JEPPIAAR INSTITUTE OF TECHNOLOGY

"Self-Belief | Self Discipline | Self Respect"

DEPARTMENT

OF

COMPUTER SCIENCE AND ENGINEERING

LECTURE NOTES

CS8493 – OPERATING SYSTEM

(Regulation 2017)

Year/Semester: II / 04 CSE

2020 – 2021

Prepared by

Dr J FARITHA BANU

Professor / CSE

UNIT IV - FILE SYSTEMS AND I/O SYSTEMS

File Systems provides the mechanism for on-line storage and access to both data and programs of the operating system and all the users of the computer system. This chapter describes Mass Storage system, Disk Structure, Scheduling, Swap Space Management, File-System Concept, Directory implementation, Allocation Methods, Free Space Management, I/O Systems, Hardware, Interface, Kernel I/O subsystem, Streams, Performance.

Mass Storage Structure- Overview

Most computer systems provide secondary storage (Mass Storage) as an extension of main memory. The main requirement for secondary storage is that it be able to hold large quantities of data permanently (Non-Volatile memory). The most common secondary-storage device is a magnetic disk, which provides storage for both programs and data. Most of the secondary storage devices are internal to the computer such as the hard disk drive, the tape disk drive and even the compact disk drive and floppy disk drive.

Magnetic Disks

Magnetic disks provide the bulk of secondary storage for modern computer systems. Each disk platter has a flat circular shape, like a CD. Common platter diameters range from 1.8 to 3.5 inches. The two surfaces of a platter are covered with a magnetic material. We store information by recording it magnetically on the platters. A read–write head flies just above each surface of every platter. The heads are attached to a disk arm that moves all the heads as a unit. The surface of a platter is logically divided into circular tracks, which are subdivided into sectors.

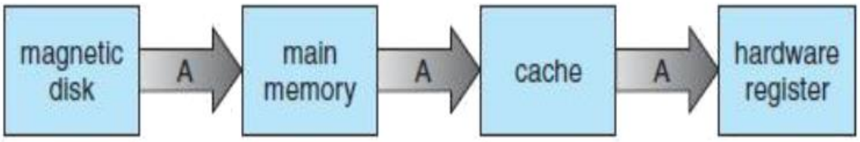


Fig 4.1 Magnetic Disk

CYLINDER: The set of tracks that are at one arm position makes up a cylinder

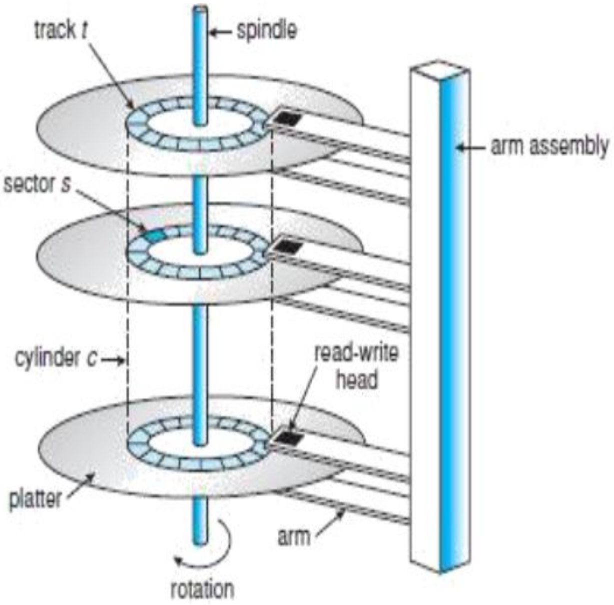


Fig 4.2 Moving Head Disk Mechanism

The storage capacity of common disk drives is measured in gigabytes. When the disk is in use, a drive motor spins it at high speed. Most drives rotate 60 to 250 times per second, specified in terms of rotations per minute.

Magnetic Tapes

Magnetic tape was used as an early secondary-storage medium. It is relatively permanent and can hold large quantities of data. Its access time is slow compared with that of main memory and magnetic disk.

In addition, random access to magnetic tape is about a thousand times slower than random access to magnetic disk, so tapes are not very useful for secondary storage. Tapes are used mainly for backup, for storage of infrequently used information, and as a medium for transferring information from one system to another.

Disk Structure

Magnetic disk drives are addressed as large one-dimensional arrays of logical blocks, where the logical block is the smallest unit of transfer. The size of a logical block is usually 512 bytes, although some disks can be low-level formatted to have a different logical block size, such as 1,024 bytes. The one-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially. Sector 0 is the first sector of the first track on the outermost cylinder. The number of sectors per track is not constant on some drives.

For the disks that use **Constant Linear Velocity** (CLV), the density of bits per track is uniform. In **Constant Angular Velocity** (CAV) the density of bits decreases from inner tracks to outer tracks to keep the data rate constant.

Disk Scheduling

Whenever a process needs I/O to or from the disk, it issues a system call to the operating system. If the desired disk drive and controller are available, the request can be serviced immediately. If the drive or controller is busy, any new requests for service will be placed in the queue of pending requests for that drive. When one request is completed, the operating system chooses which pending request to service next. This is known as **Disk Scheduling**.

The request specifies several pieces of information:

- Whether this operation is input or output
- What the disk address for the transfer is
- What the memory address for the transfer is
- What the number of sectors to be transferred is

Disk Components

The three major components of the hard disk are Seek time and Rotational Latency and bandwidth.

Seek time: The seek time is the time for the disk arm to move the heads to the cylinder containing the desired sector.

Rotational latency: The rotational latency is the additional time for the disk to rotate the desired sector to the disk head.

Disk bandwidth: The disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.

Disk Scheduling Algorithms

- First Come First Serve
- Shortest Seek Time First
- Scan Algorithm
- Circular Scan Algorithm
- Look Algorithm
- Circular Look Algorithm

FCFS Scheduling:

The simplest form of disk scheduling is, of course, the **first-come, first-served (FCFS)** algorithm.

Example: Consider, for example, Given a disk with 200 cylinders and a disk queue with requests 98, 183, 37, 122, 14, 124, 65, 67, for I/O to blocks on cylinders. Disk head is initially at 53.

