UNIT-IV

File-System Interface: File concept – Access methods – Directory structure – File-system mounting – Protection. File-System Implementation: Directory implementation – Allocation methods – Free-space management – efficiency and performance – recovery – log-structured file systems. Case studies: File system in Linux – file system in Windows XP

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.

Examples of files:

- A text file is a sequence of characters organized into lines (and possibly pages). A source file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements. An object file is a sequence of bytes organized into blocks understandable by the system's linker. An executable file is a series of code sections that the loader can bring into memory and execute.

File Attributes

- **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 those systems that support different types.
- **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

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

File types

File type	Usual extension	Function
executable	exe, com, bin, or none	Read to run machine language program
Object	obj, o	Compiled, machine language, not linked
Source code	C, cc, java, pas ,asm ,a	Source code in various languages
Batch	bat, sh	Commands to the command interpreter
Text	txt, doc	Textual data, documents
word processor	wp, tex, rrf, doc	Various word-processor formats
Library	lib, a, so, dll, mpeg, mov, rm	Libraries of routines for programmers
print or view	arc, zip, tar	ASCII or binary file in a format for printing or

		viewing
Archive	arc, zip, tar	Related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm	Binary file containing audio or A/V information

File Structure

- All disk I/O is performed in units of one block (physical record) size which 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.
- For example, the UNIX operating system defines all files to be simply a stream of bytes. Each byte is individually addressable by its offset from the beginning (or end) of the file. In this case, the logical records are 1 byte. The file system automatically packs and unpacks bytes into physical disk blocks – say, 512 bytes per block – as necessary.
- 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.

Access Methods

1. Sequential Access

The simplest access method is sequential access. Information in the file is processed in order, one record after the other. This mode of access is by far the

most common; for example, editors and compilers usually access files in this fashion.

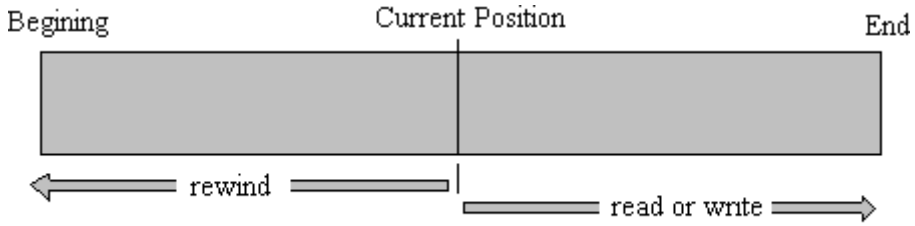


Fig 4.10 Sequential-access file

The bulk of the operations on a file is reads and writes. A read operation reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, a write appends to the end of the file and advances to the end of the newly written material (the new end of file). Such a file can be reset to the beginning and, on some systems, a program may be able to skip forward or back ward n records, for some integer n-perhaps only for n=1. Sequential access is based on a tape model of a file, and works as well on sequential-access devices as it does on random – access ones.

2. Direct Access

Another method is direct access (or relative access). A file is made up of fixed length logical records that allow programs to read and write records rapidly in no particular order. The direct- access methods is based on a disk model of a file, since disks allow random access to any file block.

For direct access, the file is viewed as a numbered sequence of blocks or records. A direct-access file allows arbitrary blocks to be read or written. Thus, we may read block 14, then read block 53, and then write block7. There are no restrictions on the order of reading or writing for a direct-access file.

Direct – access files are of great use for immediate access to large amounts of information. Database is often of this type. When a query concerning a particular subject arrives, we compute which block contains the answer, and then read that block directly to provide the desired information.

As a simple example, on an air line – reservation system, we might store all the information about a particular flight (for example, flight 713) in the block identified by the flight number.

Thus, the number of available seats for flight 713 is stored in block 713 of the reservation file. To store information about a larger set, such as people, we might compute a hash function on the people's names, or search a small in-memory index to determine a block to read and search.

Sequential access	Implementation for direct access
Reset	Cp=0;
Read next	Read cp; Cp=cp+1;
Write next	Write cp; Cp=cp+1;

3. Other Access methods

Other access methods can be built on top of a direct – access method these methods generally involve the construction of an index for the file. The index like an index in the back of a book contains pointers to the various blocks in find a record in the file. We first search the index, and then use the pointer to access the file directly and the find the desired record.

