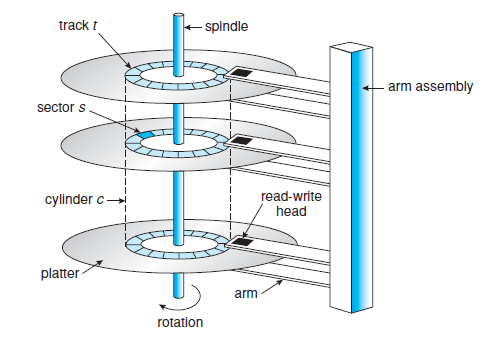
**UNIT 4**

**I/O SYSTEMS**

**Overview of Mass-Storage Structure:**

**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**. The set of tracks that are at one arm position makes up a **cylinder**. There may be thousands of concentric cylinders in a disk drive, and each track may contain hundreds of sectors.

Disk speed has two parts. The **transfer rate** is the rate at which data flow between the drive and the computer. The **positioning time**, or **random-access time**, consists of two parts: the time necessary to move the disk arm to the desired cylinder, called the **seek time**, and the time necessary for the desired sector to rotate to the disk head, called the **rotational latency**.

The disk head flies on an extremely thin cushion of air (measured in microns), there is a danger that the head will make contact with the disk surface. Although the disk platters are coated with a thin protective layer, the head will sometimes damage the magnetic surface. This accident is called a **head crash.**

A disk drive is attached to a computer by a set of wires called an I/O bus. Several kinds of buses are available, including advanced technology attachment (ATA), serial ATA (SATA), eSATA, universal serial bus (USB), and fibre channel (FC). The data transfers on a bus are carried out by special electronic processors called **controllers**. The **host controller** is the controller at the computer end of the bus. A **disk controller** is built into each disk drive.

**Solid-State Disks:**

SSD is nonvolatile memory that is used like a hard drive. There are many variations of this technology, from DRAM with a battery to allow it to maintain its state in a power failure through flash-memory technologies like single-level cell (SLC) and multilevel cell (MLC) chips. SSDs have the same characteristics as traditional hard disks but can be more reliable because they have no moving parts and faster because they have no seek time or latency. In addition, they consume less power. However, they are

more expensive per megabyte than traditional hard disks, have less capacity than the larger hard disks, and may have shorter life spans than hard disks, so their uses are somewhat limited.

**Magnetic Tapes:**

**Magnetic tape** was used as an early secondary-storage medium. Although 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 Scheduling:**

Whenever a process needs I/O to or from the disk, it issues a system call to the operating system. 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

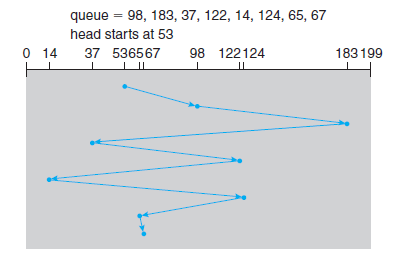
For the fast access time and large disk bandwidth any one of several disk-scheduling algorithms can be used

**FCFS Scheduling:**

The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm does not provide the fastest service. Consider, for example, a disk queue with requests for I/O to blocks on cylinders

98, 183, 37, 122, 14, 124, 65, 67,

in that order. If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67, for a total head movement of 640 cylinders.

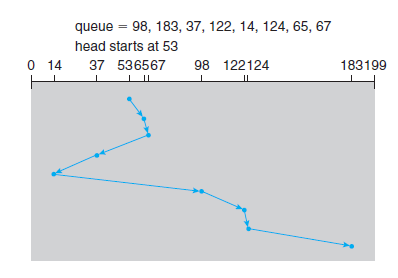


**FCFS disk scheduling.**

**SSTF Scheduling:**

It seems reasonable to service all the requests close to the current head position before moving the head far away to service other requests. This assumption is the basis for the **shortest-seek-time-first (SSTF) algorithm**. The SSTF algorithm selects the request with the least seek time from the current head position. For our example request queue, the closest request to the initial head position (53) is at cylinder 65. Once we are at cylinder 65, the next closest request is at cylinder 67. From there, the request at cylinder 37 is closer than the one at 98, so 37 is served next. Continuing, we service the request at cylinder 14, then 98, 122, 124, and finally 183. This scheduling method results in a total head movement of only 236 cylinders—little more than one-third of the distance needed for FCFS scheduling of this request queue.

SSTF may cause starvation of some requests. Suppose that we have two requests in the queue, for cylinders 14 and 186, and while the request from 14 is being serviced, a new request near 14 arrives.

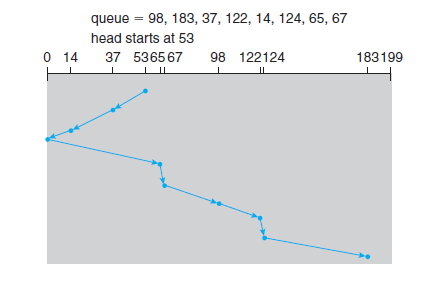


**SSTF disk scheduling**

This new request will be serviced next, making the request at 186 wait. While this request is being serviced, another request close to 14 could arrive. In theory, a continual stream of requests near one another could cause the request for cylinder 186 to wait indefinitely.

**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 movements reversed, and servicing continues. The head continuously scans back and forth across the disk. The SCAN algorithm is sometimes called the **elevator algorithm.**

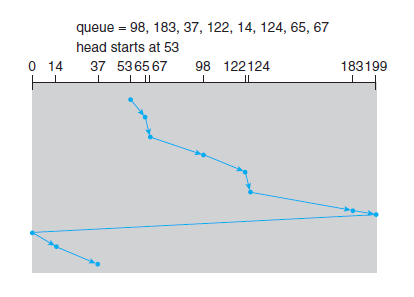
****

**SCAN disk scheduling.**

Assuming that the disk arm is moving toward 0 and that the initial head position is again 53, the head will next service 37 and then 14. At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124, and 183. If a request arrives in the queue just in front of the head, it will be serviced almost immediately; a request arriving just behind the head will have to wait until the arm moves to the end of the disk, reverses direction, and comes back.

**C-SCAN Scheduling:**

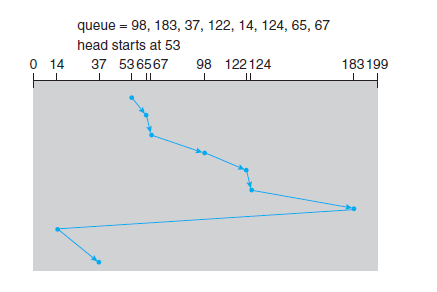
**Circular SCAN (C-SCAN) scheduling** is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip.