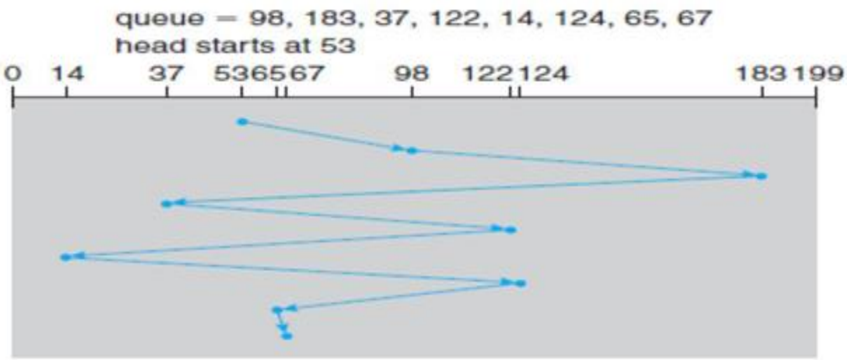


Fig 4.3 FCFS Scheduling

Current Block	Next Block to Be Accessed	No. of Head Movement for Each Access
53	98	45
98	183	85
183	37	146
37	122	85
122	14	108
14	124	110
124	65	59
65	67	02
Total head movements		**640**

Fig 4.4 Total Head Movements Calculation for FCFS

Total No of head Movements of for FCFS is 640.

Disadvantage:

Unnecessary head movements is possible. Ex : The request from 122 to 14 and then back to 124 increases the total head movements.

SSTF Scheduling:

The **shortest-seek-time-first (SSTF)** algorithm selects the request with the least seek time from the current head position. It chooses the pending request closest to the current head position.

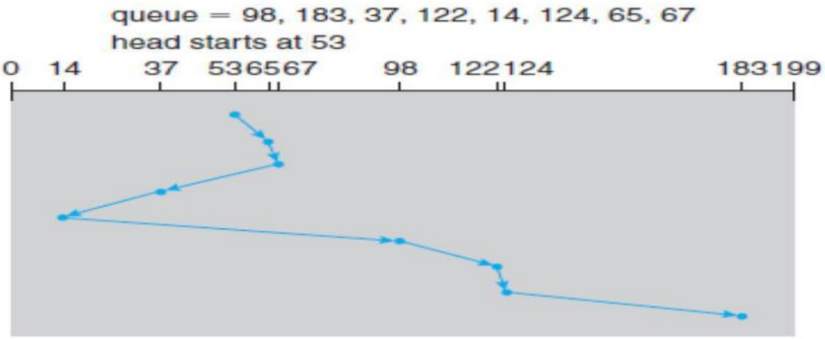


Fig 4.5 SSTF Scheduling

Disadvantage:

SSTF may cause starvation of some requests.

SCAN Scheduling:

In the **SCAN** algorithm, the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. The SCAN algorithm is sometimes called the elevator algorithm.

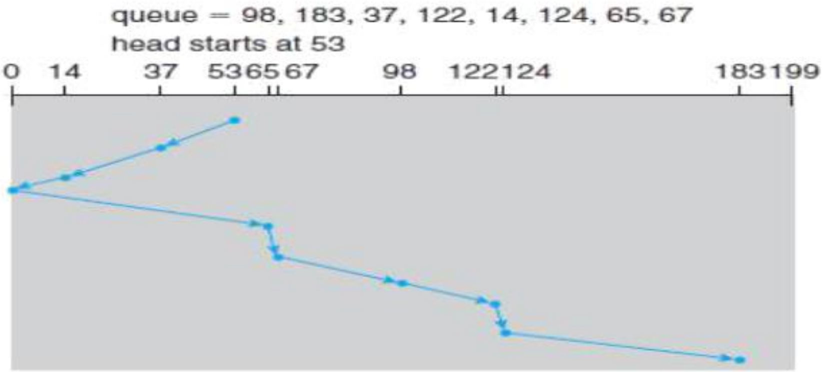


Fig 4.6 SCAN Scheduling

Current Block	Next Block to Be Accessed	No. of Head Movement for Each Access
53	37	16
37	14	23
14	0	14
0	65	65
65	67	02
67	98	31

98	122	24
122	124	02
124	183	59
Total head movements		**236**

Fig 4.7 Total Head Movements Calculation for SCAN

Total No of head Movements of for SCAN is 236. Similarly Calculate for all other algorithm.

Circular SCAN Algorithm:

Circular SCAN (C-SCAN) scheduling is a variant of SCAN designed to provide a more uniform wait time.

C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk **without servicing any requests on the return trip**.

The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.

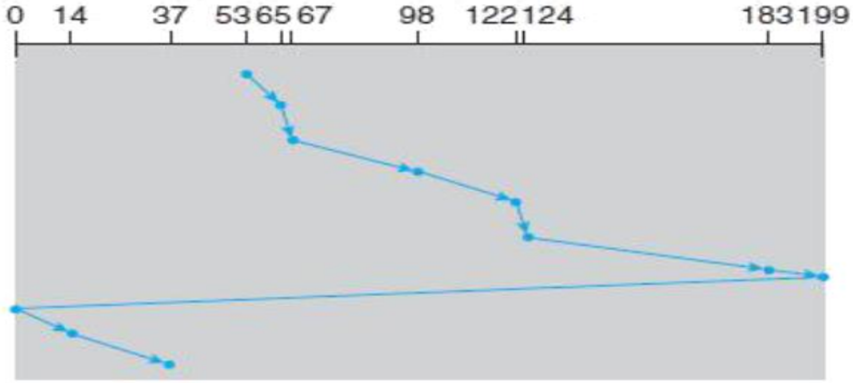


Fig 4.8 Total Head Movements Calculation for SCAN

LOOK scheduling:

The LOOK algorithm is the same as the SCAN algorithm in that it also services the requests on both directions of the disk head, but it **—Looks" ahead** to see if there are any requests pending in the direction of head movement.

If no requests are pending in the direction of head movement, then the disk head traversal **will be reversed to the opposite direction** and requests on the other direction can be served.

In LOOK scheduling, the arm goes only as far as final requests in each direction and then reverses direction without going all the way to the end

C-LOOK Scheduling:

This is just an enhanced version of C-SCAN.