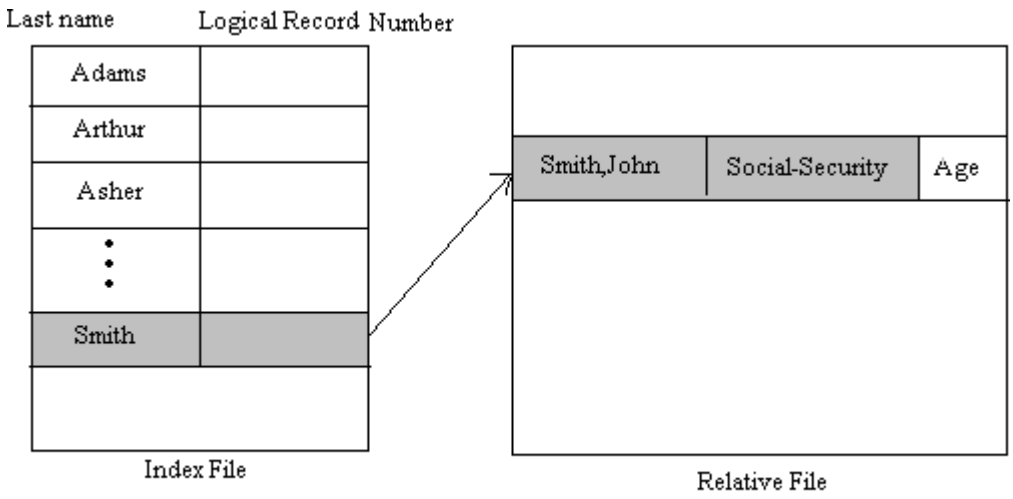


Fig 4.11 Example of Index and Relative Files

With large files, the index file itself may become too large to be kept in memory. One solution is to create an index for the index file. The primary index file would contain pointers to secondary index tiles, which would point to the actual data items.

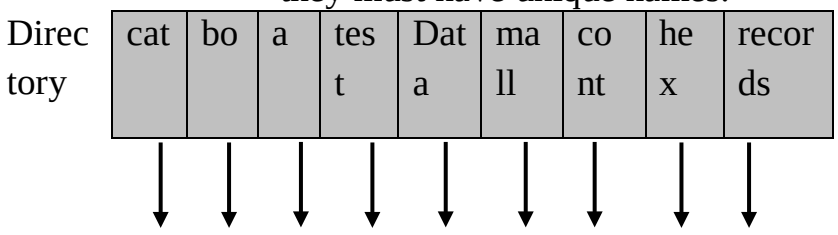
Directory Structure

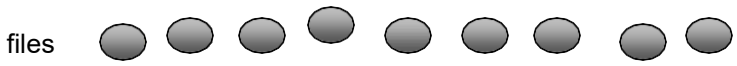
There are five directory structures. They are

1. Single-level directory
2. Two-level directory
3. Tree-Structured directory
4. Acyclic Graph directory
5. General Graph directory

1. Single – Level Directory

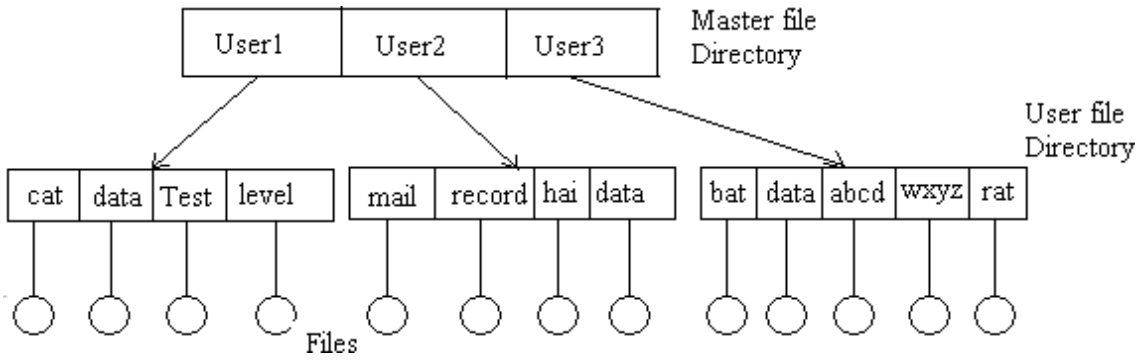
- The simplest directory structure is the single- level directory.
- All files are contained in the same directory.
- **Disadvantage:**
  - When the number of files increases or when the system has more than one user, since all files are in the same directory, they must have unique names.