**C-SCAN disk scheduling**

**LOOK Scheduling:**

The disk arm goes only as far as the final request in each direction. Then, it reverses direction immediately, without going all the way to the end of the disk. Versions of SCAN and C-SCAN that follow this pattern are called **LOOK** and **C-LOOK scheduling**, because they **look** for a request before continuing to move in a given direction.

****

**C-LOOK disk scheduling**

**Disk Management:**

**Disk Formatting:**

A new magnetic disk is a blank slate and it is just a platter of a magnetic recording material. 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**. Low-level formatting fills the disk with a special data structure for each sector. The data structure for a sector typically consists of a header, a data area and a trailer. The header and trailer contain information used by the disk controller, such as a sector number and an **error-correcting code (ECC)**. When the controller writes a sector of data during normal I/O, the ECC is updated with a value calculated from all the bytes in the data area. 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.

Before using a disk to hold files, the operating system needs to record its own data structures on the disk. It does so in two steps. The first step is to **partition** the disk into one or more groups of cylinders. The operating system can treat each partition as though it were a separate disk. For instance, one partition can hold a copy of the operating system’s executable code, while another holds user files. The second step is **logical formatting**, or creation of a file system.

**Boot Block:**

The initial **bootstrap** program tends to be simple. It initializes all aspects of the system, from CPU registers to device controllers and the contents of main memory, and then starts the operating system. For most computers, the bootstrap is stored in **read-only memory (ROM)**. This location is convenient, because ROM needs no initialization and is at a fixed location that the processor can start executing when powered up or reset. And, since ROM is read only, it cannot be infected by a computer virus. The problem is that changing this bootstrap code requires changing the ROM hardware chips. For this reason, most systems store a tiny bootstrap loader program in the boot ROM whose only job is to bring in a full bootstrap program 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 Windows system places its boot code in the first sector on the hard disk, which it terms the **master boot record**, or **MBR**.

**Bad Blocks:**

The disks have moving parts and small tolerances, they are prone to failure. More frequently, one or more sectors become defective. Most disks even come from the factory with **bad blocks**. On simple disks, such as some disks with IDE controllers, bad blocks are handled manually. 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. If blocks go bad during normal operation, a special program must be run manually to search for the bad blocks and to lock them away.

Low-level formatting has a set of 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**. A typical bad-sector transaction might be as follows:

• The operating system tries to read logical block 87.

• The controller calculates the ECC and finds that the sector is bad. It reports this finding to the operating system.

• The next time the system is rebooted, a special command is run to tell the controller to replace the bad sector with a spare.

• After that, whenever the system requests logical block 87, the request is translated into the replacement sector’s address by the controller.

**File Concept:**

A file is a named collection of related information that is recorded on secondary storage. From a user’s perspective, a file is the smallest allotment of logical secondary storage; that is, data cannot be written to secondary storage unless they are within a file.

A file has a certain defined structure, which depends on its type. 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 Attributes:**

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

• **Name:** The symbolic file name is the only information kept in human readable form.

• **Identifier**: This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.

• **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) and possibly the maximum allowed size are included in this attribute.

• **Protection**: Access-control information determines who can do reading, writing, executing, and so on.

• **Time, date, and user identification**: This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

**File Operations:**

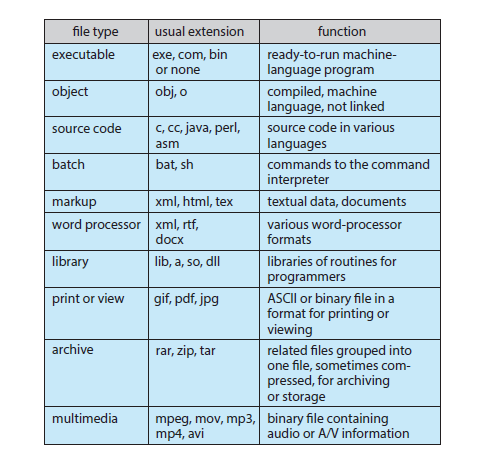
A file is an abstract data type. The operating system can provide system calls to create, write, read, reposition, delete, and truncate files.

* **Creating a file**: Two steps are necessary to create a file. First, space in the file system must be found for the file. Second, an entry for the new file must be made in the directory.
* **Writing a file**: To write a file, we make a system call specifying both the name of the file and the information to be written to the file. The system must keep a **write pointer** to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.
* **Reading a file**: To read from a file, we use a system call that specifies the name of the file and where the next block of the file should be put. Again, the directory is searched for the associated entry, and the system needs to keep a **read pointer** to the location in the file where the next read is to take place.
* **Repositioning within a file**: The directory is searched for the appropriate entry, and the current file position pointer is repositioned to a given value.
* **Deleting a file**: To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase the directory entry.
* **Truncating a file:** The user may want to erase the contents of a file but keep its attributes.

Information associated with an open file:

* **File pointer**
* **File-open count:** The file-open count tracks the number of opens and closes of files and reaches zero when last file closes.
* **Disk location of the file**
* **Access rights**

**File Types:**

****

**File Structure:**

File types also can be used to indicate the internal structure of the file. The operating system requires that an executable file have a specific structure so that it can determine where in memory to load the file and what the location of the first instruction is. Some operating systems extend this idea into a set of system-supported file structures, with sets of special operations for manipulating files with those structures. If the operating system defines five different file structures, it needs to contain the code to support these file structures.

**Internal File Structure:**

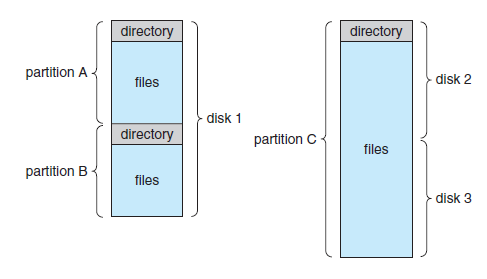
Disk systems typically have a well-defined block size determined by the size of a sector. All disk I/O is performed in units of one block (physical record), and all blocks are the same size. It is unlikely that the physical record size will exactly match the length of the desired logical record. Logical records may even vary in length. Packing a number of logical records into physical blocks is a common solution to this problem.

The logical record size, physical block size, and packing technique determine how many logical records are in each physical block. The packing can be done either by the user’s application program or by the operating system. In either case, the file may be considered a sequence of blocks.

**Directory and Disk Structure:**

A disk can be partitioned into quarters, and each quarter can hold a separate file system. Storage devices an also be collected together into RAID sets that provide protection from the failure of a single disk. Partitioning is useful for limiting the sizes of individual file systems, putting multiple file-system types on the same device, or leaving part of the device available for other uses.

Any entity containing a file system is generally known as a **volume**. Each volume that contains a file system must also contain information about the files in the system. This information is kept in entries in a **device directory** or **volume table of contents**. The device directory records information—such as name, location, size, and type—for all files on that volume.



**Storage Structure:**

A general-purpose computer system has multiple storage devices, and those devices can be sliced up into volumes that hold file systems.