Arm only goes as far as the last request in each direction, then reverses direction immediately, without servicing all the way to the end of the disk and then turns the next direction to provide the service.

Disk Management

The operating system is responsible for disk management. The Major Responsibility includes **Disk Formatting, Booting from Disk, Block Recovery.**

Disk Formatting:

The Disk can be formatted in two ways,

1. Physical or Low Level Formatting,

2. Logical or High Level Formatting

Physical or Low Level Formatting:

Before a disk can store data, it must be divided into sectors that the disk controller can read and write. This process is called low-level formatting, or physical formatting.

When the sector is read, the ECC is recalculated and compared with the stored value. If the stored and calculated numbers are different, this mismatch indicates that the data area of the sector has become corrupted and that the disk sector may be bad. It then reports a recoverable soft error.

Logical Formatting or High Level Formatting:

The operating record its own data structures on the disk during Logical formatting. It does so in two steps. The first step is to partition the disk into one or more groups of cylinders. The second step is logical formatting, or creation of a file system. In this step, the operating system stores the initial file-system data structures onto the disk. These data structures may include maps of free and allocated space and an initial empty directory.

Booting from Disk: The full bootstrap program is stored in the boot blocks at a fixed location on the disk. A disk that has a boot partition is called a boot disk or system disk.

The code in the boot ROM instructs the disk controller to read the boot blocks into memory and then starts executing that code which in turn loads the entire Operating System.

Block Recovery:

A **bad block** is a damaged area of magnetic storage media that cannot reliably be used to store and retrieve data. These blocks are handled in a variety of ways.

One strategy is to **scan the disk to find bad blocks** while the disk is being formatted. Any bad blocks that are discovered are flagged as unusable so that the file system does not allocate them.

In Some systems the controller maintains a list of bad blocks on the disk. This can be handled in two ways.

Sector Sparing: Low-level formatting also sets aside spare sectors not visible to the operating system. The controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as **sector sparing** or **forwarding**

Sector Slipping: The Process of moving all the sectors down one position from the bad sector is called as sector slipping.

Swap Space Management

Swap-space management is another low-level task of the operating system. Virtual memory uses disk space as an extension of main memory. Since disk access is much slower than memory access, Swap Space management is designed to provide the best throughput for the virtual memory system.

Systems that implement swapping may use swap space to hold an entire process image, including the code and data segments. The amount of swap space needed on a system can therefore vary from a few megabytes of disk space to gigabytes, depending on the amount of physical memory, the amount of virtual memory it is backing, and the way in which the virtual memory is used.

Each swap area consists of a series of 4-KB page slots, which are used to hold swapped pages. Associated with each swap area is a swap map—an array of integer counters, each corresponding to a page slot in the swap area. If the value of a counter is 0, the corresponding page slot is available.

Values greater than 0 indicate that the page slot is occupied by a swapped page. The value of the counter indicates the number of mappings to the swapped page. The data structures for swapping on Linux systems are shown in Figure.

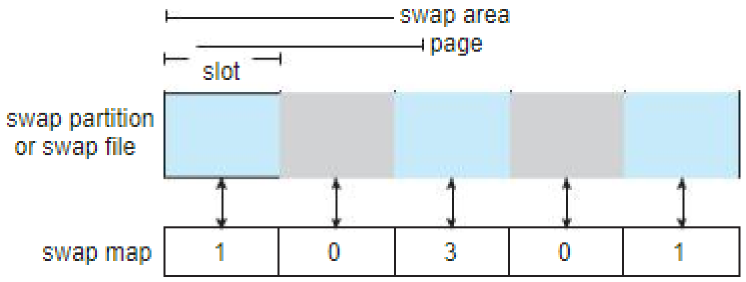


Fig 4.9 The data structures for swapping on Linux systems

For example, a value of 3 indicates that the swapped page is mapped to three different processes (which can occur if the swapped page is storing a region of memory shared by three processes.

File System Interface

File Concept : A file is defined as a **named collection of related information** that is stored on secondary storage device. Many different types of information may be stored in a file such as source or executable programs, numeric or text data, photos, music, video, and so on. A file has a certain defined structure, which depends on its type.

Types of Files:

- A **text file** is a sequence of characters organized into lines.
- A **Source file** is a sequence of functions, each of which is further organized as declarations followed by executable statements.
- An **Executable file** is a series of code sections that the loader can bring into memory and execute.

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
Point or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
Archive	rar, zip, tar	related files are grouped, compressed for archiving or storage
Multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

Fig 4.10 File Types

FILE ATTRIBUTES:

A file's attributes vary from one operating system to another but typically consist of these:

Name: The file name is the information kept in human readable form.

Identifier: This unique tag, usually a number, identifies the file within the file system

Type: This information is needed for systems that support different types of files.

Location: This information is a pointer to a device and to the location of the file on that device.

Size: The current size of the file (in bytes, words, or blocks)

Protection: Access-control information determines who can do reading, writing, executing

Time, date, and user identification: This information may be kept for creation, last modification, and last use.

FILE OPERATIONS:

A file is an abstract data type. The operating system can provide system calls to create, write, read, reposition, delete, and truncate files. The Basic Operations on a file includes

- Creating a File
- Writing a File
- Reading a File
- Repositioning within a File
- Deleting a file
- Truncating a file

FILE LOCKS:

File locks allow one process to lock a file and prevent other processes from gaining access to it. File locks are useful for files that are shared by several processes. 2 types of locks are

Shared Lock: A **shared lock** is similar to a reader lock in that several processes can acquire the lock concurrently.

Exclusive Lock: An **exclusive lock** behaves like a writer lock; only one process at a time can acquire such a lock.

File Access Methods

Files store information. The information in the file can be accessed in several ways.

Sequential access Method: The simplest access method is sequential access. Information in the file is processed in order, one record after the other.

Direct Access Method: A file is made up of fixed-length logical records that allow programs to read and write records rapidly in no particular order.

Indexed Access Methods: These methods generally involve the construction of an index for the file. The index contains pointers to the various blocks. To find a record in the file, we first search the index and then use the pointer to access the file directly and to find the desired record. With large files, the index file itself may become too large to be kept in memory.

One solution is to create an index for the index file. The primary index file contains pointers to secondary index files, which point to the actual data items.

