**IV Unit**

**File Management**

**FILE CONCEPT**

Computers can store information on various storage media, such as magnetic disks, magnetic tapes, and optical disks, so that the computer system will be convenient to use. These storage devices are usually nonvolatile, so the contents are persistent through power failures and system reboots.

A file is a named collection of related information that is recorded on secondary storage. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. The information in a file is defined by its creator.

Many different types of information may be stored in a file - source programs, object programs,executable programs, numeric data, text, payroll records, graphic images, sound recordings

**File Attributes**

A file is named, for the convenience of its human users, and is referred to by its name. A name is usually a string of characters, such as ***example.c.***

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,.
* **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.

**File Operations**

File is an **abstract data type.** To define a file properly, we need to consider the operations that can be performed on files. 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. The directory entry records the name of the file and the location in the file system.
* **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. Given the name of the file, the system searches the directory to find the file's location. The system must keep a *write* pointer to the location in the file where the next write is to take place.
* **Reading a file.** To read from a file, we use a system call that specifies the name of the file and where (in memory) 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. Once the read has taken place, the read pointer is updated.
* **Repositioning within a file.** The directory is searched for the appropriate entry, and the current-file-position is set 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. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged—except for file length—but lets the file be reset to length zero and its file space released

**Access Methods**

File access mechanism refers to the manner in which the records of a file may be accessed. There are several ways to access files −

* Sequential access
* Direct/Random access
* Indexed sequential access

**Sequential access**

A sequential access is that in which the records are accessed in some sequence, i.e., the information in the file is processed in order, one record after the other. This access method is the most primitive one. Example: Compilers usually access files in this fashion.

**Direct/Random access**

* Random access file organization provides, accessing the records directly.
* Each record has its own address on the file with by the help of which it can be directly accessed for reading or writing.
* The records need not be in any sequence within the file and they need not be in adjacent locations on the storage medium.

**Indexed sequential access**

* This mechanism is built up on base of sequential access.
* An index is created for each file which contains pointers to various blocks.
* Index is searched sequentially and its pointer is used to access the file directly.

**DIRECTORY STRUCTURE**

Systems may have zero or more file systems. Some systems store millions of files on terabytes of disk. To manage all these data, we need to organize them. This organization involves the use of directories.

**Storage Structure**

A disk (or any storage device that is large enough) can be used for a file system. Sometimes, it is desirable to place multiple file systems on a disk or to use parts of a disk for a file system and other parts for other things. These parts are known as **partitions.**



The operations that are to be performed on a directory are as follows:

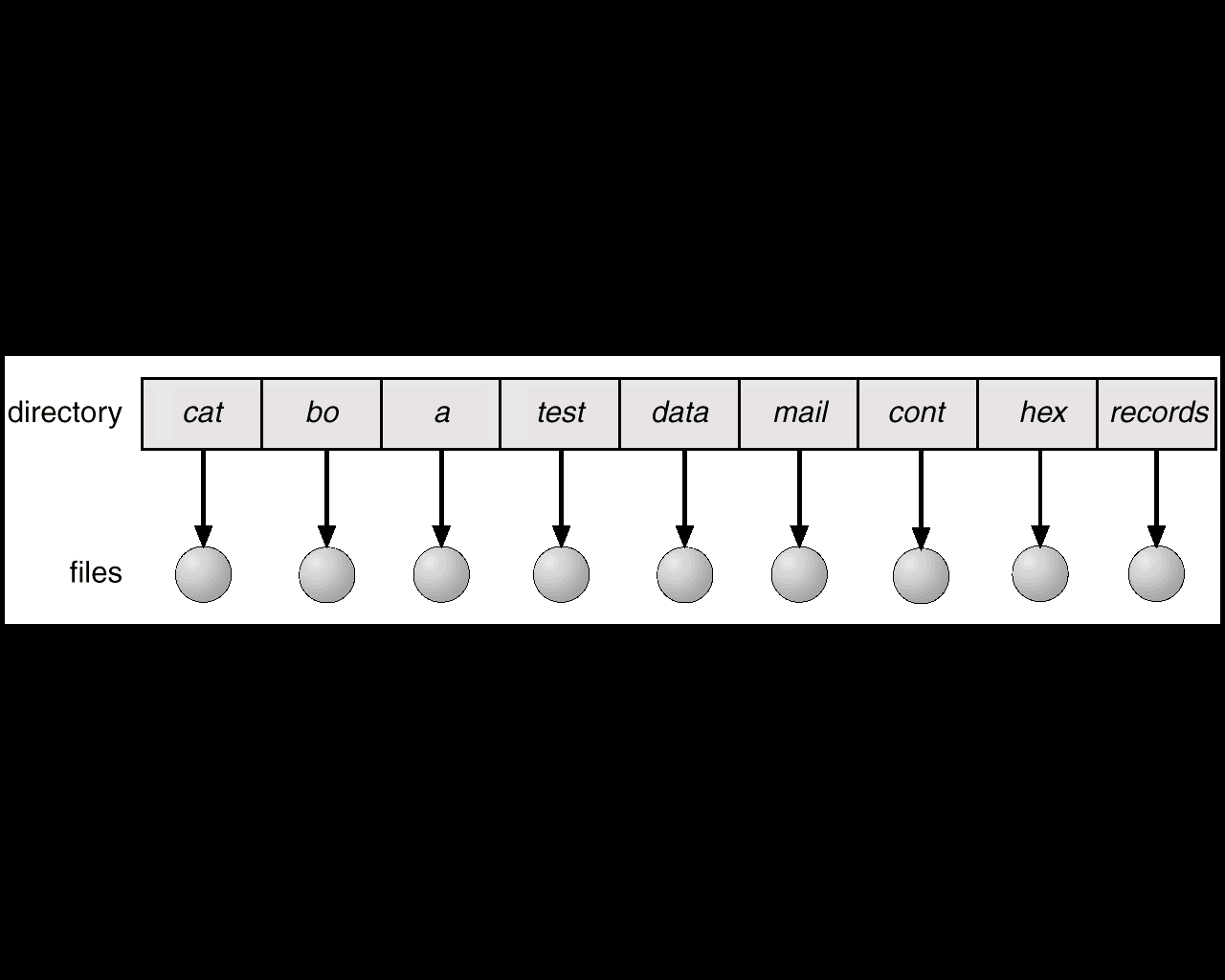
1. **Search for a file:** We can search a directory structure to find the entry for a particular file. We can search all files whose names match a particular pattern.
2. **Create a file.** New files need to be created and added to the directory.
3. **Delete a file**. When a file is no longer needed, we can remove it from the directory using delete operation.
4. **List a directory**. We can list the files in a directory and the contents of the directory entry for each file in the list.
5. **Rename a file**. Because the name of a file represents its contents to its users, we can change the name when the contents or use of the file changes. Renaming a file also allow its position within the directory structure to be changed.
6. **Traverse the file system**. We can access every directory and file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals. We do this by copying all files to magnetic tape. This technique provides a backup copy in case of system failure. In addition, if a file is no longer in use, the file can be copied to tape and the disk space of that file released for reuse by another file.

**Single-Level Directory**

The simplest directory structure is the single-level directory.

**Limitations:**

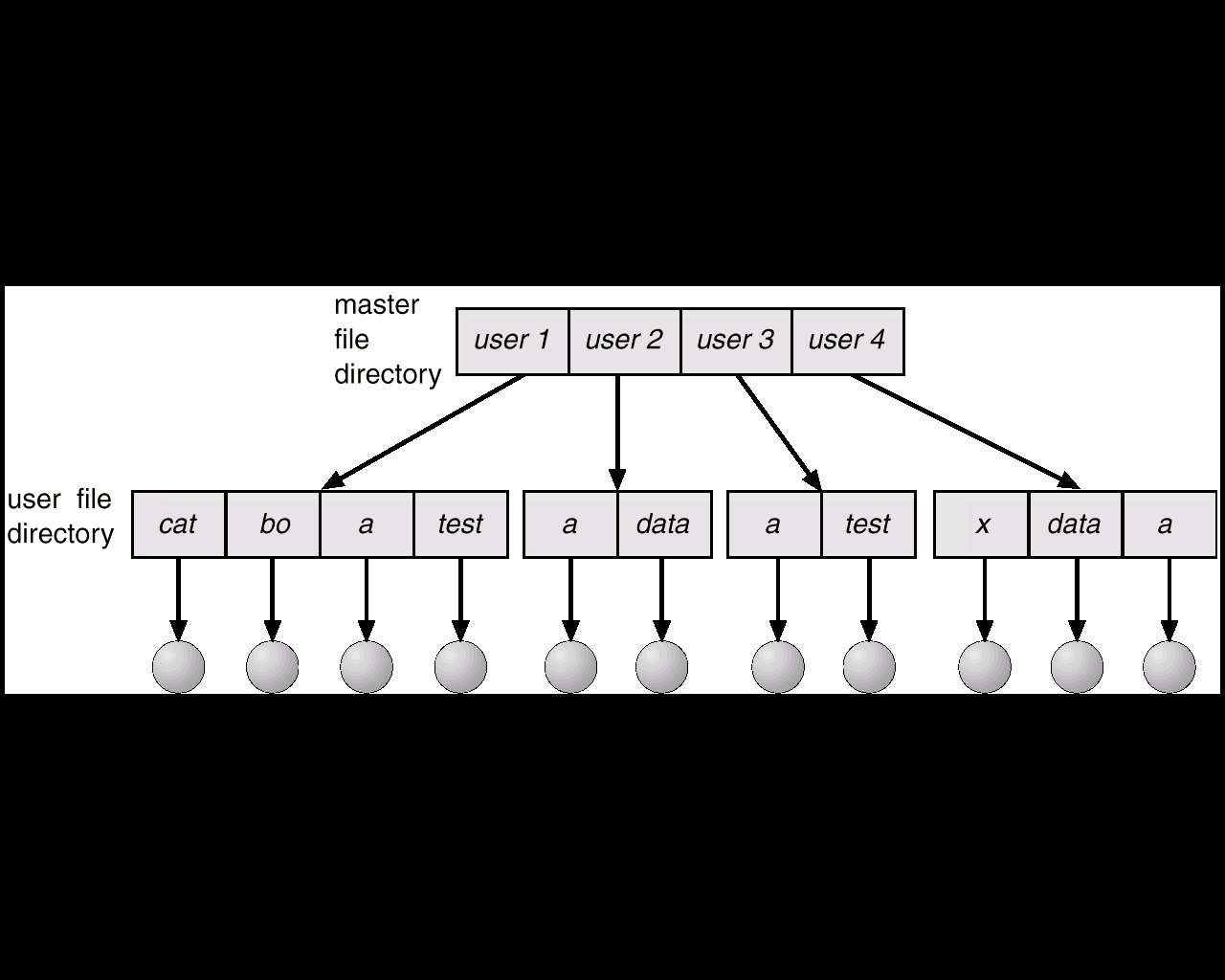
* Since all files are in the same directory, they must have unique name.
* Files are limited in length.
* Even a single user may find it difficult to remember the names of all files as the number of file increases.
* Keeping track of so many file is daunting task.