2. Two – Level Directory

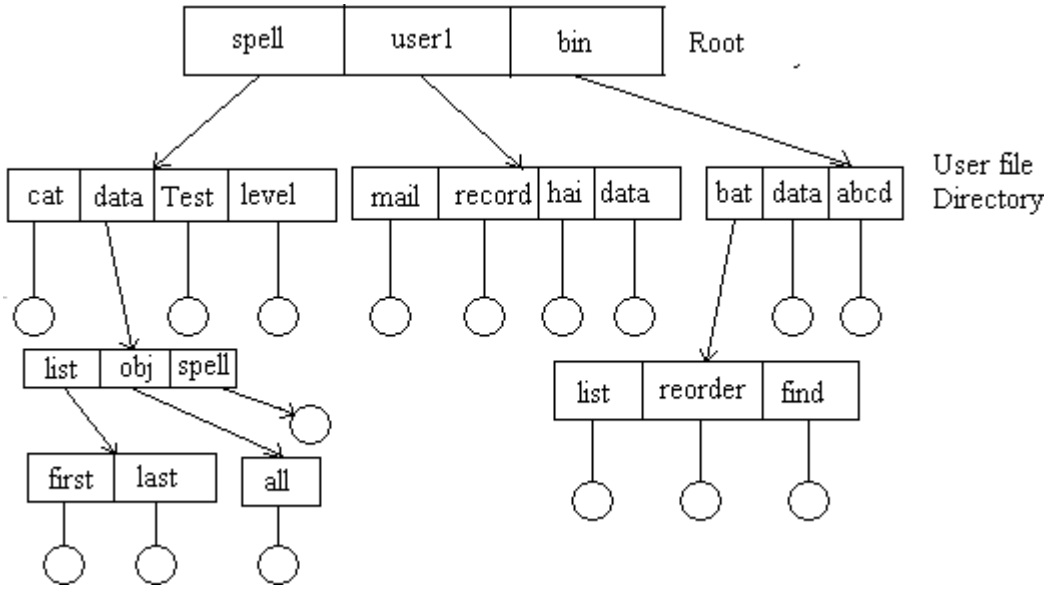
- In the two level directory structures, each user has her own user file directory (UFD).
- When a user job starts or a user logs in, the system's master file directory (MFD) is searched. The MFD is indexed by user name or account number, and each entry points to the UFD for that user.
- When a user refers to a particular file, only his own UFD is searched.
- Thus, different users may have files with the same name.
- Although the two – level directory structure solves the name-collision problem
- **Disadvantage:**
  - Users cannot create their own sub-directories.



3. Tree – Structured Directory

- A tree is the most common directory structure.
- The tree has a root directory. Every file in the system has a unique path name.
- A **path name** is the path from the root, through all the subdirectories to a specified file.
- A directory (or sub directory) contains a set of files or sub directories.
- A directory is simply another file. But it is treated in a special way.
- All directories have the same internal format.

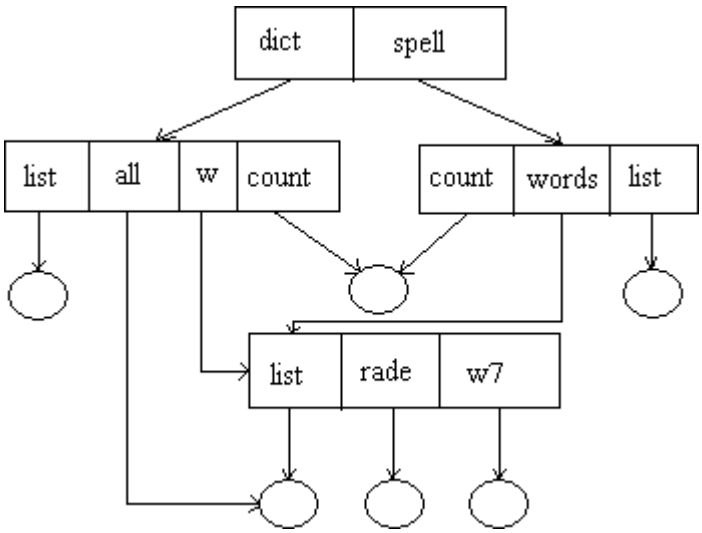
- One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1).
- Special system calls are used to create and delete directories.
- Path names can be of two types: absolute path names or relative path names.
- An absolute path name begins at the root and follows a path down to the specified file, giving the directory names on the path.
- A relative path name defines a path from the current directory.