File Systems Mounting

File System Mounting is defined as the process of attaching an additional file system to the currently accessible file system of a computer. **A file system** is a hierarchy of directories that is used to organize files on a computer or storage media. The operating system is given the name of the device and the **mount point**.

Mount Point: It is the location within the file structure where the file system is to be attached. A mount point is an empty directory.

Example: A file system containing a user's home directories might be mounted as /home. To access the directory structure within that file system, we could precede the directory names with /home, as in /home/Jane. Mounting that file system under /users would result in the path name /users/Jane, which we could use to reach the same directory.

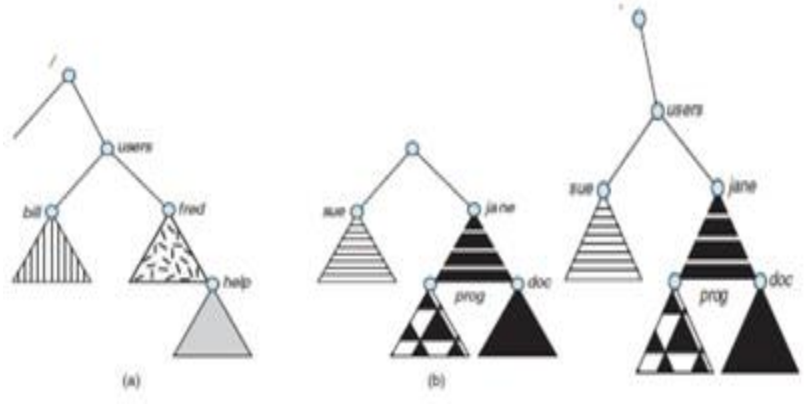


Fig 4.11 File System (a) Existing System (b) Un mounted Volume (C) Mount Point

File Sharing

File Sharing: File sharing is very important for users who want to cooperate their files with each other and to reduce the effort required to achieve a computing goal**.**

File sharing includes

- Multiple users
- Remote File Systems

- Client server model
- Distributed Information systems
- Failure Modes
- Consistency semantics - Unix Semantics, Session Semantics, Immutable Shared File Semantics.

Multiple Users:

The system with multiple users can either allow a user to access the files of other users by default or require that a user specifically grant access to the files.

The systems uses the concepts of file **owner** (or **user**) and **group for File sharing.** The owner is the user who can change attributes and grant access and who has the most control over the file. The group attribute defines a subset of users who can share access to the file.

The owner and group IDs of a given file are stored with the other file attributes. When a user requests an operation on a file, the user ID can be compared with the owner attribute to determine if the requesting user is the owner of the file.

If he is not the owner of the file, the group IDs can be compared. The result indicates which permissions are applicable. The system then applies those permissions to the requested operation and allows or denies it.

Remote File Systems:

Networking allows the sharing of resources across a campus or even around the world.

The first implemented method for remote file systems involves manually transferring files between machines via programs like FTP. The second major method uses a **distributed file system (DFS)** in which remote directories is visible from a local machine. The third method is the **World Wide Web** where the browser is needed to gain access to the remote files.

Client Server Model:

The machine containing the files is the **server**, and the machine seeking access to the files is the **client.** The server declares that a resource is available to clients and specifies exactly which resource is shared by which clients. In the case of UNIX and its network file system (NFS), authentication takes place via the client networking information. The user's IDs on the client and server must match. If they do not, the server will be unable to determine access rights to files.

Distributed Information Systems:

Distributed information systems, also known as **distributed naming services**, provide unified access to the information needed for remote computing. The **domain name system**

(DNS) provides host-name-to-network-address translations for the entire Internet. Distributed information systems provide **user name/password/user ID/group ID** space for a distributed facility.

In the case of Microsoft's **common Internet file system (CIFS)**, network information is used in conjunction with user authentication to create a network login that the server uses to decide whether to allow or deny access to a requested file system.

Failure Modes:

Local file systems can fail for a variety of reasons that includes

- Failure of the disk containing the file system,
- Corruption of the directory structure or other disk-management information
- Disk-controller failure,
- Cable failure,
- Host-adapter failure
- User or system-administrator failure
- Remote file systems have more failure modes because of the complexity of network systems and the required interactions between remote machines.

The failure semantics are defined and implemented as part of the remote-file-system protocol. Termination of all operations can result in users' losing data. To implement the recovery from failure, some kind of **state information** may be maintained on both the client and the server. If both server and client maintain knowledge of their current activities and open files, then they can recover from a failure.

Consistency Semantics:

Consistency semantics represent a criterion for evaluating any file system that supports file sharing. The semantics specify how multiple users of a system are to access a shared file simultaneously.

They specify when modifications of data by one user will be observable by other users. Consistency semantics are directly related to the process synchronization algorithms.

A series of file accesses attempted by a user to the same file is always enclosed between the open() and close() operations.

The series of accesses between the open() and close() operations makes up a **file session**.

The Examples of Consistency semantics includes

Unix Semantics: Writes to an open file by a user are visible immediately to other users who have this file open. One mode of sharing allows users to share the pointer of current location into the file. Thus, the advancing of the pointer by one user affects all sharing users

Session Semantics: Writes to an open file by a user are not visible immediately to other users that have the same file open. Once a file is closed, the changes made to it are visible only in sessions starting later. Already open instances of the file do not reflect these changes

Immutable Shared File Semantics:

Once a file is declared as shared by its creator, it cannot be modified. An immutable file has two key properties: its name may not be reused, and its contents may not be altered. An immutable file signifies that the contents of the file are fixed.

File Protection

Protection mechanisms provide controlled access by limiting the types of file access that can be made. Access is permitted or denied depending on several factors, one of which is the type of access requested.

File Protection is also defined as the process of protecting the file of a user from **unauthorized access or any other physical damage**.

Goals of Protection:

To prevent malicious misuse of the system by users or programs.

To ensure that each shared resource is used only in accordance with system policies, which may be set either by system designers or by system administrators.

To ensure that errant programs cause the minimal amount of damage possible.

Types of Access:

The need to protect files is a direct result of the ability to access files. Systems that do not permit access to the files of other users do not need protection Several different types of operations may be controlled:

- Read.
- Write.
- Execute.
- Append.
- Delete.
- List.