****

**Two-Level Directory**

Single-level directory leads to confusion of file names among different users. The standard solution is to create a *separate* directory for each user.

In the two-level directory structure, each user has his own user file directory (UFD) .When a user job starts or a user logs in, the system's master file directory (MFD) is searched. The user directories can be created and deleted as necessary. The program creates a new UFD and adds an entry for it to the MFD.



**Limitations:**

Lack of cooperation between the different users i.e., when the one user *want* to access the another user files which are not allowed in some systems.

A two-level directory is also called as a tree, or an inverted tree, of height 2. The root of the tree is the MFD. Its direct descendants are the UFDs. The descendants of the UFDs are the files themselves. The files are the leaves of the tree. Specifying a user name and a file name defines a path in the tree from the root (the MFD) to a leaf (the specified file).

A user name and a file name is defined a *path name.* Every file in the system has a path name.

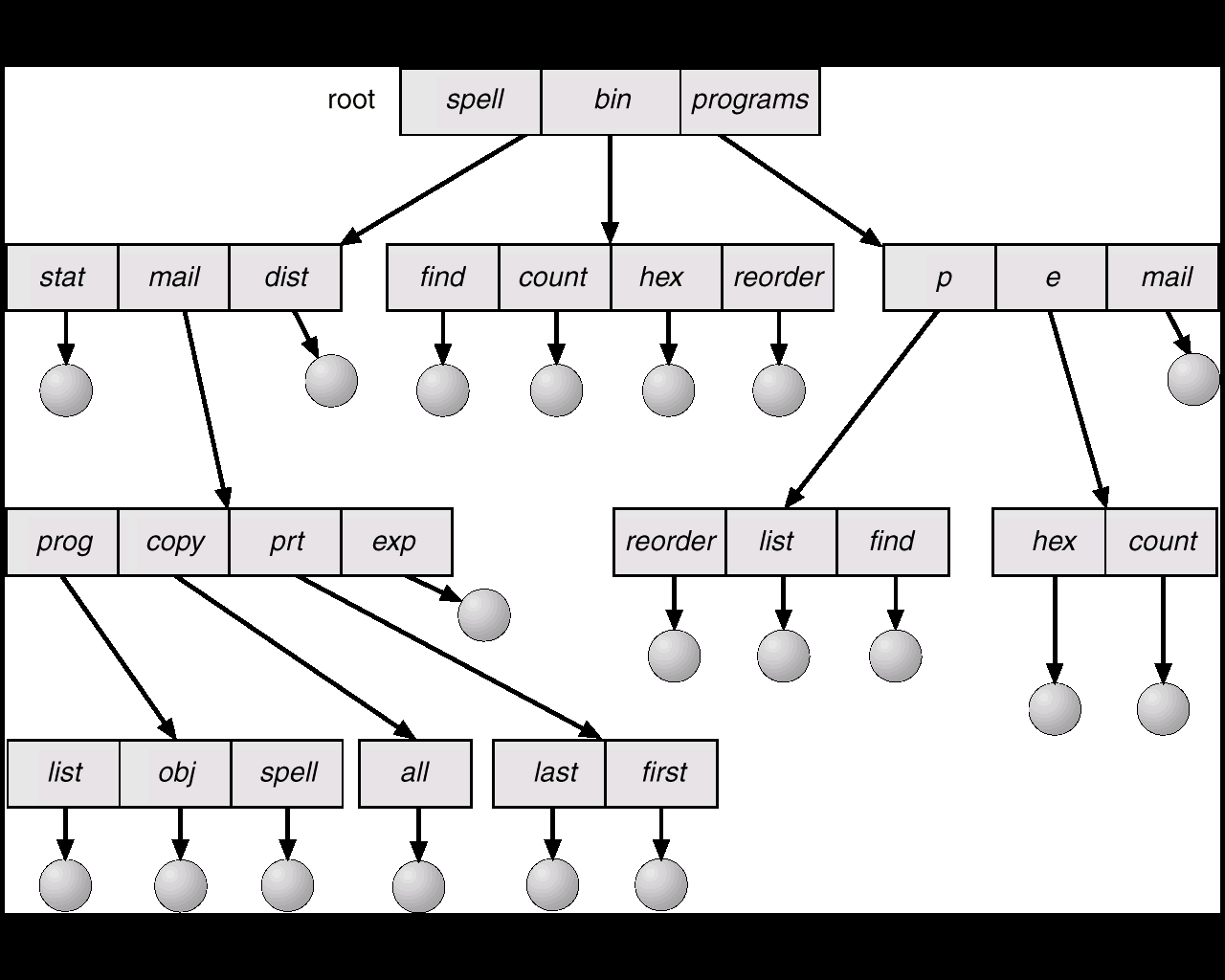
For example, if user A wishes to access her own file named *test,* she can simply refer to *test.* To access the file named *test* of user B (with directory-entry

name *userb),* we can refer to */userb/test.*

**Tree-Structured Directories**

The natural generalization of two-level directory is to extend the directory structure to a tree of arbitrary height (Figure 10.9). 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.

A directory (or subdirectory) contains a set of files or subdirectories and it is also treated as another file.



In general each process has a current directory, whichshould 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 isneeded that is not in the current directory, then the user must eitherspecify a path name or change the current directory to be the directory holdingthat file.

Path names are 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.

With a tree-structured directory system, users can be allowed to access, in addition to their files, the files of other users. For example, user B can access a file of user A by specifying its path names. User B can specify either an absolute or a relative path name. Alternatively, user B can change her current directory to be user A's directory and access the file by its file names.

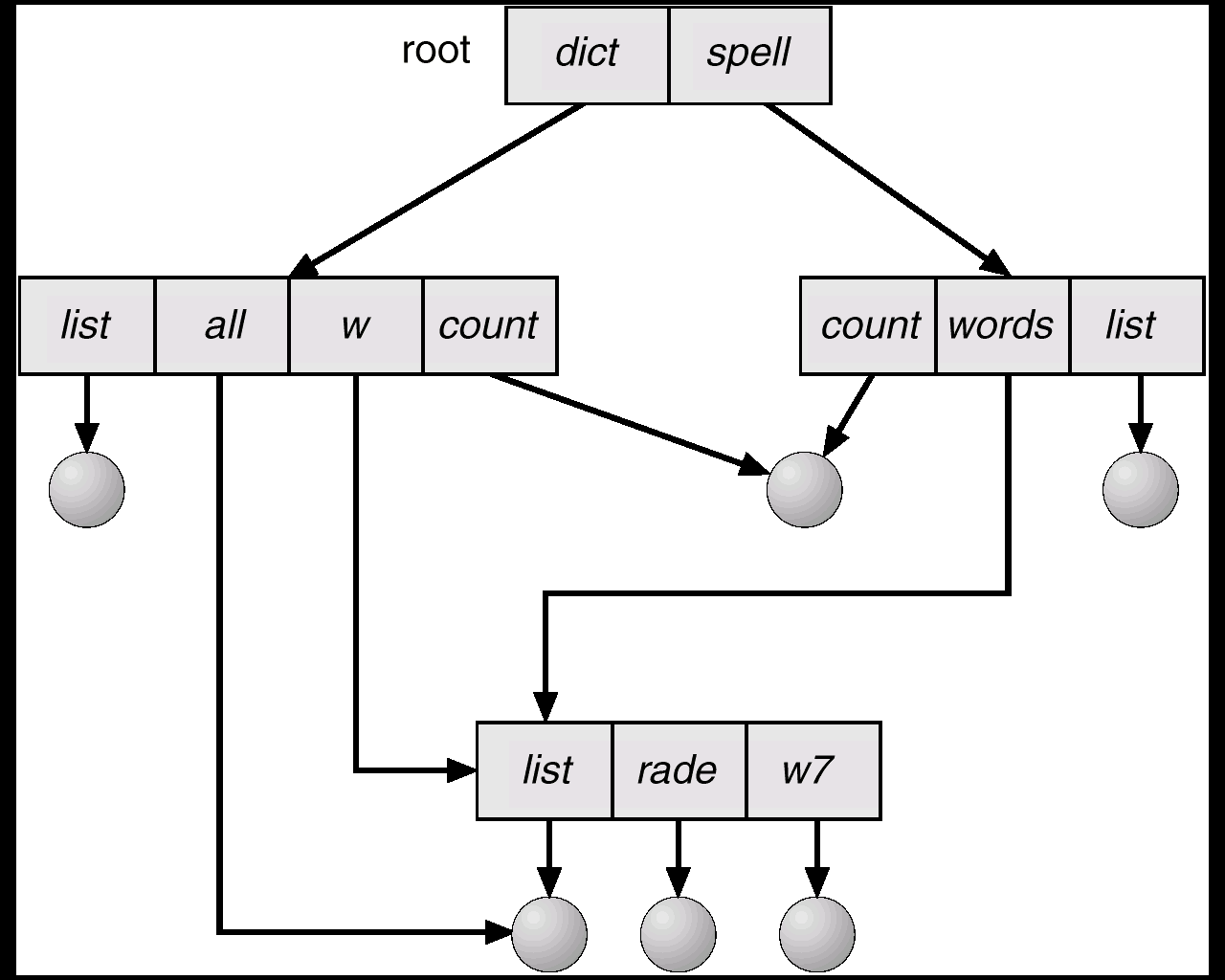
When a new hard disk or floppy disk is added to the system or the network is accessed, the operating system traverses the directory structure, searching for executable programs on the device and recording the relevant information.

This mechanism supports the double-click execution functionality. A double-click on a file causes its creator attribute to be read and the *Desktop File* to be searched for a match. Once the match is found, the appropriate executable program is started with the clicked-on file as its input.

**Acyclic-Graph Directories**

Consider two programmers who are working on a joint project. The files associated

with that project can be stored in a subdirectory, separating them from other projects and files of the two programmers. But since both programmers are equally responsible for the project, both want the subdirectory to be in their own directories. The common subdirectory should be *shared. A* shared directory *or* file will exist in the file system in two (or more) places at once.



A tree structure prohibits the sharing of files or directories. An **acyclic graph** —that is, a graph with no cycles—allows directories to share subdirectories and files (Figure 10.10). The *same* file or subdirectory may be in two different directories. The acyclic graph is a natural generalization of the tree-structured directory scheme.

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.

An acyclic-graph directory structure is more flexible than is 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. Consequently, distinct file names may refer to the same file. This situation is similar to the aliasing problem for programming languages. If we are trying to traverse the entire file system—to find a file, to accumulate statistics on all files, or to copy all files to backup storage—this problem becomes significant, since we do not want to traverse shared structures more than once

**Protection**

When information is kept in a computer system, a major concern is its protection from both physical damage *(reliability)* and improper access *(protection).*

Reliability is generally provided by duplicate copies of files. Many computers have systems programs that automatically copy disk files to tape at regular intervals (once per day or weekor month) to maintain a copy should a file system be accidentally destroyed.

File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes, and vandalism. Files may be deleted accidentally.

Protection can be provided in many ways. For a small single-user system, we might provide protection by physically removing the floppy disks and locking them in a desk drawer or file cabinet. In a multiuser system, however, other mechanisms are needed.

**Types of Access**

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.

Other operations, such as renaming, copying, or editing the file, may also be controlled.