4. Acyclic Graph Directory.

- An acyclic graph is a graph with no cycles.
- To implement shared files and subdirectories this directory structure is used.
- An acyclic – graph directory structure is more flexible than is a simple tree structure, but it is also more complex. In a system where sharing is implemented by symbolic link, this situation is somewhat easier to handle. The deletion of a link does not need to affect the original file; only the link is removed.
- Another approach to deletion is to preserve the file until all references to it are deleted. To implement this approach, we must have some

mechanism for determining that the last reference to the file has been deleted.



File System Mounting

- Just as a file must be opened before it is used, a file system must be mounted before it can be available to processes on the system.
- The mount procedure is straightforward. The operating system is given the name of the device, and the location within the file structure at which to attach the File system (or mount point).
- A mount point is an empty directory at which the mounted file system will be attached.
- For instance, on a UNIX system, a file system containing user's home directories might be mounted as /home; then to access the directory structure within that file system , one could precede the directory names with /*home, as in /home/jane.*

- Mounting that file system under */user* would result in the pathname/*users/jane*

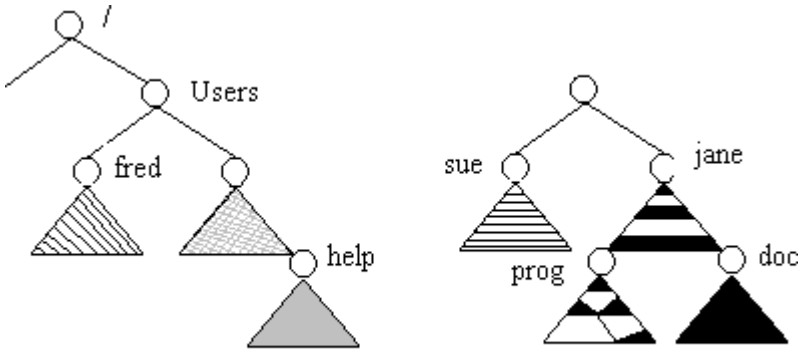


Fig 4.16 File System (a) Existing (b) Unmounted Partation

- The operating system verifies that the devices contain a valid file system.
- It does so by asking the device driver to read the device directory and verifying that the directory was the expected format.
- Finally, the operating system notes in its directory structure that a file system is mounted at the specified mount point.

File Sharing

1. Multiple Users:

- When an operating system accommodates multiple users, the issues of file sharing, file naming and file protection become preeminent.
- The system either can allow user to access the file of other users by default, or it may require that a user specifically grant access to the files.
- These are the issues of access control and protection.
- To implementing sharing and protection, the system must maintain more file and directory attributes than a on a single-user system.
    - The owner is the user who may change attributes, grand access, and has the most control over the file or directory.
    - The group attribute of a file is used to define a subset of users who may share access to the file.
- Most systems implement owner attributes by managing a list of user names and associated user identifiers (user Ids).
    - When a user logs in to the system, the authentication stage determines the appropriate user ID for the user. That user ID is associated with all

of user's processes and threads. When they need to be user readable, they are translated, back to the user name via the user name list. Likewise, group functionality can be implemented as a system wide list of group names and group identifiers.

- Every user can be in one or more groups, depending upon operating system design decisions. The user's group Ids is also included in every associated process and thread.