Access-control list (ACL): specifying user names and the types of access allowed for each user.

Owner: The user who created the file is the owner.

Group: A set of users who are sharing the file and need similar access

Universe: All other users in the system constitute the universe

Password Protection: Another approach to the protection problem is to associate a password with each file. Access to each file can be controlled with the help of passwords. If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file.

Disadvantages:

- The number of passwords that a user needs to remember may become large.
- If only one password is used for all the files, then once it is discovered, all files are accessible; protection is on an all-or-none basis.
- In a multilevel directory structure, we need to protect not only individual files but also collections of files in subdirectories.

File System Structure

The file system provides the mechanism for on-line storage and access to file contents, including data and programs. The file system resides permanently on secondary storage, which is designed to hold a large amount of data permanently.

A file system poses two quite different design problems.

- The first problem is defining how the file system should look to the user. This task involves defining a file and its attributes, the operations allowed on a file, and the directory structure files.
- The second problem is creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.

The file system itself is generally composed of many different levels.

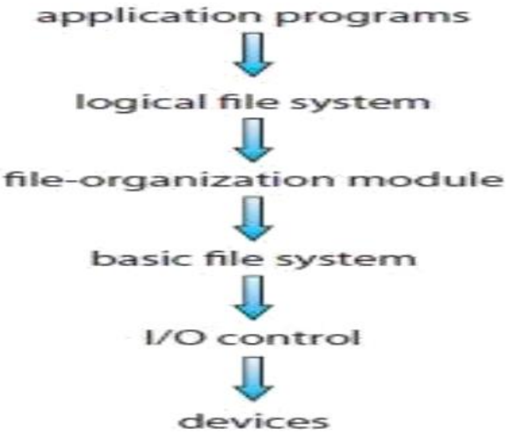


Fig 4.12 File System Structures

I/O Control: The **I/O control** level consists of device drivers and interrupts handlers to transfer information between the main memory and the disk system.

The device driver usually writes specific bit patterns to special locations in the I/O controller's memory to tell the controller which device location to act on and what actions to take.

Basic File System: The **basic file system** needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk. Each physical block is identified by its numeric disk address.

This layer also manages the memory buffers and caches that hold various file-system, directory, and data blocks. A block in the buffer is allocated before the transfer of a disk block can occur. Caches are used to hold frequently used file-system metadata to improve performance.

File Organization Module:

The **file-organization module** knows about files and their logical blocks, as well as physical blocks. The file-organization module can translate logical block addresses to physical block addresses for the basic file system to transfer. The file-organization module also includes the free-space manager, which tracks unallocated blocks and provides these blocks to the file-organization module when requested.

Logical File System: The **logical file system** manages metadata information. The logical file system manages the directory structure to provide the file-organization module with the information it needs.

It maintains file structure via file-control blocks

A **file control block (FCB)** (an **inode** in UNIX file systems) contains information about the file, including ownership, permissions, and location of the file contents.

Advantages of Layered File system:

- When a layered structure is used for file-system implementation, duplication of code is minimized.
- Each file system can then have its own logical file-system and file-organization modules.

Disadvantages:

- The use of layering, including the decision about how many layers to use and what each layer should do, is a major challenge in designing new systems.

Directory Implementation

Directory can be implemented in two ways

Linear List: The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks. This method is simple to program but time-consuming to execute.

To create a new file, we must first search the directory to be sure that no existing file has the same name. Then, we add a new entry at the end of the directory.

To delete a file, we search the directory for the named file and then release the space allocated to it. To reuse the directory entry, we can do one of several things.

Disadvantage:

- Finding a file requires a linear search.

Hash Table: The hash table takes a value computed from the file name and returns a pointer to the file time. Some provision must be made for collisions situations in which two file names hash to the same location.

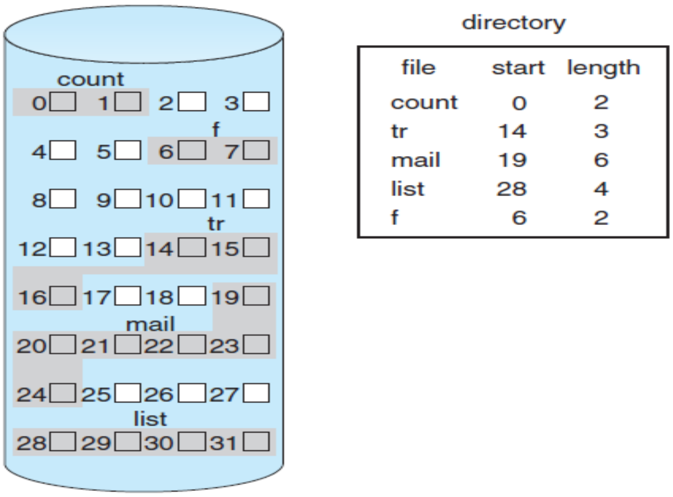
The major difficulties with a hash table are its generally fixed size and the dependence of the hash function on that size. Alternatively, we can use a chained-overflow hash table.

Each hash entry can be a linked list instead of an individual value, and we can resolve collisions by adding the new entry to the linked list. Still, this method is likely to be much faster than a linear search through the entire directory.

Directory Allocation methods

Contiguous allocation requires that each file occupy **a set of contiguous blocks on the disk**. Disk addresses define a linear ordering on the disk. It supports both **direct and sequential access**. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.

For Example: File name mail requires length of 6 blocks, Staring block is 19, then it occupies 19, 20, 21,22, 23, 24 and 25 contiguous blocks.



file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

Fig 4.13 Contiguous allocation

Advantages:

- Accessing a file that has been allocated contiguously is easy.

- The number of disk seeks required for accessing a file is minimal.
- It supports both **direct and sequential access**.

Disadvantages:

- Finding space for a new file.
- Contiguous memory allocation suffers from the problem of external fragmentation.

External Fragmentation: The total available space may not be enough to satisfy a request. Storage is fragmented into a number of holes, none of which is large enough to store the data.

Linked allocation:

Linked allocation solves all problems of contiguous allocation. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. Each directory entry has a pointer to the first disk block of the file. By accessing the First block, we can find the address of the next blocks. To read a file, we simply read blocks by following the pointers from block to block.