**Access Lists and Groups**

The most common approach to the protection problem is to make access dependent on the identity of the user. Various 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 list,* specifying the user name 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.

To condense the length of the access 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 workgroup.
* Universe. All other users in the system constitute the universe

As an example, 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.* The protection associated with this file is as follows:

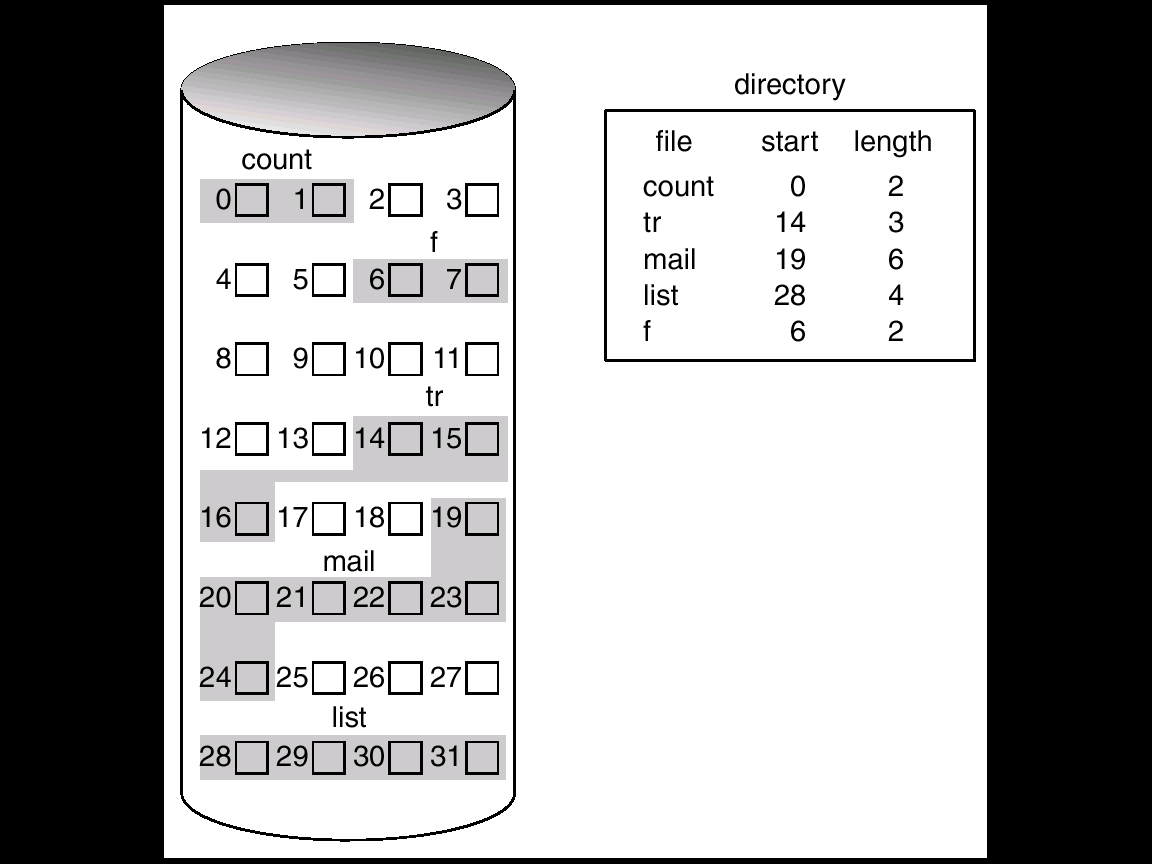
* Sara should be able to invoke all operations on the file.
* Jim, Dawn, and Jill should be able only to read and write the file; they should not be allowed to delete the file.
* All other users should be able to read the file. (Sara is interested in letting as many people as possible read the text so that she can obtain appropriate feedback.)

**ALLOCATION METHODS**

The direct-access nature of disks allows us flexibility in the implementation of files. In almost every case, many files will be stored on the same disk. The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly. Three major methodsof allocating disk space are in wide use: ***contiguous, linked,* and *indexed****.* Each method has its advantages and disadvantages.

**Contiguous allocation**

The***contiguous***allocation method requires each file to 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.

One difficulty with contiguous allocation is finding space for a new file. The implementation of the free-space management system, determines how this task is accomplished.

These algorithms suffer from the problem of ***external fragmentation****.* 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;

There are other problems with contiguous allocation. A major problem 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. How does the creator (program or person) know the size of the file to be created.

If we allocate too little space to a file, we may find that that file cannot be extended .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. To prevent them, the user will normally overestimate the amount of space needed, resulting in Considerable wasted space.

The other possibility is to find a larger hole, to copy the contents of the file to the new space, and to release the previous space. This series of actions may be repeated as long as space exists, although it can also be time-consuming.

Even if the total amount of space needed for a file is known in advance, preallocation may be inefficient. A file that grows slowly over a long period (months or years) must be allocated enough space for its final size, even though much of that space may be unused for a long time. The file, therefore, has a"large amount of ***internal fragmentation****.*

**Linked Allocation**

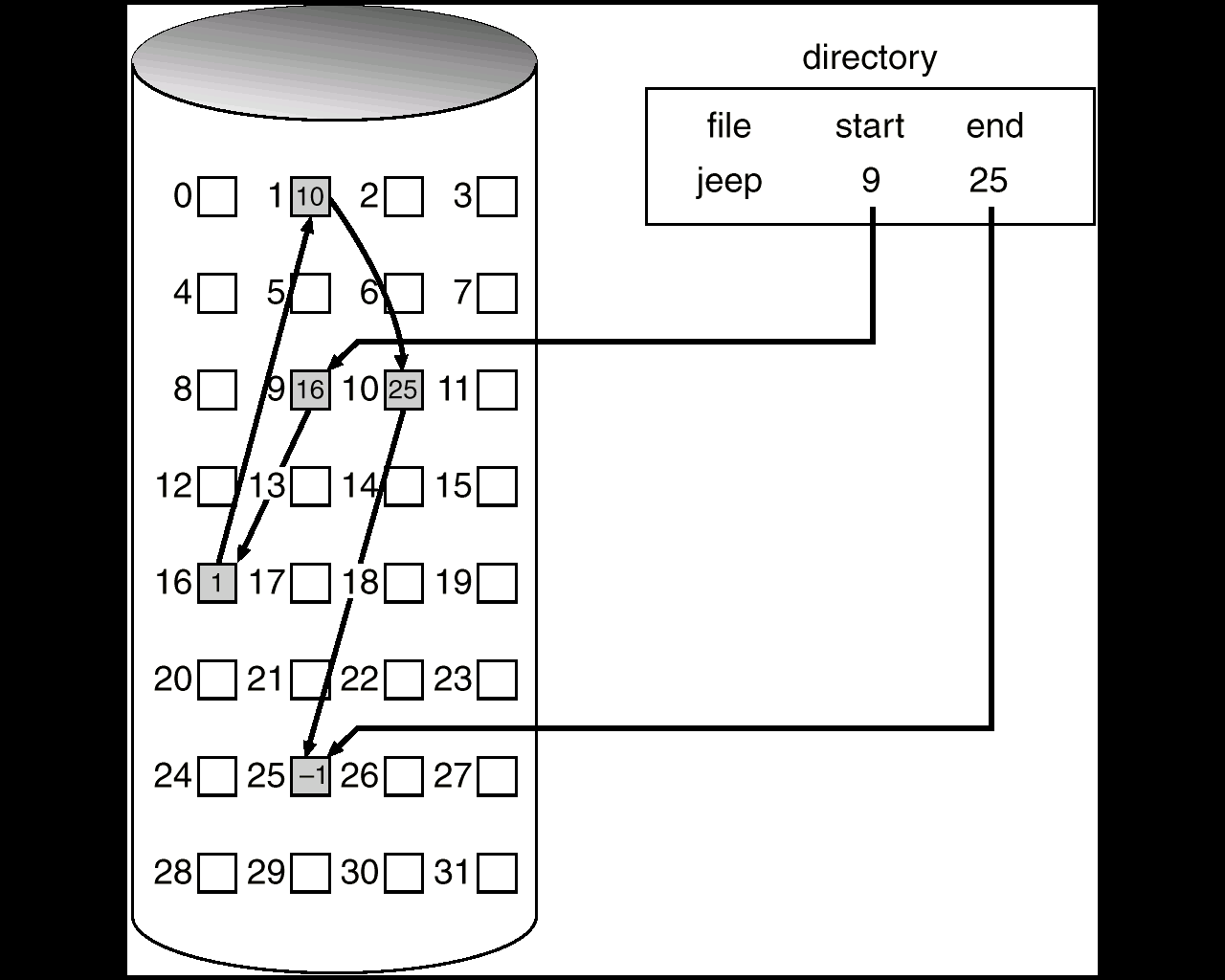
***Linked allocation***solves all problems of contiguous allocation. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. For example, a file of five blocks might start at block 9, continue at block 16, then block 1, block 10, and finally block 25

Each block contains a pointer to the next block. These pointers are not made available to the user. Thus, if each block is 512 bytes, and a disk address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes.

To create a new file, we simply create a new entry in the directory. With linked allocation, each directory entry has a pointer to the first disk block of the file. This pointer is initialized to *nil* (the end-of-list pointer value) to signify an empty file. The size field is also set to 0. A write to the file causes a free block to be found via the free-space management system , and this new block is then 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.

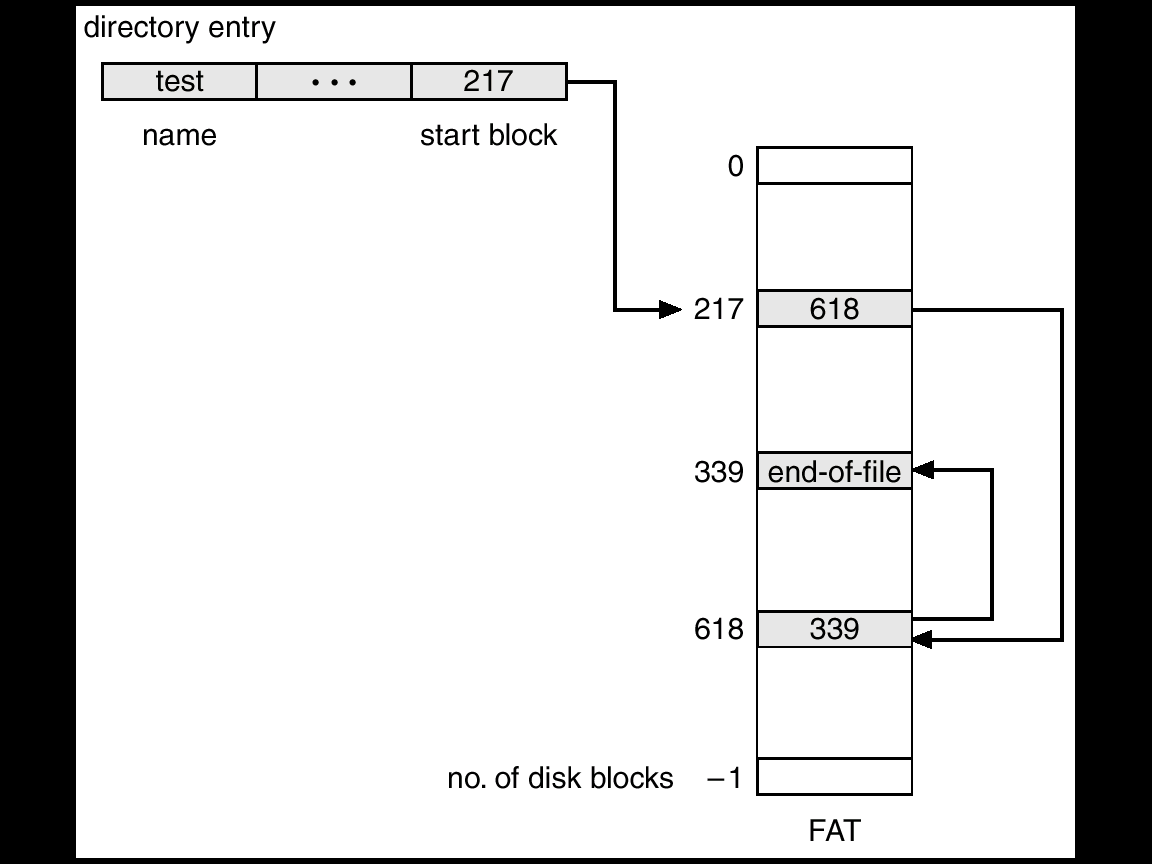
There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request.