Types of file systems in the Solaris system:

• **tmpfs** - a “temporary” file system that is created in volatile main memory and has its contents erased if

the system reboots or crashes

• **objfs** - a “virtual” file system (essentially an interface to the kernel that looks like a file system) that

gives debuggers access to kernel symbols

• **ctfs** - a virtual file system that maintains “contract” information to manage which processes start when

the system boots and must continue to run during operation

• **lofs** - a “loop back” file system that allows one file system to be accessed in place of another one

• **procfs** - a virtual file system that presents information on all processes as a file system

• **ufs, zfs** - general-purpose file systems

**Directory Overview:**

The directory can be viewed as a symbol table that translates file names into their directory entries. The directory itself can be organized in many ways. The organization must allow us to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory.

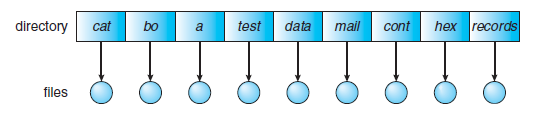
The operations that are to be performed on a directory:

**Search for a file**: We need to be able to search a directory structure to find the entry for a particular file.

* **Create a file**: New files need to be created and added to the directory.
* **Delete a file**: When a file is no longer needed, we want to be able to remove it from the directory.
* **List a directory**: We need to be able to list the files in a directory.
* **Rename a file**: Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes.
* **Traverse the file system**.

**Single-Level Directory:**

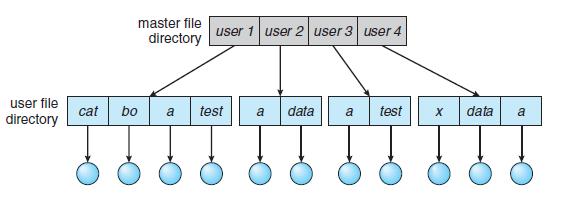
The simplest directory structure is the single-level directory. All files are contained in the same directory. A single-level directory has significant limitations, however, when the number of files increases or when the system has more than one user. Since all files are in the same directory, they must have unique names. If two users call their data file test.txt, then the unique-name rule is violated.



**Single-level directory**

**Two-Level Directory:**

In the two-level directory structure, each user has his own **user file directory (UFD)**. The UFDs have similar structures, but each lists only the files of a single user. When a user job starts or a user logs in, the system’s **master file directory (MFD)** is searched. The MFD is indexed by user name or account number, and each entry points to the UFD for that user.

****

**Two-level directory structure.**

When a user refers to a particular file, only his own UFD is searched. Thus, different users may have files with the same name, as long as all the file names within each UFD are unique. To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user’s file that has the same name.

Although the two-level directory structure solves the name-collision problem, it still has disadvantages. This structure effectively isolates one user from another. Isolation is an advantage when the users are completely independent but is a disadvantage when the users want to cooperate on some task and to access one another’s files.

If user A wishes to access her own test file named test.txt, she can simply refer to test.txt. To access the file named test.txt of user B (with directory-entry name userb), however, she might have to refer to /userb/test.txt.

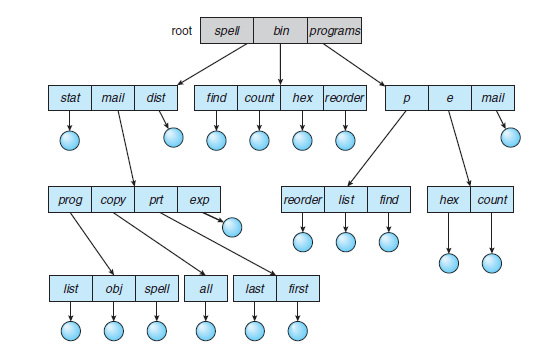
**Tree-Structured Directories:**

This generalization allows users to create their own subdirectories and to organize their files accordingly. A tree is the most common directory structure. The tree has a root directory, and every file in the

system has a unique path name.

Each process has a current directory. The **current directory** should contain most of the files that are of current interest to the process. When reference is made to a file, the current directory is searched. If a file

is needed that is not in the current directory, then the user usually must either specify a path name or change the current directory to be the directory holding that file.



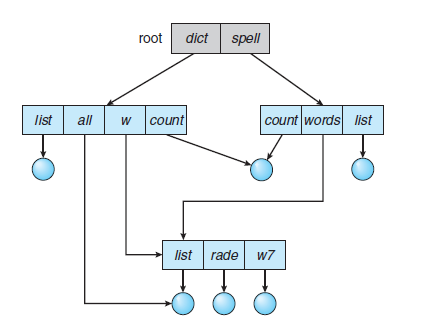
**Tree-structured directory structure**

Path names can be of two types: absolute and relative. An **absolute path name** begins at the root and follows a path down to the specified file, giving the directory names on the path. A **relative path name** defines a path from the current directory. For example, in the tree-structured file system of Figure, if the current directory is root/spell/mail, then the relative path name prt/first refers to the same file as does the absolute path name root/spell/mail/prt/first.

An interesting policy decision in a tree-structured directory concerns how to handle the deletion of a directory. If a directory is empty, its entry in the directory that contains it can simply be deleted. However, suppose the directory to be deleted is not empty but contains several files or subdirectories. One of two approaches can be taken. Some systems will not delete a directory unless it is empty. Thus, to delete a directory, the user must first delete all the files in that directory. If any subdirectories exist, this procedure must be applied recursively to them, so that they can be deleted also. An alternative approach, such as that taken by the UNIX rm command, is to provide an option: when a request is made to delete a directory, all that directory’s files and subdirectories are also to be deleted.

**Acyclic-Graph Directories:**

An **acyclic graph -** that is, a graph with no cycles - allows directories to share subdirectories and files. The same file or subdirectory may be in two different directories. The acyclic graph is a natural generalization of the tree-structured directory scheme. With a shared file, only one actual file exists, so any changes made by one person are immediately visible to the other. Sharing is particularly important for subdirectories; a new file created by one person will automatically appear in all the shared subdirectories. When people are working as a team, all the files they want to share can be put into one directory. The UFD of each team member will contain this directory of shared files as a subdirectory.



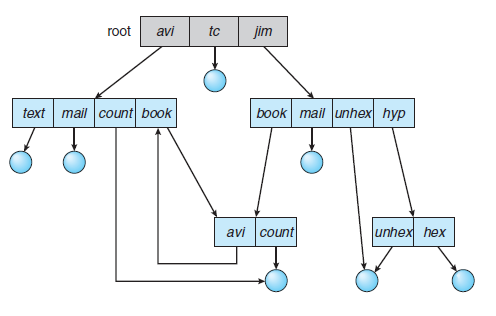
Shared files and subdirectories can be implemented in several ways. Many of the UNIX systems, creates a new directory entry called a link. A **link** is effectively a pointer to another file or subdirectory. For example, a link may be implemented as an absolute or a relative path name. When a reference to a file is made, we search the directory. If the directory entry is marked as a link, then the name of the real file is included in the link information.