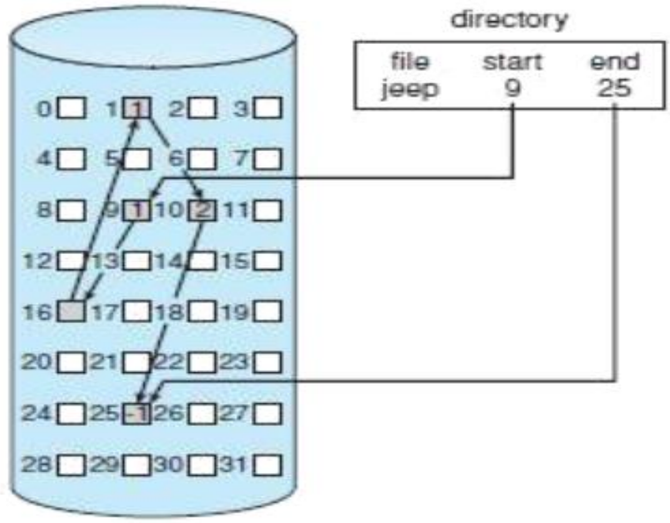


Fig 4.14 Linked allocation

Advantages:

- There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request.
- The size of a file need not be declared when the file is created.
- A file can continue to grow as long as free blocks are available.

Disadvantages:

- It is inefficient to support a direct-access capability for linked-allocation files.

- It requires more disk space for storing the pointers.

Solution: The usual solution to this problem is to collect blocks into multiples, called **clusters**, and to allocate clusters rather than blocks.

Indexed Allocation

In Indexed allocation each file has its own index block, which is an array of disk-block addresses. The directory contains the address of the index block. The index block stores all blocks corresponding to the specific file. By accessing the index block, we can access all the blocks for a specific file.

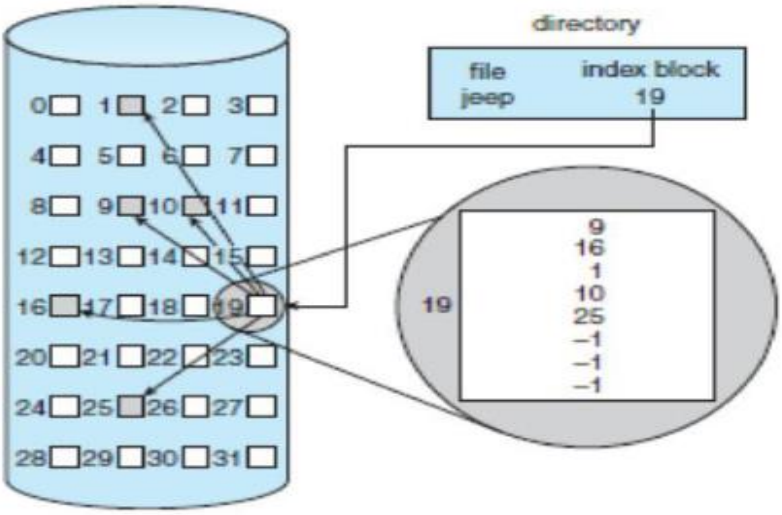


Fig 4.15 Indexed allocation

Advantages:

- Indexed allocation supports direct access.
- It does not suffer from External fragmentation. because any free block on the disk can satisfy a request for more Space.
- Indexed allocation does suffer from wasted space.

Disadvantages:

- The Pointer overhead of the index block is generally greater than the pointer overhead of linked allocation.
- Every file must have an index block, so we want the index block to be as small as possible.

Free Space Management

Free Space management keeps track of free disk space. Since disk space is limited, we need to reuse the space from deleted files for new files.

FREE SPACE LIST: To keep track of free disk space, the system maintains a **free-space list**. The free-space list records all free disk blocks those not allocated to some file or directory.

To create a file, we search the free-space list for the required amount of space and allocate that space to the new file. This space is then removed from the free-space list. When a file is deleted, its disk space is added to the free-space list.

The Free space list can be implemented in the following ways

- Bit Vector or Bit Map
- Linked list
- Groping
- Counting
- Space maps

Bit Vector:

The free-space list is implemented as a **bit map** or **bit vector**. Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0.

Example: Consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated.

Free-space bit map:

01111001111110001100000011100000...

Advantages:

- The main advantage of this approach is its relative simplicity and its efficiency in finding the first free block or n consecutive free blocks on the disk.
- Bit vectors are inefficient unless the entire vector is kept in main memory.

Disadvantages:

- If the disk size constantly increases, the problem with bit vectors will continue to increase.

Linked list:

Linked list implementation link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory. This first block contains a pointer to the next free disk block, and so on.

Example: Blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 in the disk were free and the rest of the blocks were allocated.

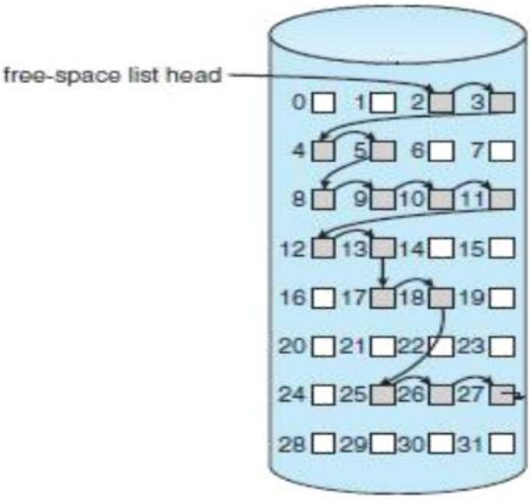


Fig 4.15 Linked List Method of Free Space List

We would keep a pointer to block 2 as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on.

Disadvantages:

- This scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/O time.

Grouping:

A modification of the free-list approach stores the addresses of n free blocks in the first free block. The first n−1 of these blocks are actually free. The last block contains the addresses of other n free blocks, and so on.

Advantages:

- The addresses of a large number of free blocks can now be found quickly.

Counting:

Free space list keep the address of the first free block and the number (n) of free contiguous blocks that follow the Each entry in the free-space list then consists of a disk address and a count.