2. Remote File System:

- Networks allowed communications between remote computers.
- Networking allows the sharing or resource spread within a campus or even around the world.
- User manually transfer files between machines via programs like **ftp**.
- A **distributed file system** (DFS) in which remote directories is visible from the local machine.
- The **World Wide Web**: A browser is needed to gain access to the remote file and separate operations (essentially a wrapper for ftp) are used to transfer files.

a) The client-server Model:

- Remote file systems allow a computer to a mount one or more file systems from one or more remote machines.
- 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. Clients can be specified by their network name or other identifier, such as IP address, but these can be spoofed (or imitate). An unauthorized client can spoof the server into deciding that it is authorized, and the unauthorized client could be allowed access.

b) Distributed Information systems:

- Distributed information systems, also known as distributed naming service, have been devised to provide a unified access to the information needed for remote computing.
- Domain name system (DNS) provides host-name-to-network address translations for their entire Internet (including the World Wide Web).
- Before DNS was invented and became widespread, files containing the same information were sent via e-mail of ftp between all networked hosts.

c) Failure Modes:

- **Redundant arrays of inexpensive disks** (**RAID**) can prevent the loss of a disk from resulting in the loss of data.

- Remote file system has more failure modes. By nature of the complexity of networking system and the required interactions between remote machines, many more problems can interfere with the proper operation of remote file systems.

d) Consistency Semantics:

- It is characterization of the system that specifies the semantics of multiple users accessing a shared file simultaneously.
- These semantics should specify when modifications of data by one user are observable by other users.
- The semantics are typically implemented as code with the file system.
- A series of file accesses (that is reads and writes) attempted by a user to the same file is always enclosed between the open and close operations.
- The series of access between the open and close operations is a **file session**.

(i) UNIX Semantics:

The UNIX file system uses the following consistency semantics:

1. Writes to an open file by a user are visible immediately to other users that have this file open at the same time.
2. 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.

(ii) Session Semantics:

The Andrew file system (AFS) uses the following consistency semantics:

1. Writes to an open file by a user are not visible immediately to other users that have the same file open simultaneously.
2. 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 this change.

(iii) Immutable –shared File Semantics:

- Once a file is declared as shared by its creator, it cannot be modified.
- An immutable file has two key properties:
  - ✓ Its name may not be reused and its contents may not be altered.

File Protection

(i) Need for file protection.

- When information is kept in a computer system, we want to keep it safe from **physical damage** (reliability) and **improper access** (protection).
- Reliability is generally provided by duplicate copies of files. Many computers have systems programs that automatically (or though computer-operator intervention) copy disk files to tape at regular intervals (once per day or week or 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. Bugs in the file-system software can also cause file contents to be lost.
- 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 multi-user system, however, other mechanisms are needed.

(ii) Types of Access

- Complete protection is provided by prohibiting access.
- Free access is provided with no protection.
- Both approaches are too extreme for general use.
- What is needed is **controlled 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:

    1. **Read:** Read from the file.
    2. **Write:** Write or rewrite the file.
    3. **Execute:** Load the file into memory and execute it.
    4. **Append:** Write new information at the end of the file.
    5. **Delete:** Delete the file and free its space for possible reuse.
    6. **List:** List the name and attributes of the file.

(iii) Access Control

- Associate with each file and directory an access-control list (ACL) 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.

- 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 needs to 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:

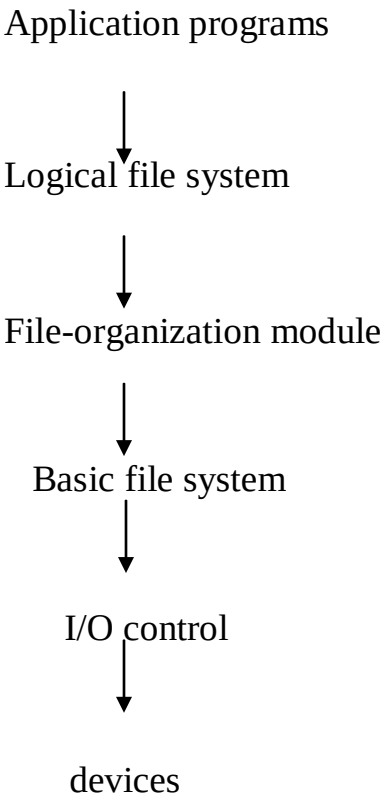
  - **Owner:** The user who created the file is the owner.
  - **Group:** A set of users who are sharing the file and need similar access is a group, or work group.
  - **Universe:** All other users in the system constitute the universe.