Linked allocation does have disadvantages, however. The major problem is that it can be used effectively for only sequential-access files. To find the ith block of a file, we must start at the beginning of that file, and follow the pointers until we get to the ith block.



The disadvantage to linked allocation is the space required for the pointers. If a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information.

An important variation on the linked allocation method is the use of *a* ***file allocation table (FAT)****.* This simple but efficient method of disk-space allocation is used by the MS-DOS and OS/2 operating systems. A section of disk at the beginning of each partition 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 much as is a linked list.



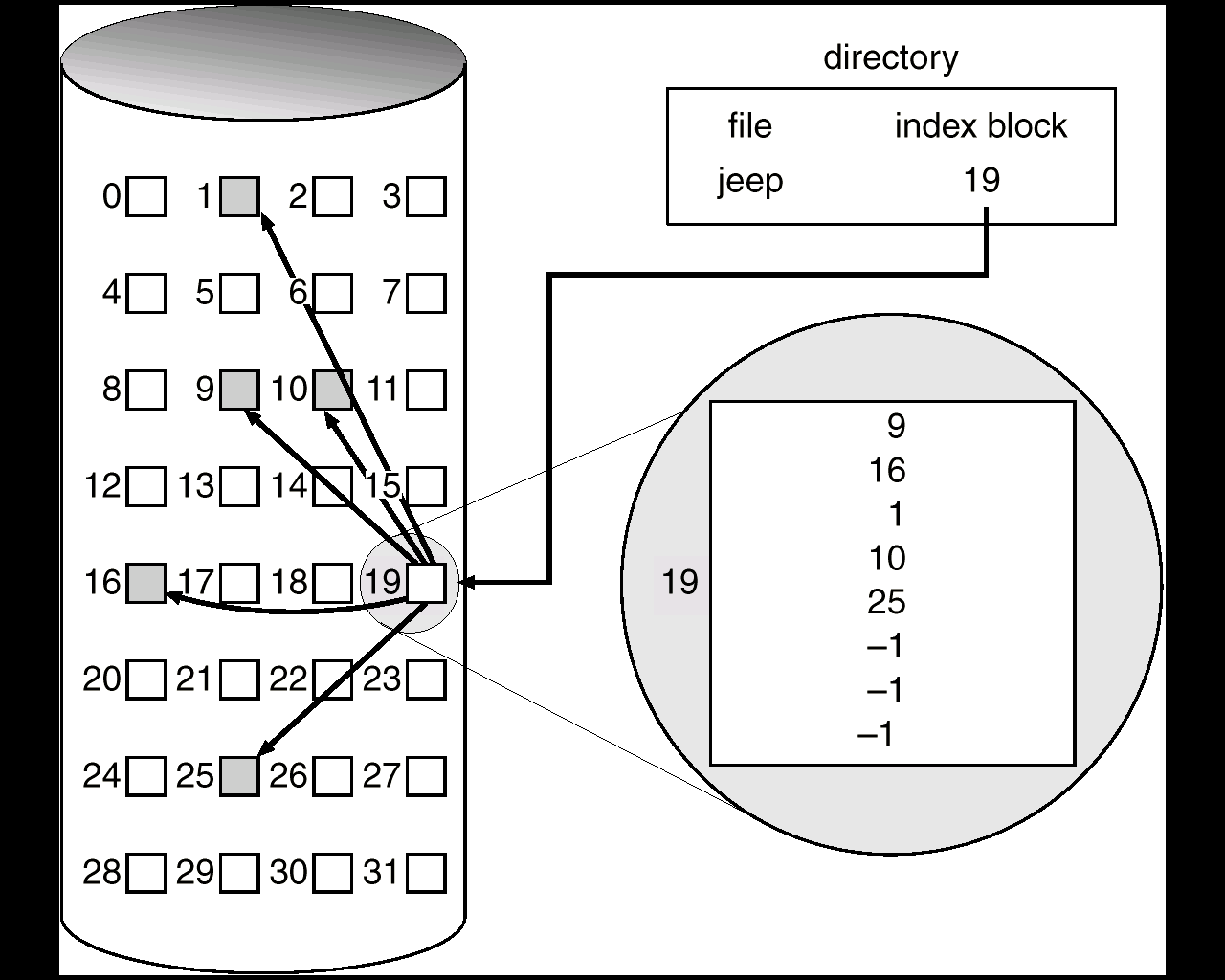
The directory entry contains the block number of the first block of the file. The table entry indexed by that block number then contains the block number of the next block in the file. This chain continues until the last block, which has a special end-of-file value as the table entry. Unused blocks are indicated by a 0 table value. Allocating a new block to a file is a simple matter of finding the first 0-valued table entry, and replacing the previous end-of-file value with the address of the new block.

An illustrative example is the FAT structure of Figure for a file consisting of disk blocks 217, 618, and 339.

**Indexed Allocation**

Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation. 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 (Figure 11.6). To read the ith block, we use the pointer in the ith index-block entry to find and read the desired block.



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

Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk may satisfy a request for more space.

Indexed allocation does suffer from wasted space. The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation. Consider a common case in which we have a file of only one or two blocks. With linked allocation, we lose the space of only one pointer per block.

This point raises the question of 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, and a mechanism will have to be available to deal with this issue:

**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 may, link together several index blocks. For example, an index block might contain a small header giving the name of the file, and a set of the first 100 disk-block

addresses- The next address (the last word in the index block) is *nil* (for a small file) or is a pointer to another index block (for a large file).

**Free-Space Management**

Since there is only a limited amount of disk space, it is necessary 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 disk blocks that *are free* — 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.

**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.

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

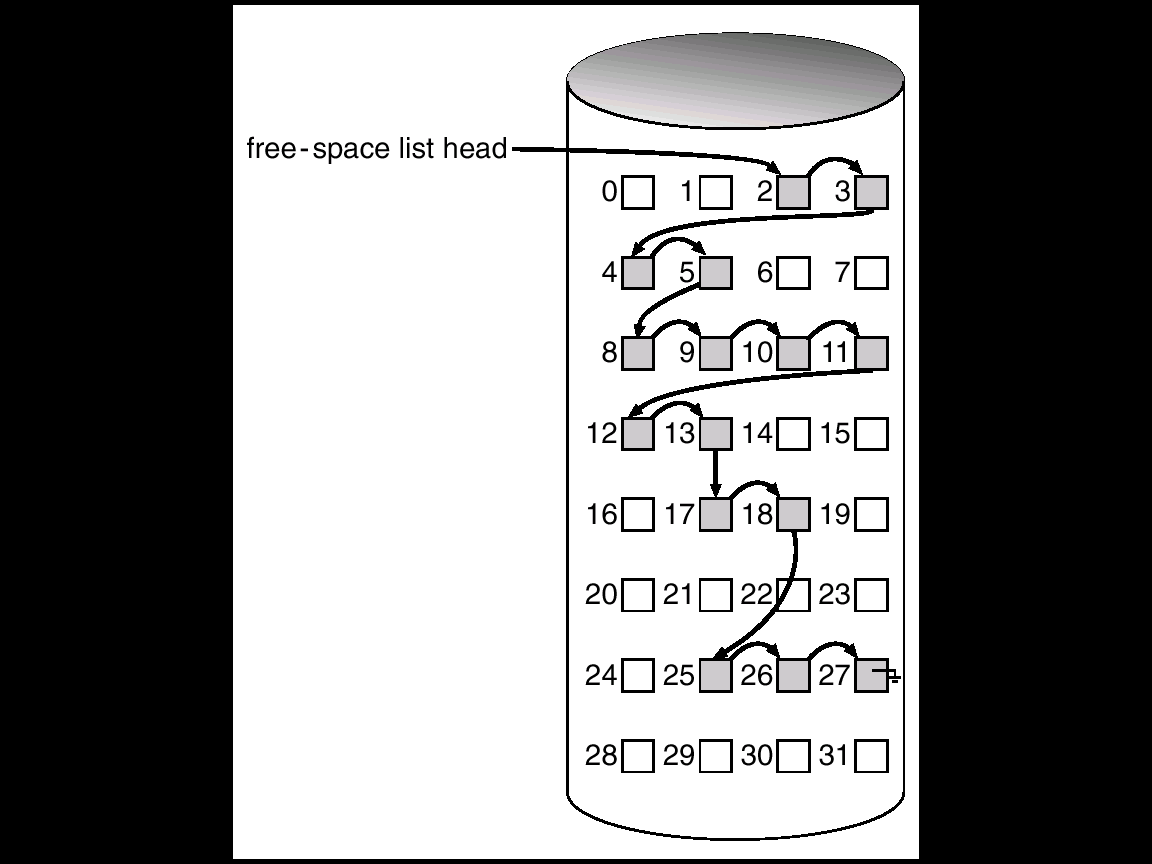
001111001111110001100000011100000 ...

The main advantage of this approach is that it is relatively simple and efficient to find the first free block, or *n* consecutive free blocks on the disk.

**Linked List**

Another approach 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. 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.

However, 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 is to store the addresses of *n* free blocks in the first free block. The first n-1of these blocks are actually free. The last block contains the addresses of another *n* free blocks, and so on. The importance of this implementation is that the addresses of a large number of free blocks can be found quickly.

**Counting**

Another approach is to take 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. 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. Although each entry requires more space than would a simple disk address, the overall list will be shorter, as long as the count is generally greater than 1.

**GoALS OF PROTECTION**

**Protection:**

Protection is a mechanism of controlling access of computer resources by users or processes. A protection enabled system can find the differences between authorized and unauthorized access or usage and can take measures to defend the system against misuse. If protection is not employed then errors may also occur among subcomponents of system. This happens usually when a defected subsystem interacts with healthy subsystem through its interface. Then healthy subsystem gets corrupted.