Another common approach to implementing shared files is simply to duplicate all information about them in both sharing directories. Thus, both entries are identical and equal.

An acyclic-graph directory structure is more flexible than a simple tree structure, but it is also more complex. Several problems must be considered carefully. A file may now have multiple absolute path names.

**General Graph Directory:**

If cycles are allowed to exist in the directory, we likewise want to avoid searching any component twice, for reasons of correctness as well as performance. A poorly designed algorithm might result in an infinite loop continually searching through the cycle and never terminating.



**General graph directory**

**File Sharing:**

**Multiple Users:**

When an operating system accommodates multiple users, the issues of file sharing, file naming, and file protection become preeminent. The operating system 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.

To implement sharing and protection most systems have evolved to use the concepts of file (or directory) **owner** (or **user**) and **group**. 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 (or directory) 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. Likewise, the group IDs can be compared. The result indicates

which permissions are applicable.

**Remote File Systems:**

Networking allows the sharing of resources spread across a campus or even around the world. One obvious resource to share is data in the form of files. Through the evolution of network and file technology, remote file-sharing methods have changed. The first implemented method involves manually transferring files between machines via programs like ftp. The second major method uses a **distributed file system (DFS)** in which remote directories are visible from a local machine.

**The Client–Server Model:**

The machine containing the files is the **server**, and the machine seeking access to the files is the **client**. The client–server relationship is common with networked machines. Generally, the server declares that a resource is available to clients and specifies exactly which resource and exactly which clients. A server can serve multiple clients, and a client can use multiple servers, depending on the implementation details of a given client–server facility.

Client identification is more difficult. A client can be specified by a network name or other identifier, such as an IP address, but these can be **spoofed**, or imitated. As a result of spoofing, an unauthorized client could be allowed access to the server. More secure solutions include secure authentication of the client via encrypted keys. In the case of UNIX and its network file system (NFS), authentication takes place via the client networking information, by default. In this scheme, 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. Other distributed information systems provide ***user name/password/user ID/group ID*** space for a distributed facility. UNIX systems have employed a wide variety of distributed information methods. Sun Microsystems introduced **yellow pages.**

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

The industry is moving toward use of the **lightweight directory-access protocol (LDAP)** as a secure distributed naming mechanism. In fact, active directory is based on LDAP. Oracle Solaris and most other major operating systems include LDAP and allow it to be employed for user authentication as well as system-wide retrieval of information

**Failure Modes:**

Local file systems can fail for a variety of reasons, including failure of the disk containing the file system, corruption of the directory structure or other disk-management information (collectively called **metadata**), disk-controller failure, cable failure, and host-adapter failure. User or system-administrator failure can also cause files to be lost or entire directories or volumes to be deleted.

Remote file systems have even more failure modes. Because of the complexity of network systems and the required interactions between remote machines, many more problems can interfere with the proper operation of remote file systems. In the case of networks, the network can be interrupted between two hosts. Such interruptions can result from hardware failure, poor hardware configuration, or networking implementation issues.

Consider a client in the midst of using a remote file system. It has files open from the remote host. Now consider a partitioning of the network, a crash of the server, or even a scheduled shutdown of the server. Suddenly, the remote file system is no longer reachable. The system can either terminate all operations to the lost server or delay operations until the server is again reachable. Termination of all operations can result in users’ losing data.

To implement this kind of 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 seamlessly recover from a failure.

**Consistency Semantics:**

Consistency semantics specify how multiple users of a system are to access a shared file simultaneously. In particular, they specify when modifications of data by one user will be observable by other users. These semantics are typically implemented as code with the file system.

**UNIX Semantics:**

The UNIX file system uses the following consistency 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. Here, a file has a single image that interleaves all accesses, regardless of their origin.

**Session Semantics**

The Andrew file system (OpenAFS) uses the following consistency 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-Files Semantics:**

A unique approach is that of **immutable shared files**. 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.

**Protection:**

When information is stored in a computer system, we want to keep it safe from physical damage (the issue of reliability) and improper access (the issue of protection).

**Types of Access:**

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. Several different types of operations may be controlled:

• **Read** - Read from the file.

• **Write** - Write or rewrite the file.

• **Execute** - Load the file into memory and execute it.

• **Append** - Write new information at the end of the file.

• **Delete** - Delete the file and free its space for possible reuse.

• **List** - List the name and attributes of the file.

**Access Control:**

The most common approach to the protection problem is to make access dependent on the identity of the user. Different users may need different types of access to a file or directory. The most general scheme to implement identity dependent access is to associate with each file and directory an **access-control list (ACL)** specifying user names and the types of access allowed for each user. When a user requests access to a particular file, the operating system checks the access list associated with that file. If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.

The main problem with access lists is their length. If we want to allow everyone to read a file, we must list all users with read access. This technique has two undesirable consequences:

• Constructing such a list may be a tedious and unrewarding task, especially if we do not know in

advance the list of users in the system.

• The directory entry, previously of fixed size, now must be of variable size, resulting in more

complicated space management.

To condense the length of the access-control list, many systems recognize three classifications of users in connection with each file:

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

• **Group**: A set of users who are sharing the file and need similar access is a group, or work group.

• **Universe**: All other users in the system constitute the universe.

To illustrate, consider a person, Sara, who is writing a new book. She has hired three graduate students (Jim, Dawn, and Jill) to help with the project. The text of the book is kept in a file named book.tex. The protection associated with this file is as follows:

• Sara should be able to invoke all operations on the file.

• Jim, Dawn, and Jill should be able only to read and write the file; they should not be

allowed to delete the file.

• All other users should be able to read, but not write, the file.

With the more limited protection classification, only three fields are needed to define protection. Often, each field is a collection of bits, and each bit either allows or prevents the access associated with it. For example, the UNIX system defines three fields of 3 bits each—rwx, where r controls read access, w controls write access, and x controls execution. Thus, for our example, the protection fields for the file book.tex are as follows: for the owner Sara, all bits are set; for the group text, the r and w bits are set; and for the universe, only the r bit is set.

**Other Protection Approaches:**

Another approach to the protection problem is to associate a password with each file. If the passwords are chosen randomly and changed often, this scheme may be effective in limiting access to a file. The use of passwords has a few disadvantages, however. First, the number of passwords that a user needs to remember may become large, making the scheme impractical. Second, 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. Some systems allow a user to associate a password with a subdirectory, rather than with an individual file, to address this problem.

In a multilevel directory structure, we need to protect not only individual files but also collections of files in subdirectories; that is, we need to provide a mechanism for directory protection.