This method of tracking free space is similar to the extent method of allocating blocks. These entries can be stored in a balanced tree, rather than a linked list, for efficient lookup, insertion, and deletion.

Space maps:

A space map uses log-structured file-system techniques to record the information about the free blocks. The space map is a log of all block activity (allocating and freeing), in time order, in counting format.

Oracle's **ZFS** file system creates **Meta slabs** to divide the space on the device into chunks of manageable size. A given volume may contain hundreds of Meta slabs. Each Meta slab has an associated space map. ZFS uses the counting algorithm to store information about free blocks.

Efficiency and Performance, Recovery

Efficiency

The efficient use of disk space depends heavily on the disk-allocation and directory algorithms in use.

For instance, UNIX inodes are pre allocated on a volume. However, by pre allocating the inodes and spreading them across the volume, we improve the file system's performance.

In Solaris operating system, process table and the open-file table data structures are allocated at system startup as fixed length. When the process table became full, no more processes could be created. When the file table became full, no more files could be opened. The system would fail to provide services to users. Table sizes could be increased only by recompiling the kernel and rebooting the system.

With later releases of Solaris, all kernel structures were allocated dynamically.

Performance

Even after the basic file-system algorithms have been selected, we can still improve performance in several ways. Most disk controllers include local memory to form an **on-board cache** that is large enough to store entire tracks at a time. Once a seek is performed, the disk controller then transfers any sector from the cache to the operating system.

Some systems maintain a separate section of main memory for a **buffer cache**, where blocks are kept under the assumption that they will be used again shortly. Other systems cache file data using a **page cache**. The page cache **uses virtual memory** techniques to cache file data as pages rather than as file-system-oriented blocks.

Caching file data using virtual addresses is far more efficient than caching through physical disk blocks, as accesses interface with virtual memory rather than the file system. Several systems including Solaris, Linux, and Windows use page caching to cache both process pages and file data. This is known as **unified virtual memory**.

Recovery

Files and directories are kept both in main memory and on disk, and care must be taken to ensure that a system failure does not result in loss of data or in data inconsistency. A system can recover from such a failure by using **Consistency Checking, Log-Structured File Systems, Backup and Restore.**

Consistency Checking: by Scanning all the metadata regularly on each file system can confirm or deny the consistency of the system.

Log-Structured File Systems: All metadata changes are written sequentially to a log. Each set of operations for performing a specific task is a transaction. Once the changes are written to this log, they are considered to be committed, and the system call can return to the user process, allowing it to continue execution.

As the changes are made, a pointer is updated to indicate which actions have completed and which are still incomplete. When an entire committed transaction is completed, it is removed from the log file, The resulting implementations are known as **log-based transaction-oriented file systems**.

Backup and Restore: System programs can be used to **back up** data from disk to another storage device, such as a magnetic tape or other hard disk. Recovery from the loss of an individual file, or of an entire disk, may then be a matter of **restoring** the data from backup.

I/O Systems

The role of the operating system in computer I/O is to manage and control I/O operations and I/O devices. This is met by a combination of hardware device controllers and software device driver techniques.

I/O Hardware

The hardware aspects of I/O are complex and is described by summarizing the essential devices and techniques.

- Bus
- Controller
- I/O port and Registers
- Communication to I/O Devices
- Handshaking
- Polling and Interrupts

Bus: is a collection of wires and a has defined protocol that specifies a set of messages that can be sent on the wires.

Controller: I/O controllers are a series of microchips which help in the communication of data between the central processing unit and the motherboard. Ex. of Controller are PCI : Peripheral Component Interconnect, SCSI: Small Computer Systems Interface, IDE: Integrated Drive Electronics

I/O Port and Registers: The device communicates with the machine via a connection point called a port. An I/O port typically consists of four registers, called the status, control, data-in, and data-out registers.

- The **data-in register** is read by the host to get input.
- The **data-out register** is written by the host to send output.
- The **status register** indicate states, such as whether the current command has completed, whether a byte is available to be read from the data-in register, and whether a device error has occurred.
- The **control register** can be written by the host to start a command or to change the mode of a device.

Communication to I/O Devices

The CPU must have a way to pass information to and from an I/O device. There are three approaches available to communicate with the CPU and Device.

- Special Instruction I/O
- Memory-mapped I/O
- Direct memory access (DMA)

Handshaking: It is a protocol for interaction between the host and a controller. The controller indicates its state through the busy bit in the status register. The controller sets the busy bit when it is busy working and clears the busy bit when it is ready to accept the next command. The host signals its wishes via the command-ready bit in the command register. The host sets the command-ready bit when a command is available for the controller to execute.

Polling and Interrupts:

A computer must have a way of detecting the arrival of any type of input. There are two ways that this can happen, known as polling and interrupts.

Polling is the simplest way for an I/O device to communicate with the processor. The process of **periodically checking status of the device** to see if it is time for the next I/O operation, is called polling. The I/O device simply puts the information in a Status register, and the processor must come and get the information.

This is an inefficient method and much of the processors time is wasted on unnecessary polls (checking device status).

Interrupts: An interrupt is a signal to the microprocessor from a device that requires attention. A device controller puts an interrupt signal on the bus when it needs CPU’s attention when CPU receives an interrupt, It saves its current state and invokes the appropriate interrupt handler using the interrupt vector (addresses of OS routines to handle various events). When the interrupting device has been dealt with, the CPU continues with its original task as if it had never been interrupted.

Application I/O Interface

Application I/O Interface represents the **structuring techniques and interfaces for the operating system to enable I/O devices to be treated in a standard, uniform way**.

The actual differences lies kernel level modules called device drivers which are custom tailored to corresponding devices but show one of the standard interfaces to applications. The purpose of the device-driver layer is to hide the differences among device controllers from the I/O subsystem of the kernel, such as the I/O system calls.

Following are the characteristics of I/O interfaces with respected to devices:

• **Character-stream / block**: A character-stream device transfers bytes in one by one fashion, whereas a block device transfers a complete unit of bytes.

• **Sequential / Random-access:** A sequential device transfers data in a fixed order determined by the device, random access device can be instructed to seek position to any of the available data storage locations.

• **Synchronous / Asynchronous:** A synchronous device performs data transfers with known response time where as an asynchronous device shows irregular or unpredictable response time.