**Goals of Protection:**

There are several reasons for providing protection.

* The need of protection is to prevent mischievous, intentional violation of an access restriction by a user and others.
* To prevent intentional damage and violation of access restriction by a user.
* To ensure that each program should program should use the system resources in a consistent and desires manner only.
* To defend the system from unauthorized users.

A protection system must have the flexibility to enforce a variety of policies that can be declared to it. One important principle is the separation of *policy* from *mechanism.* Mechanisms determine *how something* will be done. In contrast, policies decide *what* will be done.

**Domain of Protection**

A computer system is a collection of processes and objects. There are two types of objects. They are

* Hardware objects such as the CPU, memory segments, printers, disks, and tape drives
* Software objects such as files, programs.

Each object is identified in the system with its unique name and there are some specific operations only through which we can access them .For example memory can read and written whereas DVD-ROM, CD-ROM can only be read.

A process should be allowed to access only those resources it has been authorized to access. Furthermore, at any time, it should be able to access only those resources that it currently requires to complete its task.

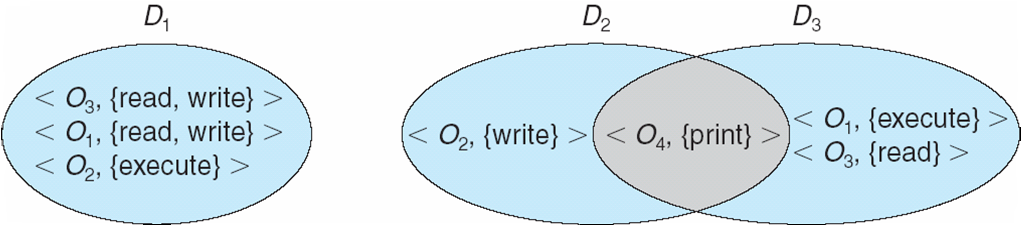
This requirement, commonly referred to as the ***need-to-know***principle, is useful in limiting the amount of damage a faulty process can cause in the system.

**Domain Structure**

To implement this protection scheme, every process has to be executed within a protection domain which specifies the objects and resources that a particular process can access. Each domain defines a set of objects and the types of operations that may be invoked on each object. The ability to execute an operation on an object is an *access right. A domain* is a collection of access rights, each of which is an ordered pair *<object-name, rights-set>.*

For example, if domain D has an access right **<file F,{read, write}>,**then a process executingin domain D can both read and write file F; it cannot, perform any other operation on that object.

Domains do not need to be disjoint; they may share access rights. For example, in below figure , we have three domains: *D1, D2,* and *D3.* The access right <O4, {print}> is shared by both D2 and D3, implying that a process executing in either one of these two domains can print object O4. Note that a process must be executing in domain D1 to read and write object O1. On the other hand, only processes in domain D3 may execute object O1*.*

**

If a domain is static, we must define the domain to include both read and write access. However, this arrangement provides more rights than needed in each of the two phases, since we have read access in the phase where we need only write access, and vice versa. Thus, the *need-to-know* principle is violated.

If the association is dynamic, a mechanism is available to allow a process to switch from one domain to another. We may also want to allow the content of a domain to be changed. If we cannot change the content of a domain, we can provide the same effect by creating a new domain with the changed content, and switching to that new domain.

Domain can be implemented in a variety of ways:

* **Each *user* may be a domain:** In this case, the set of objects that can be accessed depends on the identity of the user. Domain switching occurs when the user is changed - generally when one user logs out and another user logs in.
* **Each *process* may be a domain:** In this case, the set of objects that can be accessed depends on the identity of the process. Domain switching corresponds to one process sending a message to another process, and then waiting for a response.
* **Each *procedure* may be a domain:** In this case, the set of objects that can be accessed corresponds to the local variables defined within the procedure. Domain switching occurs when a procedure call is made.

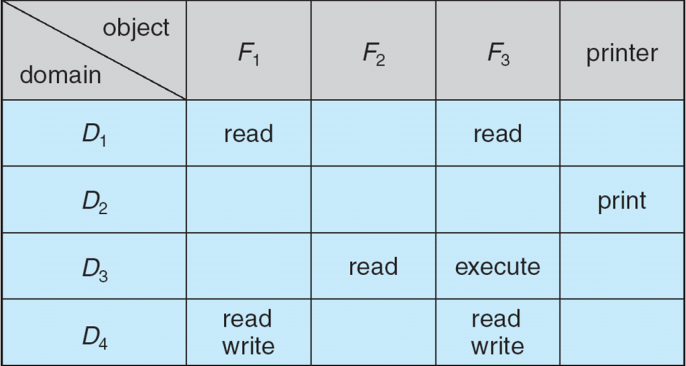
**Access Matrix**

Protection can be viewed abstractly as a matrix, called an *access matrix.* The rows of the access matrix represent *domains*, and the columns represent *objects*. Each entry in the matrix consists of a set of *access rights*. Because objects are defined explicitly by the column, we can omit the object name from the access right. The entry access(i,j) defines the set of operations that a process, executing in domain Di, can invoke on object Oj.

To illustrate these concepts, we consider the access matrix shown in below Figure There are four domains and four objects: three files (F1, F2, F3), and one laser printer. When a process executes in domain *D1,* it can read files F1 and F2.

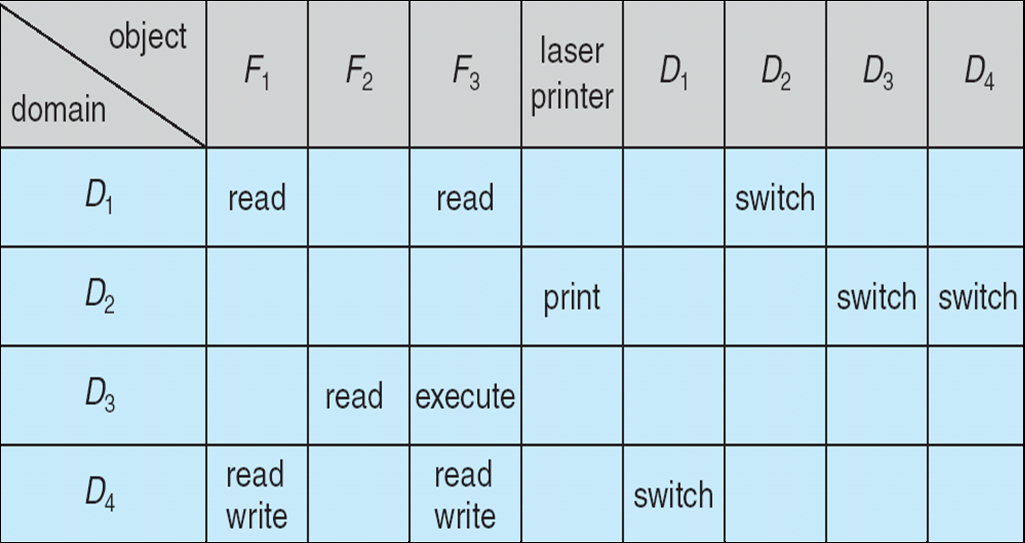
A process executing in domain D4 has the same privileges as it does in domain D1, but in addition, it can also write onto files F1 and *F3.* Note that the laser printer can be accessed only by a process executing in domain D2.

The access-matrix scheme provides mechanism for specifying a variety of policies. More specifically, we must ensure that a process executing in domain Di, can access only those objects specified in row *i.*

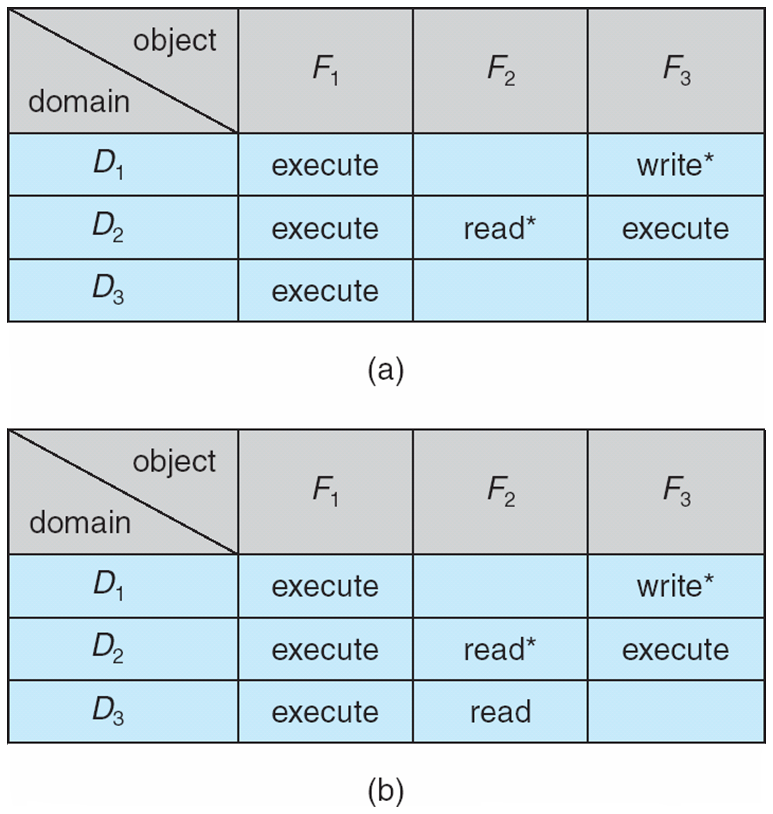
**

Processes should be able to switch from one domain to another. Domain switching from domain Di to domain Dj is allowed to occur if and only if the access right *switch***∈**access(i, j). Thus, in below figure, a process executing in domain D2 can switch to domain D3 or to domain D4. A process in domain *D4*can switch to *D1,* and one in domain D1can switch to domain D2.

Allowing controlled change to the contents of the access-matrix entries, requires three additional operations: copy, owner, and control.



The ability to copy an access right from one domain (row) of the access matrix to another is denoted by an asterisk (\*) appended to the access right.



The *copy* right allows the copying of the access right only within the column (that is, for the object) for which the right is denied

For example, in the above Figure (a) a process executing in domain D2 can copy the read operation into any entry associated with file F2. Hence, the access matrix of Figure (a) can be modified to the access matrix shown in the above Figure (b)

1. There are two variants to this scheme:

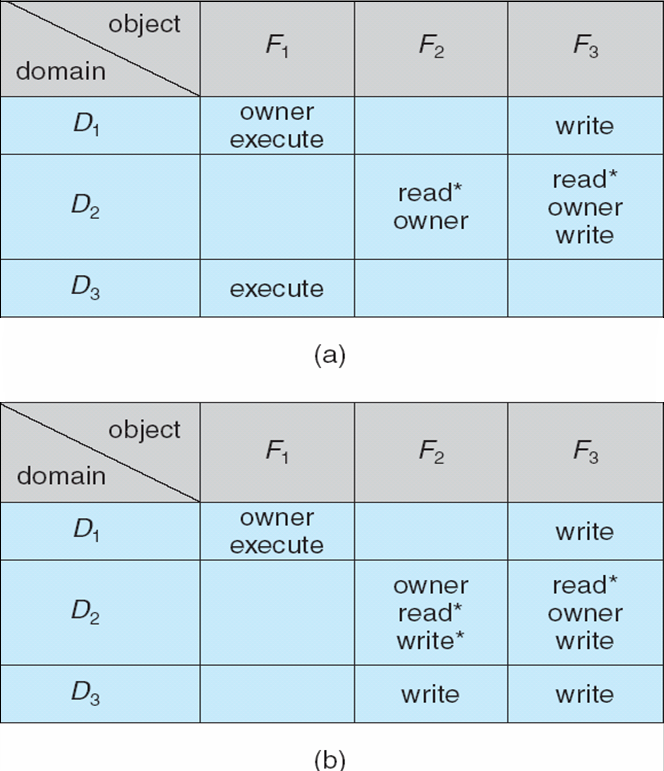
A right is copied from **access** (i, j) to **access**(k, j); it is then removed from **access** (i,j); this action is a *transfer* of a right, rather than a copy.