**File-System Structure:**

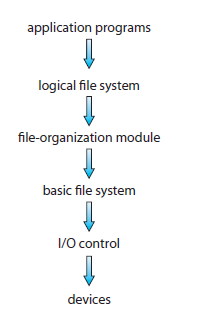
Disks provide most of the secondary storage on which file systems are maintained. Two characteristics make them convenient for this purpose:

* A disk can be rewritten in place; it is possible to read a block from the disk, modify the block, and write it back into the same place.
* Disk can access directly any block of information it contains. Thus, it is simple to access any file either sequentially or randomly

To improve I/O efficiency, I/O transfers between memory and disk are performed in units of **blocks**. Each block has one or more sectors.

**File systems** provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily. 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 for organizing 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.

The **I/O control** level consists of device drivers and interrupt handlers to transfer information between the main memory and the disk system. A device driver can be thought of as a translator. Its input consists of high level commands such as “retrieve block 123.” Its output consists of low-level, hardware-specific instructions that are used by the hardware controller, which interfaces the I/O device to the rest of the system.



**Layered 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.

The **file-organization module** knows about files and their logical blocks, as well as physical blocks. By knowing the type of file allocation used and the location of the file, the file-organization module can translate logical block addresses to physical block addresses for the basic file system to transfer.

Finally, the **logical file system** manages metadata information. Metadata includes all of the file-system structure except the actual data. It maintains file structure via file-control blocks. A **file control block (FCB)** contains information about the file, including ownership, permissions, and location of the file contents.

When a layered structure is used for file-system implementation, duplication of code is minimized. The I/O control and sometimes the basic file-system code can be used by multiple file systems. Each file system can then have its own logical file-system and file-organization modules.

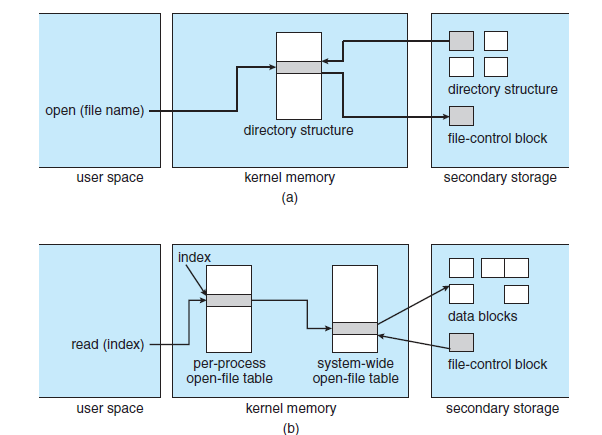
**File-System Implementation:**

Several on-disk and in-memory structures are used to implement a file system. On disk, the file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files.

* A **boot control block** (per volume) can contain information needed by the system to boot an operating system from that volume.
* A **volume control block** (per volume) contains volume (or partition) details, such as the number of blocks in the partition, the size of the blocks, a free-block count and free-block pointers, and a free-FCB count and FCB pointers.
* A directory structure (per file system) is used to organize the files.
* A per-file FCB contains many details about the file. It has a unique identifier number to allow
* association with a directory entry.

The in-memory information is used for both file-system management and performance improvement via caching. The data are loaded at mount time, updated during file-system operations, and discarded at dismount.

* An in-memory **mount table** contains information about each mounted volume.
* An in-memory directory-structure cache holds the directory information of recently accessed directories.
* The **system-wide open-file table** contains a copy of the FCB of each open file, as well as other information.
* The **per-process open-file table** contains a pointer to the appropriate entry in the system-wide open-file table, as well as other information.
* Buffers hold file-system blocks when they are being read from disk or written to disk.

****

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

**Partitions and Mounting:**

A disk can be sliced into multiple partitions, or a volume can span multiple partitions on multiple disks. Each partition can be either “raw,” containing no file system, or “cooked,” containing a file system. **Raw disk** is used where no file system is appropriate. UNIX swap space can use a raw partition, for example, since it uses its own format on disk and does not use a file system. Likewise, some databases use raw disk and format the data to suit their needs.

Boot information can be stored in a separate partition. Many systems can be **dual-booted**, allowing us to install multiple operating systems on a single system.

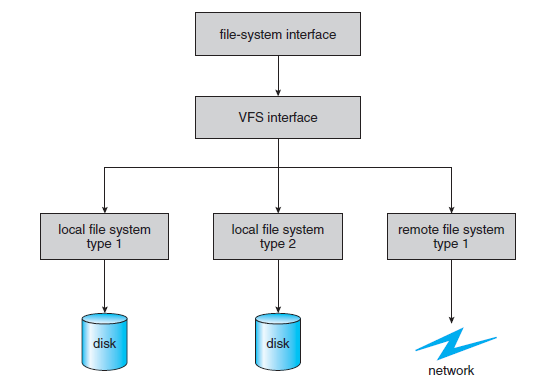
The **root partition**, which contains the operating-system kernel and sometimes other system files, is mounted at boot time. Other volumes can be automatically mounted at boot or manually mounted later, depending on the operating system.

Microsoft Windows–based systems mount each volume in a separate name space, denoted by a letter and a colon. To record that a file system is mounted at F:, for example, the operating system places a pointer to the file system in a field of the device structure corresponding to F:. When a process specifies the driver letter, the operating system finds the appropriate file-system pointer and traverses the directory structures on that device to find the specified file.

**Virtual File Systems:**

**The virtual file system (VFS)** layer serves two important functions:

* It separates file-system-generic operations from their implementation by defining a clean VFS interface.
* It provides a mechanism for uniquely representing a file throughout a network. The VFS is based on a file-representation structure, called a **vnode**, that contains a numerical designator for a network-wide unique file. This network-wide uniqueness is required for support of network file systems.

****

**Schematic view of a virtual file system**

The VFS architecture in Linux. The four main object types defined by the Linux VFS are:

• The **inode object**, which represents an individual file

• The **file object**, which represents an open file

• The **superblock object**, which represents an entire file system

• The **dentry object**, which represents an individual directory entry

For each of these four object types, the VFS defines a set of operations that may be implemented.

For example, some of the operations for the file object include:

• int open(. . .) - Open a file.

• int close(...) - Close an already-open file.

• ssize t read(. . .) - Read from a file.

• ssize t write(. . .) - Write to a file.

• int mmap(. . .) - Memory-map a file.

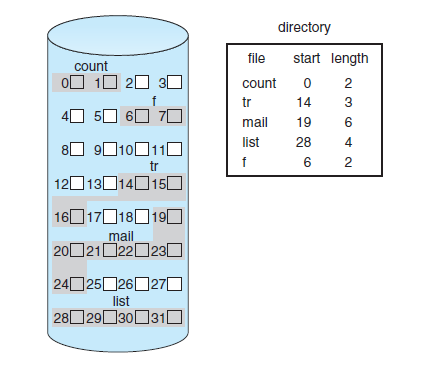
**Allocation Methods:**

Three major methods of allocating disk space are in wide use: contiguous, linked, and indexed. Each method has advantages and disadvantages. Although some systems support all three, it is more common for a system to use one method for all files within a file-system type.