File System Structure

- **Disk** provide the bulk of secondary storage on which a file system is maintained.
- **Characteristics of a disk:**
  1. They can be rewritten in place, it is possible to read a block from the disk, to modify the block and to write it back into the same place.
  2. They can access directly any given block of information to the disk.
- To produce an efficient and convenient access to the disk, the operating system imposes one or more file system to allow the data to be stored, located and retrieved easily.
- The file system itself is generally composed of many different levels. Each level in the design uses the features of lower level to create new features for use by higher levels.

Layered File System

- The **I/O control** consists of device drivers and interrupt handlers to transfer information between the main memory and the disk 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 (for example, drive –1, cylinder 73, track 2, sector 10)



- The **file-organization module** knows about file 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 address to physical block addresses for the basic fie system to transfer . The file-organization module also includes the free-space manager, which tracks unallocated blocks and provides these blocks to the file-organization module when requested.
- The **logical file system** manages metadata information. Metadata includes all of the file-system structure, excluding the actual *data* (or contents of the files). The logical file system manages the directory structure to provide the file-organization module with the information the latter needs, given a symbolic file name. 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. The logical file system is also responsible for protection and security.

File System Implementation

- Several-on-disk and in-memory structures are used to implement a file system

- The on-disk structures include:

    1. A **boot control block** can contain information needed by the system to boot an operating from that partition. If the disk does not contain an operating System, this block can be empty. It is typically the first block of a partition. In **UFS**, this is called the **boot block**; In **NTFS**, it is **partition boot sector**.

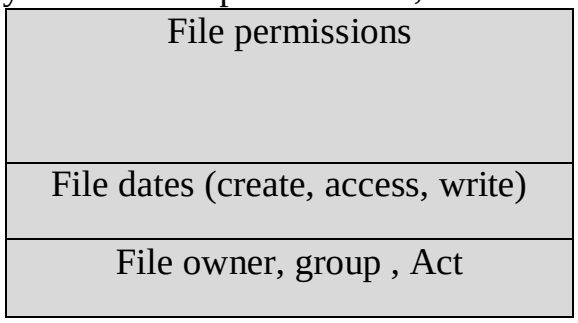
    2. A **partition control block** contains partition details such as the number of blocks in the partition, size of the blocks, free-block count and free block pointers and free FCB count and FCB pointers. In **UFS** this is called a **super block**; in **NTFS**, it is the **Master File Table**.

    3. A **directory structure** is used to organize the files.

    4. An **FCB** contains many of the files details, including file permissions , ownership, size and location of the data blocks. IN **UFS** this called the **inode**. In NTFS, this information's actually stored within the Master File Table, which uses a relational database structure, with a row per file.

- The in-memory structures include:

    1. An **in-memory partition table** containing , information about each mounted partition.
    2. An **in-memory directory structure** that hold s the directory information of recently accessed directories.
    3. The **system-wide open-file table** contains a copy of the FCB of each open files, as well as other information.
    4. The **per-process open-file table** contains a pointer tot eh appropriate entry in the systems-wide open file table, as well as other information.