1. Propagation of the *copy* right may be limited. That is, when the right *R\** is copied from **access**(i, j) to **access**(k, j), only the right *R* (not R\*) is created. A process executing in domain Dk cannot further copy the right *R.*

The *copy* right allows a process to copy some rights from an entry in one column to another entry in the same column. We also need a mechanism to allow addition of new rights and removal of some rights.

The *owner* right controls these operations. If access(i, j) includes the *owner* right, then a process executing in domain Di can add and remove any right in any entry in column j.

For example, in the above Figure (a), domain D1 is the owner of F1, and thus can add and delete any valid right in column F1. Similarly, domain *D2*is the owner of *F2*and F3, and thus can add and remove any valid right within these two columns. Thus, the access matrix of the above Figure (a) can be modified to the access matrix shown in the above Figure (b)

****

The *copy* and *owner* rights allow a process to change the entries in a column. A mechanism is also needed to change the entries in a row. The ***control***right is applicable to only domain objects. If access(i, j) includes control right then a process executing in domain Di can remove any access right from row j.

****

For example, suppose that, in Figure 14.4, we include the control right in access (*D*2, *D*4). Then, a process executing in domain *D*2 could modify domain *D*4, as shown in Figure 14.7.

**Implementation of Access Matrix**

The various schemes used for the implementation of access matrix are

1. **Global Table**

The simplest implementation of the access matrix is a global table consisting of a set of ordered triples <***domain, object, rights-set***>. Whenever an operation M is executed on an object Oj within domain Di, the global table is searched for a triple <Di, Oj, Rk*>,* with M ∈Rk*.* If this triple is found, the operation is allowed to continue; otherwise, an exception (error) condition is raised.

Virtual memory techniques are often used for managing this table.

**Drawbacks:**

The table need additional I/O as it is large and cannot be kept in main memory.

1. **Access Lists for Objects**

Each *column* in the access matrix can be implemented as an access list for one object. The empty entries can be discarded in the list. The resulting list for each object consists of ordered pairs <***domain, rights-set***>, which define all domains with a nonempty set of access rights for that object.

Each column = Access-control list for one object   
Defines who can perform what operation.  
  
 Domain 1 = Read, Write  
 Domain 2 = Read  
 Domain 3 = Read

1. **Capability Lists for Domains**

In this method, each row in the access matrix is implemented as a capability for every domain. A *capability list* for a domain is a list of objects together with the operations allowed on those objects. An object is represented by its *physical name* or *address*, called a *capability.* To execute operation M on object ***Oj,***the process executes the operation M, specifying the capability (pointer) for object ***Oj***as a parameter.

Each Row = Capability List (like a key)  
For each domain, what operations allowed on what objects.

Object 1 – Read

Object 4 – Read, Write, Execute

Object 5 – Read, Write, Delete, Copy

1. **A Lock-Key Mechanism**

The *lock-key scheme* is an intermediate method between access lists and capability lists. Each object has a list of unique bit patterns, called *locks.* Similarly, each domain has a list of unique bit patterns, called *keys.* Aprocess executing in a domain can access an object only if that domain has a key that matches one of the locks of the object.

In this strategy the list of keys or locks in a particular domain must be managed by the Operating system and accessed by the user indirectly. Users are not allowed to examine or modify the list of keys (or locks) directly.

**Revocation of Access Rights**

In a dynamic protection system, it may sometimes be necessary to revoke access rights to objects that are shared by different users.

Various questions about revocation may arise:

* **Immediate versus delayed**: Does revocation occur immediately, or is it delayed? If revocation is delayed, can we find out when it will take place?
* **Selective versus general**: When an access right to an object is revoked, does it affect allthe users who have an access right to that object, or can we specify a select group of users whose access rights should be revoked?
* **Partial versus total**: Can a subset of the rights associated with an object be revoked, or must we revoke all access rights for this object?
* **Temporary versus permanent**: Can access be revoked permanently (that is, the revoked access right will never again be available), or can access be revoked and later be obtained again?

With an access-list scheme, revocation is easy. The access list is searched for the access right(s) to be revoked, and they are deleted from the list. Revocation is immediate, and can be general or selective, total or partial, and permanent or temporary.

With Capabilities scheme, revocation is more difficult. Since the capabilities are distributed throughout the system. There are several different schemes for implementing revocation for capabilities, including the following:

* **Reacquisition:** Periodically, capabilities are deleted from each domain. If a process wants to use a capability, it may find that that capability has been deleted. The process may then try to reacquire the capability. If access has been revoked, the process will not be able to reacquire the capability.
* **Back-pointers:**  A list of pointers is maintained with each object, pointing to all capabilities associated with that object. When revocation is required, we can follow these pointers, changing the capabilities as necessary.
* **Indirection:** The capabilities do not point to the objects directly, but instead point indirectly. Each capability points to a unique entry in a global table, which in turn points to the object. We implement revocation by searching the global table for the desired entry and deleting it.
* **Keys:**  A key is a unique bit pattern that can be associated with each capability. This key is defined when the capability is created, and it can be neither modified nor inspected by the process owning that capability.

A *master key* associated with each object can be defined or replaced with the set-key operation. When a capability is created, the current value of the master key is associated with the capability. When the capability is exercised, its key is compared to the master key. If the keys match, the operation is allowed to continue; otherwise, an exception condition is raised. Revocation replaces the master key with a new value by the **set-key** operation, invalidating all previous capabilities for this object.

**Security**

Computer resources must be guarded against unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency. These resources include information stored in the system (both data and code), as well as the CPU, memory, disks, tapes and networking that are the computer.

Security violations (or misuse) of the system can be categorized as intentional (malicious) or accidental. It is easier to protect against accidental misuse than against malicious misuse.

In addition, a ***threat***is the potential for a security violation, such as the discovery of error, whereas an ***attack***is the attempt to break security.

The following are some of the security violation threats:

1. ***Breach of confidentiality***is the security violation which involves unauthorized reading of data (or theft of information).
2. ***Breach of integrity***involves unauthorized modification of data.
3. ***Breach of availability***involves unauthorized destruction of data.
4. ***Theft of service***involves unauthorized use of resources.
5. ***Denial of service*** involves preventing legitimate use of the system.

To protect a system, we must take security measures at two levels:

* **Physical level:** The site or sites containing computer systems must be secured physically against armed or secret entry by intruders. Both the machine rooms and the terminals or workstations that have access to the machines must be secured.
* **Human level:** Authorizing users must be done carefully to assure that only appropriate users have access to the system. Even authorized users, however, may be "encouraged" to let others use their access (in exchange for a bribe, for example). They may also be tricked into allowing access via ***social engineering***.One type of social-engineering attack is ***phishing***.Here, a legitimate-looking e-mail or web page misleads a user into entering confidential information.

**User Authentication**

**Passwords**

The most common approach to authenticating a user identity is the use of ***passwords***.When the user identifies themselves by user ID or account name, they are asked for a password. If the user-supplied password matches the password stored in the system, they are authenticated. Passwords are often used to protect objects in the computer system, in the absence of more complete protection schemes For instance, a password could be associated with each resource (such as a file). Whenever a request is made to use the resource, the password must be given. If the password is correct, access is granted.

**Password Vulnerabilities**

Passwords are common because they are easy to understand and use. Unfortunately, passwords can be guessed, accidentally exposed, sniffed, or illegally transferred from an authorized user to an unauthorized one.

There are two common ways to guess a password. One way is for the intruder (either human or program) to know the user or to have information about the user. All people use frequently obvious information (such as the names of their cats or spouses) as their passwords. The other way is to use brute force, trying enumeration—or all possible combinations of valid password characters (letters, numbers, and punctuation on some systems)—until the password is found. Short passwords are especially vulnerable to this method.

For example, a four-decimal password provides only 10,000 variations. On average, guessing 5,000 times would produce a correct hit. A program that could try a password every millisecond would take only about 5 seconds to guess a four-digit password. Enumeration is less successful where systems allow longer passwords that include both uppercase and lowercase letters, along with numbers and all punctuation characters

In addition to being guessed, passwords can be exposed as a result of visual or electronic monitoring. An intruder can look over the shoulder of a user **(*shoulder surfing*)** when the user is logging in and can learn the password easily by watching the keyboard.

Passwords can be either generated by the system or selected by a user. System-generated passwords may be difficult to remember, and thus users may write them down.

Several alternatives on these simple password schemes can be used. For example, the password can be changed more frequently. In the extreme, the password is changed from session to session. A new password is selected (either by the system or by the user) at the end of *each* session, and that password must be used for the next session. In such a case, even if a password is misused, it can be used only once. When the legitimate user tries to use a now-invalid password at the next session, he discovers the security violation

**Encrypted Passwords**

There is a one problem with all the approaches of generating password is the difficulty of keeping the password secret within the computer. The UNIX system uses encryption to avoid the necessity of keeping its password list secret. Each user has a password. The system contains a function that is extremely difficult—the designers hope impossible—to invert but is simple to compute. That is, given a value *x,* it is easy to compute the function value *f(x).* Given a function value *f(x),* however, it is impossible to compute *x*. This function is used to encode all passwords. Only encoded passwords are stored. When a user presents a password, it is encoded and compared against the stored encoded password. Even if the stored encoded password is seen, it cannot be decoded, so the password cannot be determined. Thus, the password file does not need to be kept secret. The function*f(x* is typically an encryption algorithm

**One-Time Passwords**

To avoid the problems of password sniffing and shoulder surfing, a system could use a set of ***paired passwords***.When a session begins, the system randomly selects and presents one part of a password pair; the user must supply the other part. In this system, the user is *challenged*and must *respond* with the correct answer to that challenge. This approach can be generalized to the use of an algorithm as a password.

The algorithm might be an integer function, for example. The system selects a random integer and presents it to the user. The user applies the function and replies with the correct result. The system also applies the function. If the two results match, access is allowed.

**PROGRAM THREATS**

In an environment where a program written by one user may be used by another user, there is an opportunity for misuse, which result in unexpected behavior

**Trojan horse**

A Trojan Horse is a ordinary program which looks like a safe program to the user but opens a 'trap door' which is used to transfer user data to another system or cause destruction to user data without the knowledge of the user.