**Contiguous Allocation:**

**Contiguous allocation** requires that each file occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk. Contiguous allocation of a file is defined by the disk address and length (in block units) of the first block. If the file is *n* blocks long and starts at location *b,* then it occupies blocks *b, b* + 1, *b* + 2, ..., *b* + *n* − 1. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.

Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk address of the last block referenced and, when necessary, reads the next block. For direct access to block *i* of a file that starts at block *b,* we can immediately access block *b* + *i.* Thus, both sequential and direct access can be supported by contiguous allocation.



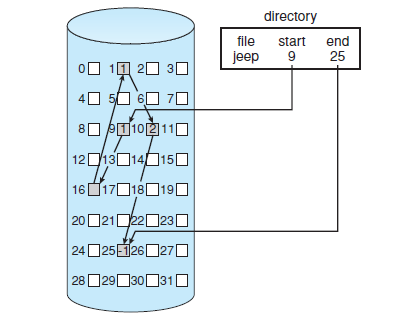
Contiguous allocation has some problems, however. One difficulty is finding space for a new file. As files are allocated and deleted, the free disk space is broken into little pieces. External fragmentation exists whenever free space is broken into chunks. It becomes a problem when the largest contiguous chunk is insufficient for a request; storage is fragmented into a number of holes, none of which is large enough to store the data.

One strategy for preventing loss of significant amounts of disk space to external fragmentation is to copy an entire file system onto another disk. The original disk is then freed completely, creating one large contiguous free space. We then copy the files back onto the original disk by allocating contiguous space from this one large hole. This scheme effectively **compacts** all free space into one contiguous space, solving the fragmentation problem. Compacting these disks may take hours and may be necessary on a weekly basis.

Another problem with contiguous allocation is determining how much space is needed for a file. When the file is created, the total amount of space it will need must be found and allocated. If we allocate too little space to a file, we may find that the file cannot be extended. Hence, we cannot make the file larger in place. Two possibilities then exist. First, the user program can be terminated, with an appropriate error message. The user must then allocate more space and run the program again. These repeated runs may be costly.

**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. The directory contains a pointer to the first and last blocks of the file.



For example, a file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10, and finally block 25(Figure). Each block contains a pointer to the next block. These pointers are not made available to the user.

To create a new file, we simply create a new entry in the directory. With inked allocation, each directory entry has a pointer to the first disk block of the file. A write to the file causes the free-space management system to find a free block, and this new block is written to and is linked to the end of the file. To read a file, we simply read blocks by following the pointers from block to block.

Linked allocation does have disadvantages, however. The major problem is that it can be used effectively only for sequential-access files. To find the *i*th block of a file, we must start at the beginning of that file and follow the pointers until we get to the *i*th block. Each access to a pointer requires a disk read, and some require a disk seek.

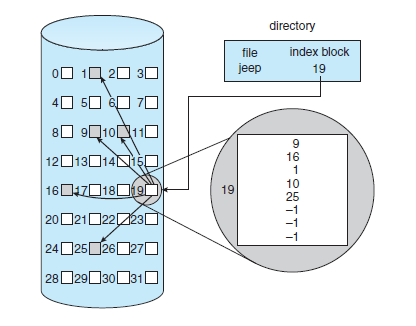
Another problem of linked allocation is reliability. Recall that the files are linked together by pointers scattered all over the disk, and consider what would happen if a pointer were lost or damaged. A bug in the operating-system software or a disk hardware failure might result in picking up the wrong pointer. This error could in turn result in linking into the free-space list or into another file.

An important variation on linked allocation is the use of a **file-allocation table (FAT)**. This simple but efficient method of disk-space allocation was used by the MS-DOS operating system. A section of disk at the beginning of each volume is set aside to contain the table. The table has one entry for each disk block and is indexed by block number. The FAT is used in much the same way as a linked list. The directory entry contains the block number of the first block of the file. The table entry indexed by that block number contains the block number of the next block in the file. This chain continues until it reaches the last block, which has a special end-of-file value as the table entry.

**Indexed Allocation:**

Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation. However, in the absence of a FAT, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered. **Indexed allocation** solves this problem by bringing all the pointers together into one location: the **index block**.

Each file has its own index block, which is an array of disk-block addresses. The *ith* entry in the index block points to the *ith* block of the file. The directory contains the address of the index block.



When the file is created, all pointers in the index block are set to null. When the i th block is first written, a block is obtained from the free-space manager, and its address is put in the i th index-block entry.

**How large the index block should be**. Every file must have an index block, so we want the index block to be as small as possible. If the index block is too small, however, it will not be able to hold enough pointers for a large file. Mechanisms for this purpose include the following:

* **Linked scheme**: An index block is normally one disk block. Thus, it can be read and written directly by itself. To allow for large files, we can link together several index blocks.
* **Multilevel index**: Avariant of linked representation uses a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks. To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block. This approach could be continued to a third or fourth level, depending on the desired maximum file size.
* **Combined scheme**: Another alternative, used in UNIX-based file systems, is to keep the first, say, 15 pointers of the index block in the file’s inode. The first 12 of these pointers point to **direct blocks**; that is, they contain addresses of blocks that contain data of the file. Thus, the data for small files (of no more than 12 blocks) do not need a separate index block. The next three pointers point to **indirect blocks**. The first points to a **single indirect block**, which is an index block containing not data but the addresses of blocks that do contain data. The second points to a **double indirect block**, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. The last pointer contains the address of a **triple indirect block**.

**Free-Space Management:**

Since disk space is limited, we need to reuse the space from deleted files for new files, if possible. 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.

**Bit Vector:**

Frequently, 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. For 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. The free-space bit map would be

001111001111110001100000011100000 ...

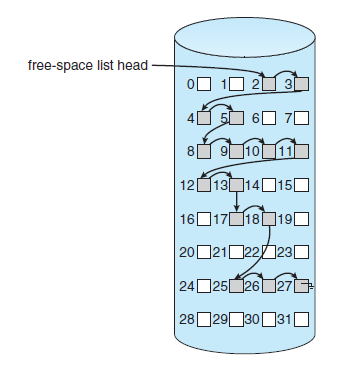
The main advantage of this approach is its relative simplicity and its efficiency in finding the first free lock or *n* consecutive free blocks on the disk.

The calculation of the block number is

(number of bits per word) × (number of 0-value words) + offset of first 1 bit.

**Linked List:**

Another approach to free-space management is to 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. Blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 were free and the rest of the blocks were allocated. In this situation, 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.



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 another *n* free blocks, and so on.

