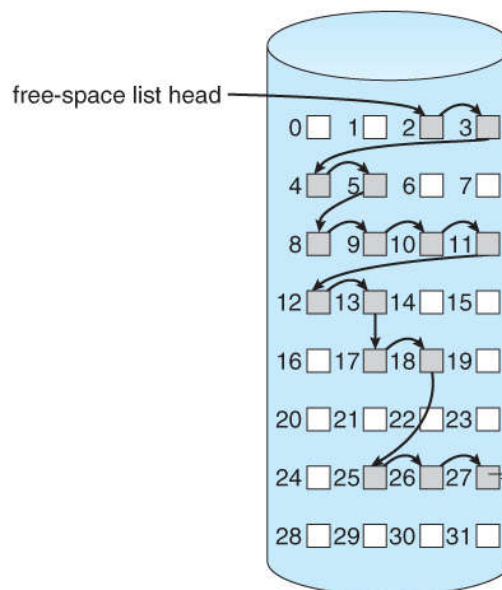


Figure - Linked free-space list on disk.

3. 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.

4. 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.)

5. 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.

Directory Implementation

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

1. 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.

2. 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

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.

Efficiency and Performance

1. 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.

2. 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 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.

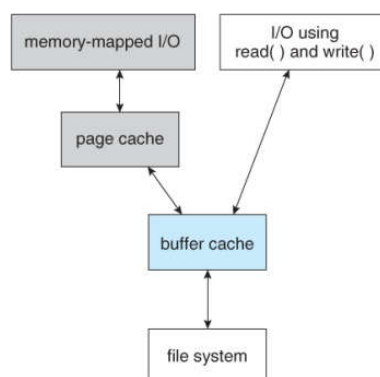


Figure - I/O without a unified buffer cache.

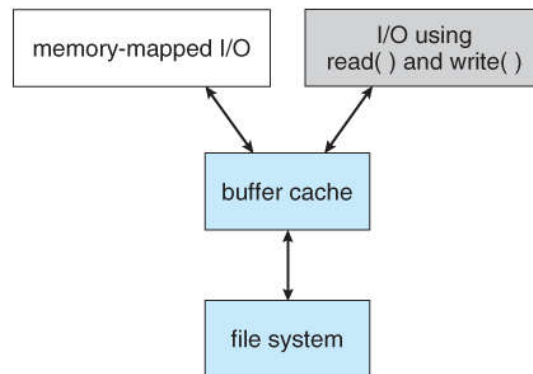


Figure - I/O using a unified buffer cache.

- 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. (See Chapter 12.) 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.