• **Sharable / Dedicated:** A sharable device can be used concurrently by several processes or threads but a dedicated device cannot be used.

• **Speed of operation**: Device speeds may range from a few bytes per second to a few gigabytes per second.

• **Read-write, read only, or write only**: Some devices perform both input and output, but others support only one data direction that is read only.

Kernel I/O Subsystem

Kernel I/O Subsystem is responsible to provide many services related to I/O. Following are some of the services provided.

Scheduling: Kernel schedules a set of I/O requests to determine a good order in which to execute them. When an application issues a blocking I/O system call, the request is placed on the queue for that device. The Kernel I/O scheduler rearranges the order of the queue to improve the overall system efficiency and the average response time experienced by the applications.

Buffering: Kernel I/O Subsystem maintains a memory area known as buffer that stores data while they are transferred between two devices or between a device with an application operation. Buffering is done to cope with a speed mismatch between the producer and consumer of a data stream or to adapt between devices that have different data transfer sizes.

Caching: Kernel maintains cache memory which is region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original.

Spooling and Device Reservation: A spool is a buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams. The spooling system copies the queued spool files to the printer one at a time. In some operating systems, spooling is managed by a system daemon process. In other operating systems, it is handled by an in kernel thread.

Error Handling: An operating system that uses protected memory can guard against many kinds of hardware and application errors.

I/O Protection : Errors and the issue of protection are closely related. A user process may attempt to issue illegal I/O instructions to disrupt the normal function of a system. We can use the various mechanisms to ensure that such disruption cannot take place in the system.

To prevent illegal I/O access, we define all I/O instructions to be privileged instructions. The user cannot issue I/O instruction directly.

Streams

A **stream** is a **full-duplex connection** between a device driver and a user-level process. It consists of a stream head that interfaces with the user process, a driver end that controls the device, and zero or more stream modules between the stream head and the driver end.

- Each of these components contains a pair of queues, a read queue and a write queue.
- Message passing is used to transfer data between queues.
- A process can open a serial-port device via a stream and messages are exchanged between queues in adjacent modules, a queue in one module may overflow an adjacent queue. To prevent this, a queue may support flow control. A queue that also supports flow buffer space.
- Without flow control, a queue accepts all messages and immediately sends them on to the queue in the adjacent module without buffering them.
- A user process writes data to a device using either the write() or putmsg() system call.
- The write() system call writes raw data to the stream, whereas putmsg() allows the user process to specify a message.
- Regardless of the system call used by the user process, the stream head copies the data into a message and delivers it to the queue for the next module in line. This copying of messages continues until the message is copied to the driver.
- Similarly, the user process reads data from the stream head using either the read() or getmsg() system call. If read() is used, the stream head gets a message from its adjacent queue and returns ordinary data (an unstructured byte stream) to the process. If getmsg() is used, a message is returned to the process.

- Streams I/O is asynchronous (or nonblocking) except when the user process communicates with the stream head. When writing to the stream, the user process will block, assuming the next queue uses flow control, until there is room to copy the message. Likewise, the user process will block when reading from the stream until data are available.
- Most UNIX variants support Streams, and it is the preferred method for writing protocols and device drivers. For example, UNIX and Solaris implement the socket mechanism using STREAMS.

The STREAMS structure is shown in Figure.

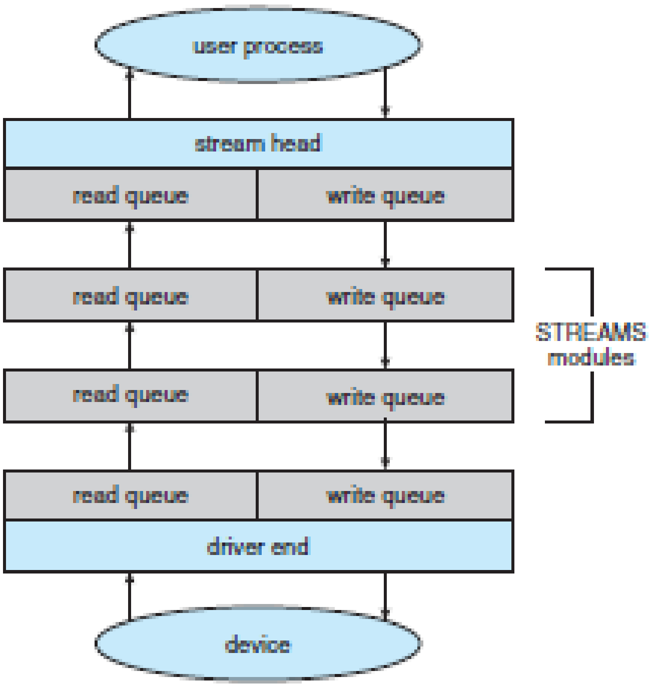


Fig 4.16 Linked List Method of Free Space List

Performance

I/O is a major factor in system performance. It places heavy demands on the CPU to execute device-driver code and to schedule processes fairly and efficiently as they block and unblock. The resulting context switches stress the CPU and its hardware caches. I/O also exposes any inefficiencies in the interrupt-handling mechanisms in the kernel.

In addition, I/O loads down the memory bus during data copy between controllers and physical memory and again during copies between kernel buffers and application data space. Although modern computers can handle many thousands of interrupts per second, interrupt handling is a relatively expensive task.

Each interrupt causes the system to perform a state change, to execute the interrupt handler, and then to restore state. Programmed I/O can be more efficient than interrupt-driven I/O, if the number of cycles spent in busy waiting is not excessive. An I/O completion typically unblocks a process, leading to the full overhead of a context switch.

We can employ **several principles to improve the efficiency** of I/O:

- Reduce the number of context switches.
- Reduce the number of times that data must be copied in memory while passing between device and application.
- Reduce the frequency of interrupts by using large transfers, smart controllers, and polling (if busy waiting can be minimized).
- Increase concurrency by using DMA-knowledgeable controllers or channels to offload simple data copying from the CPU.
- Move processing primitives into hardware, to allow their operation in device controllers to be concurrent with CPU and bus operation.
- Balance CPU, memory subsystem, bus, and I/O performance, because an overload in any one area will cause idleness in others.