File size
File data blocks

A typical file control block

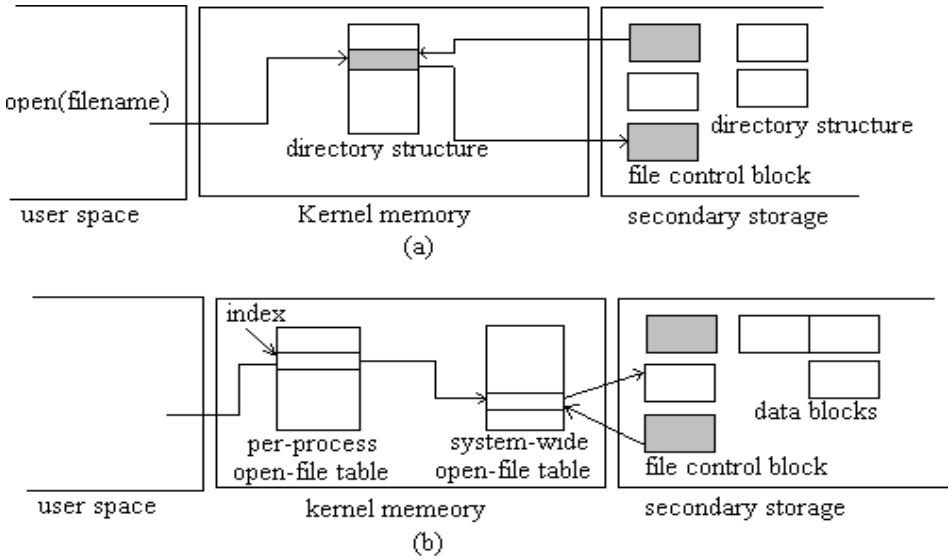


Fig 5.3 In-memory file-system structures (a) File Open (b) File Read

Directory Implementation

1. Linear List

- The simplest method of implementing a directory is to use a linear list of file names with pointer to the data blocks.
- A linear list of directory entries requires a linear search to find a particular entry.
- This method is simple to program but time- consuming to execute. To create a new file, we must first search the but time – consuming to execute.
- The real disadvantage of a linear list of directory entries is the linear search to find a file.

2. Hash Table

- In this method, a linear list stores the directory entries, but a hash data structure is also used.

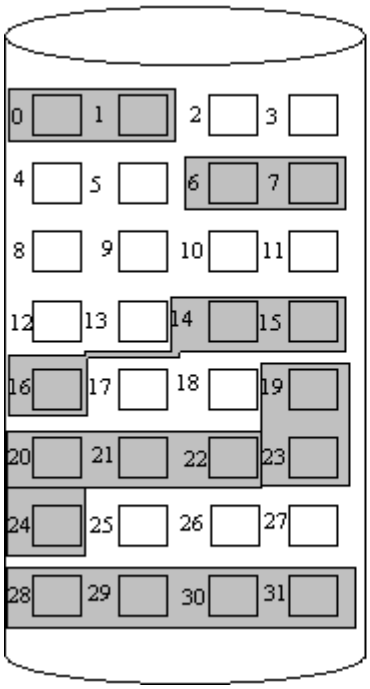
- The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list.
- Therefore, it can greatly decrease the directory search time.
- Insertion and deleting are also fairly straight forward, although some provision must be made for collisions – situation where two file names hash to the same location.
- The major difficulties with a hash table are its generally fixed size and the dependence of the hash function on that size.

Allocation Methods

- The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly .
- There are there major methods of allocating disk space:
    1. Contiguous Allocation
    2. Linked Allocation
    3. Indexed Allocation

1. Contiguous Allocation

- The contiguous – allocation method requires each file to occupy a set of contiguous blocks on the disk.



Directory

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

- 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.

Disadvantages:

1. Finding space for a new file.

- The contiguous disk space-allocation problem suffer from the problem of external fragmentation. As file are allocated and deleted, the free disk 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, no one of which is large enough to store the data.

2. 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 an allocated how does the creator know the size of the file to be created?
- If we allocate too little space to a file, we may find that file cannot be extended. The other possibility is to find a larger hole, copy the contents of the file to the new space, and release the previous space. This series of actions may be repeated as long as space exists, although it can be time –

consuming. However, in this case, the user never needs to be informed explicitly about what is happening ; the system continues despite the problem, although more and more slowly.

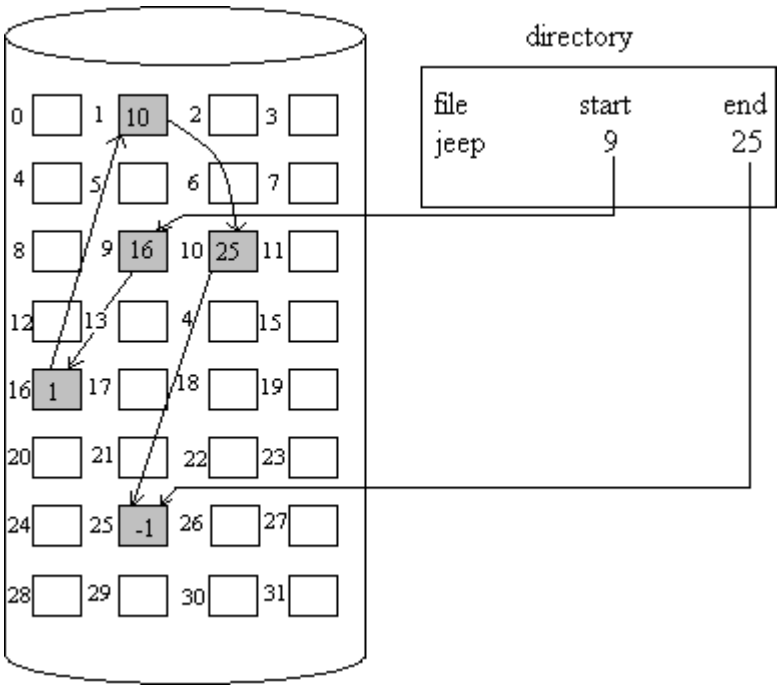
- Even if the total amount of space needed for a file is known in advance pre-allocation may be inefficient.
- A file that grows slowly over a long period (months or years) must be allocated enough space for its final size, even though much of that space may be unused for a long time the file, therefore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