Trojan Horses are actually normal computer programs. A Trojan Horse pretends to be a regular downloadable program, such as a game, but actually does something different—most likely does damage to a computer, like erasing a disk. This is different from a regular virus, because it doesn’t attempt to reproduce itself. Trojan Horses do not affect a great number of people because they are discovered quickly.

The Trojan Horse, at first glance will appear to be useful [software](http://www.webopedia.com/TERM/S/software.html) but will actually do damage once installed or run on your computer.  When a Trojan is activated on your computer, the results can vary. Some Trojans are designed to be more annoying than malicious (like changing your[desktop](http://www.webopedia.com/TERM/D/desktop.html), adding silly active desktop icons) or they can cause serious damage by deleting files and destroying information on your system.

**Trap Door:**

Trap door is a type of security breach where the designer of a program or a system leaves a hole in the software that only he is capable of using.

A Trap door is a Secret entry point in to a program that allows someone to gain access without normal methods to access authentication**.**

Trap doors, also referred to as **backdoors**, are bits of code embedded in programs by the programmer(s) to quickly gain access at a later time, often during the testing or debugging phase

If an unscrupulous programmer purposely leaves this code in or simply forgets to remove it, a potential security hole is introduced. Hackers often plant a backdoor on previously compromised systems to gain later access. Trap doors can be almost impossible to remove in a reliable manner. Often, reformatting the system is the only sure way.

**System Threats:**

Most Operating systems provide a means for processes to spawn other processes. In such an environment, it is possible to create a situation where operating system resources and user files are misused. The two most common methods for achieving this misuse are worms and viruses.

**Worms:**

A Worm is a self propagating program which does not cause any destruction to the user data. Instead, the worm tries to unearth the security problems and tries to cause unwanted delay in processing of the CPU.

It spread from computer to computer, but unlike a virus, it has the capability to travel without any human action. It takes advantage of file or information transport features on your system, which is what allows it to travel unaided.

The biggest danger with a worm is its capability to replicate itself on our system, so rather than our computer sending out a single worm, it could send out hundreds or thousands of copies of itself, creating a huge devastating effect. One example would be for a worm to send a copy of itself to everyone listed in your e-mail address book. Then, the worm replicates and sends itself out to everyone listed in each of the receiver's address book, and the manifest continues on down the line.

Due to the copying nature of a worm and its capability to travel across networks the end result in most cases is that the worm consumes too much system memory (or [network](http://www.webopedia.com/TERM/N/network.html) bandwidth), causing Web [servers](http://www.webopedia.com/TERM/S/server.html), network servers and individual computers to stop responding.

**Virus**

VIRUS stands for ***V****irtual* ***I****nformation* ***R****esource* ***U****nder* ***S****eize*. It is a computer program that spreads itself (infects or replicates) and interferes with the normal functioning of a computer system. It may "corrupting" user data or formatting hard disk, rebooting system all of a sudden or does something else with which a computer user is affected.

It spreads itself infecting every user program there by destroying one of more user programs on the system. A virus attaches its signature to a program file. The virus gets into the memory of the system when the user executes by running the program. Normally, deletion of the infected program file cannot solve this problem.

Viruses are programs inserted into another programme. They get activated by its host programme. They replicate themselves and spread to others through floppy transfer. It infects data or programme every time the user runs the infected programme and the virus takes advantages and replicates itself.

**Symptoms of a computer virus**

* The computer runs slower than usual.
* The computer stops responding, or it locks up frequently.
* The computer crashes, and then it restarts every few minutes.
* The computer restarts on its own and does not run as usual.
* Applications on the computer do not work correctly.
* Disks or disk drives are inaccessible.
* You cannot print items correctly.
* You see unusual error messages.
* You see distorted menus and dialog boxes.
* There is a double extension on an attachment that you recently opened, such as a .jpg, .vbs, .gif, or .exe extension.
* An antivirus program is disabled for no reason and it cannot be restarted.
* An antivirus program cannot be installed on the computer, or it will not run.
* New icons appear on the desktop that you did not put there, or the icons are not associated with any recently installed programs.
* Strange sounds or music plays from the speakers unexpectedly.

**FAULT TOLERANCE**

**INTRODUCTION**

To avoid disruptions due to failures and to improve availability, systems are designed to be fault-tolerant. System can be designed to be fault-tolerant in two ways. A system may mask failures or exhibit a well definedfailure behavior in the event of failure.

When a system is designed to ***mask failures***, it continues to perform its specified function in the event of a failure. A system designed for ***well defined*** behavior may or may not perform the specified function in the event of a failure; however, it can facilitate actions suitable for recovery.

An example of well defined behavior during a failure is: the changes made to a database by a transaction are made visible to other transactions only if the transaction successfully commits; if the transaction fails, the changes made to the database by the failed transaction are not made visible to the other transactions, thus not affecting those transactions.

One key approach used to tolerate failures is redundancy. In this approach, a system employs a multiple number of processes, a multiple number of hardware components, multiple copies of data, etc., each with independent failure modes.

We used techniques, such as commit protocols and voting protocols in the design of fault-tolerant systems.

**ISSUES**

Since a fault-tolerant system must behave in a specified manner in the event of a failure, it is important to study the implications of certain types of failures.

**Process Deaths**: When a process dies, it is important that the resources allocated to that process are recover, otherwise they may be permanently lost. Many distributed systems are structured along the client-server model in which a client requests a service by sending a message to a server. If the server process fails, it is necessary that the client machine be informed so that the client process waiting for a reply can be unblocked to take suitable action. Likewise, if a client process dies after sending a request to a server, it is necessary that the server be informed that the client process no longer exists. This will facilitate the server in reclaiming any resources it has allocated to the client process.

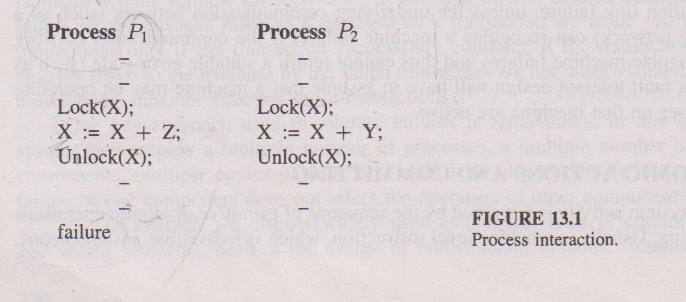
**Machine Failure:** In the case of machine failure, all the processes running at the machine will as far as the behavior of a client process or a server process is concerned, there is not much difference in their behavior in the event of a machine failure or a process death. The only difference lies in how the failure is detected. In the case of a process death, other processes including the kernel remain active. Hence, a message stating that the process has died can be sent to an inquiring process. On the other hand, an absence of any kind of message indicates either process death or a failure due to machine failure.

**Network Failure:** A communication link failure can partition a network into subnets, making it impossible for a machine to communicate with another machine in a different subnet. A process cannot tell the difference between a machine and a communication link failure, unless the underlying communication network can recognize a machine failure. If the communication network cannot recognize machine failures and thus cannot return a suitable error code (such as Ethernet), a fault-tolerant design will have to assume that a machine may be operating and processes on that machine are active.

**ATOMIC ACTIONS AND COMMITTING**

Suppose two processes P1 and P2 share a memory location X and both modify X as shown in the below Figure, Suppose P1 succeeds in locking X before P2, then P1 updates X and releases the lock, making it possible for P2 to access X. If P1 fails after P2 has seen the changes made to X by P1, then P2 will also have to be aborted or rolled back. Thus, what is necessary is that P2 should not be able to interact with P1 through X until it can do safely. In other words, P1 should be atomic. Its effect on X should not be visible to P2 or any other process until P1 is guaranteed to finish. The effect of P1 on the system (even though it executes concurrently with F2) should look like an undivided and uninterrupted operation.

**Process Interaction**



Atomic actions are the basic building blocks in constructing fault- tolerant operations. They provide a means to a system designer to specify the process interactions that are to be prevented to maintain the integrity of the system. Atomic actions have the following characteristics.

* An action is atomic if the process performing it is not aware of the existence of any other active processes, and no other process is aware of the activity of the process during the time the process performs the action.
* An action is atomic if the process performing it does not communicate with other processes while the action is being performed.
* An action is atomic if the process performing it can detect no state changes except those performed by itself, and if it does not reveal its state changes until the action is complete.
* Actions are atomic if they can be considered, other processes to be indivisible and instantaneous, such that the effects on the system are as if they were interleaved as opposed to concurrent.

A transaction groups a sequence of actions (for example, on a database) and the group is treated as an atomic action to maintain the consistency of a database. At some point during itsexecution, the transaction decides whether to commit or abort its actions. A ***commit*** is an unconditional guarantee that the transaction will be completed. In other words, the effects of its actions on the database will be permanent. An ***abort*** is an unconditional guarantee to back out of the transaction, and none of the effects of its actions will persist.

A transaction may abort due to any of the following events: deadlocks, timeouts, protection violation, wrong input provided by user, or consistency violations to facilitate backing out of an aborting transaction, the write-ahead-log protocol or shadow pages can be employed.

In distributed systems, several processes may coordinate to perform a task. Their actions may have to be atomic with respect to other processes. For example, transaction may spawn many processes that are executed at different sites. As another example, in distributed database systems, a transaction must be processed at every site or at none of the sites to maintain the integrity of the database. This is referred to as global atomicity. The protocols that enforce global atomicity are referred to as commit protocols.

**COMMIT PROTOCOLS**

The following situation illustrates the difficulties that arise in the design of commit protocols.

**The Generals Paradox**.

There are two generals of the same army who have encamped a short distance apart. Their objective is to capture a hill, which is possible only if they attack simultaneously. If only one general attacks, he will be defeated. The two generals can communicate only by sending messengers. There is a chance that these messengers might lose their way or be captured by the enemy. The challenge is to use a protocol that allows the generals to agree on a time to attack, even though some messengers do not get through.

The goal of commit protocols is to have all the sites (generals) agree either to commit (attack) or to abort (do not attack) a transaction. By relaxing the requirement that the number of messages employed by a commit protocol be bounded by a fixed number of messages, a commit protocol can be designed.

**The Two-Phase Commit Protocol**

This protocol assumes that one of the cooperating processes acts as a ***coordinator***. Other processes are referred to as ***cohorts***. (Cohorts are assumed to be executing at different sites.) .This protocol assumes that a stable storage is available at each site and the write- ahead log protocol is active. At the beginning of the transaction, the coordinator sends a start transaction message to every cohort.

**Phase I.**

**At the coordinator**:

* The coordinator sends a COMMIT-REQUEST message to every cohort requesting the cohorts to commit.
* The coordinator waits for replies from all the cohorts.

**At cohorts:**

* On receiving the COMMIT-REQUEST message, a cohort takes the following actions. If the transaction executing at the cohort is successful, it writes UNDO and REDO log on the stable storage and sends an AGREED message to the coordinator. Otherwise, it sends an ABORT message to the coordinator.