**Counting:**

Another approach takes advantage of the fact that, generally, several contiguous blocks may be allocated or freed simultaneously, particularly when space is allocated with the contiguous-allocation algorithm or through clustering. Thus, rather than keeping a list of *n* free disk addresses, we can keep the address of the first free block and the number (*n*) of free contiguous blocks that follow the first block. Each entry in the free-space list then consists of a disk address and a count.

**Space Maps:**

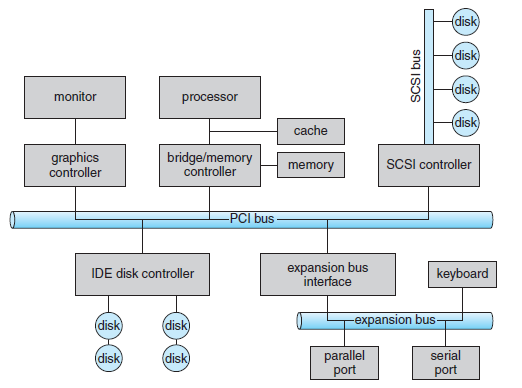
Oracle’s **ZFS** file system was designed to encompass huge numbers of files, directories, and even file systems.

ZFS creates **metaslabs** to divide the space on the device into chunks of manageable size. A given volume may contain hundreds of metaslabs. Each metaslab has an associated space map. ZFS uses the counting algorithm to store information about free blocks and uses log-structured file-system techniques to record them. The space map is a log of all block activity (allocating and freeing), in time order, in counting format. When ZFS decides to allocate or free space from a metaslab, it loads the associated space map into memory in a balanced-tree structure (for very efficient operation), indexed by offset, and replays the log

into that structure.

**I/O Hardware:**

A device communicates with a computer system by sending signals over a cable or even through the air. The device communicates with the machine via a connection point, or **port**—for example, a serial port. If devices share a common set of wires, the connection is called a bus. A **bus** is a set of wires and a rigidly defined protocol that specifies a set of messages that can be sent on the wires. In terms of the electronics, the messages are conveyed by patterns of electrical voltages applied to the wires with defined timings. When device *A* has a cable that plugs into device *B,* and device *B* has a cable that plugs into device *C,* and device *C* plugs into a port on the computer, this arrangement is called a **daisy chain**.



In the figure, a **PCI bus** (the common PC system bus) connects the processor–memory subsystem to fast devices, and an **expansion bus** connects relatively slow devices, such as the keyboard and serial and USB ports. In the upper-right portion of the figure, four disks are connected together on a **Small Computer System Interface (SCSI)** bus plugged into a SCSI controller.

A **controller** is a collection of electronics that can operate a port, a bus, or a device. A serial-port controller is a simple device controller. A SCSI bus controller is not simple, because the SCSI protocol is complex, the SCSI bus controller is often implemented as a separate circuit board (or a **host adapter**) that plugs into the computer. It typically contains a processor, microcode, and some private memory to enable it to process the SCSI protocol messages.

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** contains bits that can be read by the host. These bits 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. For instance, a certain bit in the control register of a serial port chooses between

full-duplex and half-duplex communication.

**Polling:**

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.

For this example, the host writes output through a port, coordinating with the controller by handshaking as follows.

**1.** The host repeatedly reads the busy bit until that bit becomes clear.

**2.** The host sets the write bit in the command register and writes a byte into the data-out register.

**3.** The host sets the command-ready bit.

**4.** When the controller notices that the command-ready bit is set, it sets the busy bit.

**5.** The controller reads the command register and sees the write command. It reads the data-out register to get the byte and does the I/O to the device.

**6.** The controller clears the command-ready bit, clears the error bit in the status register to indicate that the device I/O succeeded, and clears the busy bit to indicate that it is finished.

-------------------------------------------------------------------------------------------------------------------

**Kernel I/O Subsystem:**

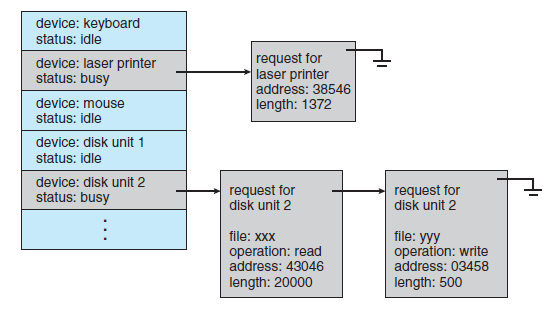
Kernels provide many services related to I/O. Several services—scheduling, buffering, caching, spooling, device reservation, and error handling—are provided by the kernel’s I/O subsystem and build on the hardware and device driver infrastructure.

**I/O Scheduling:**

To schedule a set of I/O requests means to determine a good order in which to execute them. The order in which applications issue system calls rarely is the best choice. Scheduling can improve overall system performance, can share device access fairly among processes, and can reduce the average waiting time for I/O to complete.

For example a disk arm is near the beginning of a disk and that three applications issue blocking read calls to that disk. Application 1 requests a block near the end of the disk, application 2 requests one near the beginning, and application 3 requests one in the middle of the disk. The operating system can reduce the distance that the disk arm travels by serving the applications in the order 2, 3, 1. Rearranging the order of service in this way is the essence of I/O scheduling. The operating system may also try to be fair, so that no one application receives especially poor service, or it may give priority service for delay-sensitive requests.

When a kernel supports asynchronous I/O, it must be able to keep track of many I/O requests at the same time. For this purpose, the operating system might attach the wait queue to a **device-status table**. The kernel manages this table, which contains an entry for each I/O device. Each table entry indicates the device’s type, address, and state (not functioning, idle, or busy). If the device is busy with a request, the type of request and other parameters will be stored in the table entry for that device.



**Device-status table**

**Buffering:**

A **buffer** is a memory area that stores data being transferred between two devices or between a device and an application. Buffering is done for three reasons.

One reason is to cope with a speed mismatch between the producer and consumer of a data stream. Suppose, for example, that a file is being received via modem for storage on the hard disk. The modem is about a thousand times slower than the hard disk. So a buffer is created in main memory to accumulate the bytes received from the modem. When an entire buffer of data has arrived, the buffer can be written to disk in a single operation.

A second use of buffering is to provide adaptations for devices that have different data-transfer sizes. Such disparities are especially common in computer networking, where buffers are used widely for fragmentation and reassembly of messages. At the sending side, a large message is fragmented into small network packets. The packets are sent over the network, and the receiving side places them in a reassembly buffer to form an image of the source data.

A third use of buffering is to support copy semantics for application I/O.

**Caching:**

A **cache** is a region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original. The difference between a buffer and a cache is that a buffer may hold the only existing copy of a data item, whereas a cache, by definition, holds a copy on faster storage of an item that resides elsewhere.

**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. Although a printer can serve only one job at a time, several applications may wish to print their output concurrently, without having their output mixed together. The operating system solves this problem by intercepting all output to the printer. Each application’s output is spooled to a separate disk file. When an application finishes printing, the spooling system queues the corresponding spool file for output to the printer. The spooling system copies the queued spool files to the printer one at a time.

Some operating systems (including VMS) provide support for exclusive device access by enabling a process to allocate an idle device and to deallocate that device when it is no longer needed. Other operating systems enforce a limit of one open file handle to such a device.

**Error Handling:**

An operating system that uses protected memory can guard against many kinds of hardware and application errors, so that a complete system failure is not the usual result of each minor mechanical malfunction. Devices and I/O transfers can fail in many ways, either for transient reasons, as when a network becomes overloaded, or for “permanent” reasons, as when a disk controller becomes defective. Operating systems can often compensate effectively for transient failures. For instance, a disk read() failure results in a read() retry, and a network send() error results in a resend(), if the protocol so specifies.

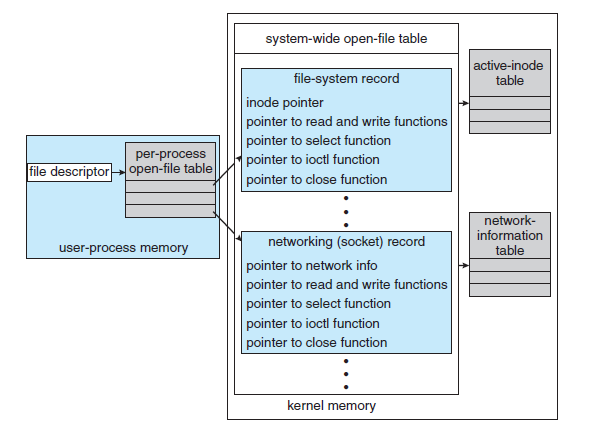
**I/O Protection:**

Errors are closely related to the issue of protection. A user process may accidentally or purposely attempt to disrupt the normal operation of a system by attempting to issue illegal I/O instructions.

To prevent users from performing illegal I/O, we define all I/O instructions to be privileged instructions. Thus, users cannot issue I/O instructions directly; they must do it through the operating system. To do I/O, a user program executes a system call to request that the operating system perform I/O on its behalf. The operating system, executing in monitor mode, checks that the request is valid and, if it is, does the I/O requested. The operating system then returns to the user.

**Kernel Data Structures:**

The kernel needs to keep state information about the use of I/O components. It does so through a variety of in-kernel data structures, such as the open-file table structure. The kernel uses many similar structures to track network connections, character-device communications, and other I/O activities.



**UNIX I/O kernel structure**

The I/O subsystem supervises these procedures:

• Management of the name space for files and devices

• Access control to files and devices

• Operation control (for example, a modem cannot seek ())

• File-system space allocation

• Device allocation

• Buffering, caching, and spooling

• I/O scheduling

• Device-status monitoring, error handling, and failure recovery

• Device-driver configuration and initialization

---------------------------------------------------------------------------------------------------------------------------

**Transforming I/O Requests to Hardware Operations:**

An I/O operation requires a great many steps that together consume a tremendous number of CPU cycles.

1. A process issues a blocking read() system call to a file descriptor of a file that has been opened previously.

2. The system-call code in the kernel checks the parameters for correctness. In the case of input, if the data are already available in the buffer cache, the data are returned to the process, and the I/O request is completed.

3. Otherwise, a physical I/O must be performed. The process is removed from the run queue and is placed on the wait queue for the device, and the I/O request is scheduled. Eventually, the I/O subsystem sends the request to the device driver. Depending on the operating system, the request is sent via a subroutine call or an in-kernel message.

4. The device driver allocates kernel buffer space to receive the data and schedules the I/O. Eventually, the driver sends commands to the device controller by writing into the device-control registers.

5. The device controller operates the device hardware to perform the data transfer.

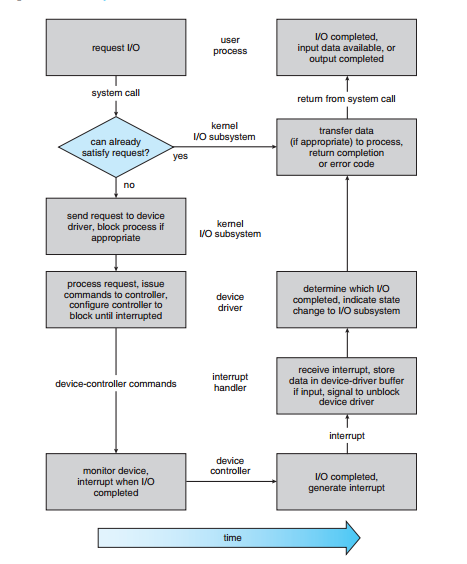


Figure: The life cycle of an I/O request

6. The driver may poll for status and data, or it may have set up a DMA transfer into kernel memory. We assume that the transfer is managed by a DMA controller, which generates an interrupt when the transfer completes.

7. The correct interrupt handler receives the interrupt via the interruptvector table, stores any necessary data, signals the device driver, and returns from the interrupt.

8. The device driver receives the signal, determines which I/O request has completed, determines the request’s status, and signals the kernel I/O subsystem that the request has been completed.

9. The kernel transfers data or return codes to the address space of the requesting process and moves the process from the wait queue back to the ready queue.

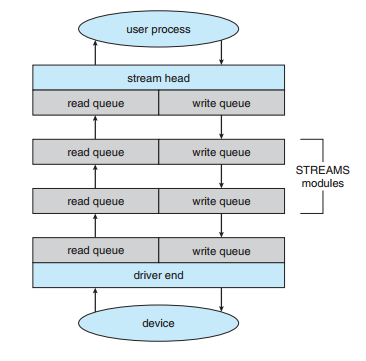
10. Moving the process to the ready queue unblocks the process. When the scheduler assigns the process to the CPU, the process resumes execution at the completion of the system call.

---------------------------------------------------------------------------------------------------------------------------

**STREAMS:**

UNIX System V has an interesting mechanism, called STREAMS, that enables an application to assemble pipelines of driver code dynamically. 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.

Modules provide the functionality of STREAMS processing; they are pushed onto a stream by use of the ioctl() system call. For example, a process can open a serial-port device via a stream and can push on a module to handle input editing. Because messages are exchanged between queues in adjacent modules, a queue in one module may overflow an adjacent queue. To prevent this from occurring, a queue may support flow control.

****

**Figure: Stream Structure**

Without flow control, a queue accepts all messages and immediately sends them on to the queue in the adjacent module without buffering them. A queue that supports flow control buffers messages and does not accept messages without sufficient buffer space. This process involves exchanges of control messages between queues in adjacent modules.

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.

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 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.