**Phase II.**

**At the coordinator:**

* If all the cohorts reply AGREED and the coordinator also agrees, then the coordinator writes a COMMIT record into the log. Then it sends a COMMIT message to all the cohorts. Otherwise, the coordinator sends an ABORT message to all the cohorts.
* The coordinator then waits for acknowledgments from each cohort.
* If an acknowledgment is not received from any cohort within a timeout period, the coordinator resends the commit/abort message to that cohort.
* If all the acknowledgments are received, the coordinator writes a COMPLETE record to the log (to indicate the completion of the transaction.)

**At cohorts:**

* On receiving a COMMIT message, a cohort releases all the resources and locks held by it for executing the transaction, and sends an acknowledgment.
* On receiving an ABORT message, a cohort undoes the transaction using the UNDO log record, releases all the resources and locks held by it for performing the transaction, and sends an acknowledgment.

When there are no failures or message losses, it is easy to see that all sites will commit only when all the participants (including the coordinator) agree to commit. In the case of lost messages (sent from either cohorts or the coordinator), the coordinator simply resends messages after the timeout.

**SITE FAILURES**

For site failures, we look at the following cases:

* Suppose the coordinator crashes before having written the COMMIT record. On recovery, the coordinator broadcasts an ABORT message to all the cohorts. All the cohorts who had agreed to commit will simply undo the transaction using the UNDO log and abort. Other cohorts will simply abort the transaction. Note that all the cohorts are blocked until they receive an ABORT message.
* Suppose the coordinator crashes after writing the COMMIT record but before writing the COMPLETE record. On recovery, the coordinator broadcasts a COMMIT message to all the cohorts and waits for acknowledgments. In this case also the cohorts are blocked until they receive a COMMIT message.
* Suppose the coordinator crashes after writing the COMPLETE record. On recovery, there is nothing to be done for the transaction.
* If a cohort crashes in Phase I, the coordinator can abort the transaction because it did not receive a reply from the crashed cohort.
* Suppose a cohort crashes in Phase II, that is, after writing its UNDO and REDO log. On recovery, the cohort will check with the coordinator whether to abort (i.e., perform an undo operation) or to commit the transaction.

**VOTING PROTOCOLS**

* Drawback: blocking protocol. Cohorts blocked if coordinator fails.
* Resources and locks held unnecessarily.
* Conditions that cause blocking:
* Assume that only one site is operational. This site cannot decide to abort a transaction as some other site may be in commit state.
* It cannot commit as some other site can be in abort state.
* Hence, the site is blocked until all failed sites recover.

Alternative to the commit protocol is voting protocols.

**Basic idea of voting protocol:**

* Each replica assigned some number of votes
* A majority of votes need to be collected before accessing a replica.
* Voting mechanism: more fault tolerant to site failures, network partitions, and message losses.

**Types of voting schemes:**

* Static
* Dynamic

**Static Voting Scheme**

**System Model:**

* File replicas are kept at different sites. File lock rule: either ***one writer + no reader*** or ***multiple readers + no writer***.
* Every file is associated with a version number that gives the number of times a file has been updated.
* Version numbers are stored on stable storage. Every successful write updates version number.

**Basic Idea:**

* Every replica assigned a certain number of votes. This number stored on stable storage.
* A read or write operation permitted if a certain number of votes, called *read quorum* or *write quorum*, are collected by the requesting process.

**Voting Algorithm:**

* Let a site i issue a read or write request for a file.
* Site i issues a Lock Request to its local lock manager.
* When lock request is granted, i sends a Vote\_Request message to all the sites.
* When a site j receives a Vote\_Request message, it issues a Lock\_Request to its lock manager. If the lock request is granted, then it returns the version number of the replica (VNj) and the number of votes assigned to the replica (Vj) at site i.
* Site i decides whether it has the quorum or not, based on replies received within a timeout period as follows.
* For read requests, Vr(vote for reading) = Sum of Vk, k in P, where P is the set of sites from which replies were received.
* For write requests, Vw = Sum of Vk, k in Q such that:

M = max{VN j: j is in P}(replica having maximum version number)

Q = {j in P : VNj = M}

Only the votes of the current (version) replicas are counted in deciding the write quorum.

P= {S1, S2, S3, S4, S5, S6}

Version number (VNj) where j=1 to 6(J represents sites)

VN1=10(replica version number at site 1),VN2=12,VN3=10,VN4=16,VN5=16

VN6=16, M=Max (VNJ)={VN3,VN4,VN5},

Q= (sites with max VNJ)={S4,S5,S6}

If only one site is participating in voting write quorum is not applied. Since the rule of voting protocol is to get maximum number of votes from all the sites.

* If i is not successful in getting the quorum, it issues a Release \_Lock to the lock manager & to all sites that gave their votes.
* If i is successful in collecting the quorum, it checks whether its copy of file is current (VN**i**= M). If not, it obtains the current copy.
* If the request is read, i reads the local copy. If write, i updates the local copy and VN.
* i sends all updates and VN**i** to all sites in Q, i.e., update only current replicas. i send Release\_Lock request to its lock manager as well as those in P.
* All sites on receiving updates, perform updates. On receiving Release\_Lock, releases lock.

**Vote Assignment:**

* Let v be the total number of votes assigned to all copies. Read & write quorum, r & w, are selected such that: r + w > v; w > v/2.
* Above values are determined so that there is a non-null intersection between every read and write quorum, i.e., at least 1 current copy in any reading quorum gathered.
* Write quorum is high enough to disallow simultaneous writes on 2 distinct subset of replicas.
* The scheme can be modified to collect write quorums from non-current replicas. Another modification: obsolete replicas updated.
* (e.g.,) System with 4 replicas at 4 sites. Votes assigned: V1 = 1, V2 = 1, V3 = 2, & V4 = 1.
* Let disk latency at S1 = 75 msec, S2 = 750 msec, S3 = 750 msec. & S4 = 100 msec.
* If r = 1 & w = 5, and read access time is 75 ms and write access is 750 msec.

**LOCKING PROTOCOL**

One way to ensure serializability is to associate with each data item a lock to require that each transaction follow a ***locking protocol***that governs how locksare acquired and released. There are various modes in which a data item can belocked. There are two modes.

* Shared: If a transaction Ti has obtained a shared-mode lock (denoted by S) on data item Q, then Ti can read this item, but it cannot write Q*.*
* Exclusive: If a transaction Ti has obtained an exclusive-mode lock (denoted by X) on data item Q, then Ti can both read and write Q.

We require that every transaction request a lock in an appropriate mode on dataitem Q, depending on the type of operations it will perform on Q.

To access a data item Q, transaction Ti must first lock Q in the appropriatemode. If Q is not currently locked, then the lock is granted, and Ti can nowaccess it. However, if the data item Q is currently locked by some other transaction, then Ti may have to wait. Suppose that Tirequests an exclusive lock on Q. In this case, Timust wait until the lock on Q isreleased.

If Ti requests a shared lock on Q, then Ti must wait if Q is locked inexclusive mode. Otherwise, it can obtain the lock and access Q.

A transaction may unlock a data item that it had locked at an earlier point.It must, however, hold a lock on a data item as long as it accesses that item.Moreover, it is not always desirable for a transaction to unlock a data itemimmediately after its last access of that data item, because serializability maynot be ensured.

One protocol that ensures serializability is the ***two-phase locking protocol****.*This protocol requires that each transaction issue lock and unlock requests intwo phases:

* Growing phase: A transaction may obtain locks, but may not release any lock.
* Shrinking phase: A transaction may release locks, but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks asneeded. Once the transaction releases a lock, it enters the shrinking phase, andno more lock requests can be issued.

The two-phase locking protocol ensures conflict serializability. It does not ensure deadlock free transaction.

To improve the performance of two-phase locking protocol, it needs either additional information about the transactions or to impose some structure or ordering on the set of data.

**Timestamp-Based Protocols**

In the locking protocols, the order between every pair ofconflicting transactions is determined at execution time by the first lock thatthey both request and involves incompatible modes. Another method fordetermining the serializability order is to select an ordering among transactionsin advance. The most common method is to use a *timestamp ordering*scheme.

With each transaction Ti in the system, we associate a unique fixed timestamp,denoted by TS(Ti). This timestamp is assigned by the system before the transaction Ti starts execution. If a transaction Ti has been assigned timestampTS(Ti), and later on a new transaction Tj enters the system, then TS(Ti) < TS(Tj).There are two methods for implementing this scheme:

* Use the value of the system clock as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system. This method will not work for transactions that occur on separate systems or for processors that do not share a clock.
* Use a logical counter as the timestamp; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system.The counter is incremented after a new timestamp is assigned.

The timestamps of the transactions determine the serializability order.Thus, if TS(Ti) < TS(Tj), then the system must ensure that the produced scheduleis equivalent to a serial schedule in which transaction Ti appears beforetransaction *Tj.*

To implement this scheme, we associate with each data item Q two timestampvalues:

* **W-timestamp**(Q), which denotes the largest timestamp of any transaction that executed **write**(Q) successfully
* **R-timestamp**(Q), which denotes the largest timestamp of any transaction that executed **read**(Q) successfully

These timestamps are updated whenever a new **read**(Q) or **write**(Q) instructionis executed.

The timestamp-ordering protocol ensures that any conflicting **read** and**write** operations are executed in timestamp order. This protocol operates asfollows:

* Suppose that transaction *Ti*issues **read**(Q)
  + If TS(Ti) < W-timestamp(Q), then this state implies that Ti needs to read a value of Q which was already overwritten. Hence, the **read** operationis rejected, and Ti is rolled back.
  + If TS(Ti) ≥ W-timestamp(Q), then the **read** operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(Ti).
* Suppose that transaction Ti issues write(Q):
  + If TS(Ti) < R-timestamp(Q), then this state implies that the value of Q that Ti is producing was needed previously and *Ti*assumed that this value would never be produced. Hence, the **write** operation is rejected,and *Ti*is rolled back.
  + If TS(Ti) < W-timestamp(Q), then this state implies that Ti is attempting to write an obsolete value of Q. Hence, this **write** operation is rejected,and *Ti*is rolled back.
  + Otherwise, the **write** operation is executed.

A transaction Ti, which is rolled back by the concurrency-control scheme asa result of the issuing of either a read or write operation, is assigned a newtimestamp and is restarted.

To illustrate this protocol, we consider schedule 3 of Figure 11.3 withtransactions T2 and T3. We assume that a transaction is assigned a timestampimmediately before its first instruction. Thus, in schedule 3, TS(T2) < TS(T3),and the schedule is possible under the timestamp protocol.

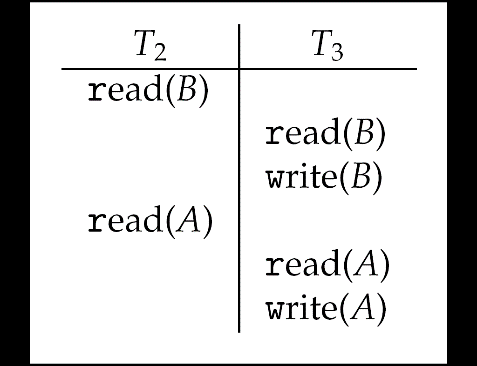


Figure 11.3 Schedule 3: A schedule possible under the timestamp protocol.

The timestamp-ordering protocol ensures conflict serializability. It ensures freedom from deadlock, because no transactionever waits.