To overcome these disadvantages:

- Use a modified contiguous allocation scheme, in which a contiguous chunk of space called as an **extent** is allocated initially and then, when that amount is not large enough another chunk of contiguous space an extent is added to the initial allocation.
- Internal fragmentation can still be a problem if the extents are too large, and external fragmentation can be a problem as extents of varying sizes are allocated and deallocated.

2. 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 any where 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 bock 25.
- Each block contains a pointer to the next block. These pointers are not made available to the user.
- There is no external fragmentation with linked allocation, and any free block on the free space list can be used to satisfy a request.
- The size of a file does not need to the declared when that file is created. A file can continue to grow as long as free blocks are available consequently, it is never necessary to compacts disk space.



Disadvantages:

1. Used effectively only for 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. Each aces to a pointer requires a disk read, and sometimes a disk seek consequently, it is inefficient to support a direct- access capability for linked allocation files.

2. 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.
- Solution to this problem is to collect blocks into multiples, called **clusters**, and to allocate the clusters rather than blocks. For instance, the file system may define a clusters as 4 blocks, and operate on the disk in only cluster units.

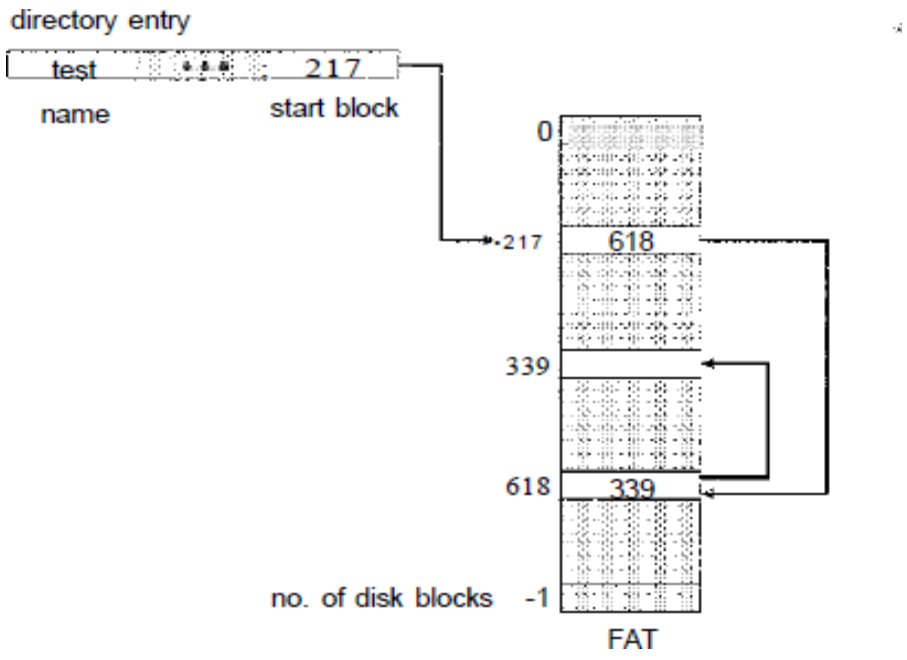
3. Reliability

- Since the files are linked together by pointers scattered all over the disk hardware failure might result in picking up the wrong pointer. This error

could result in linking into the free- space list or into another file. Partial solution are to use doubly linked lists or to store the file names in a relative block number in each block; however, these schemes require even more over head for each file.

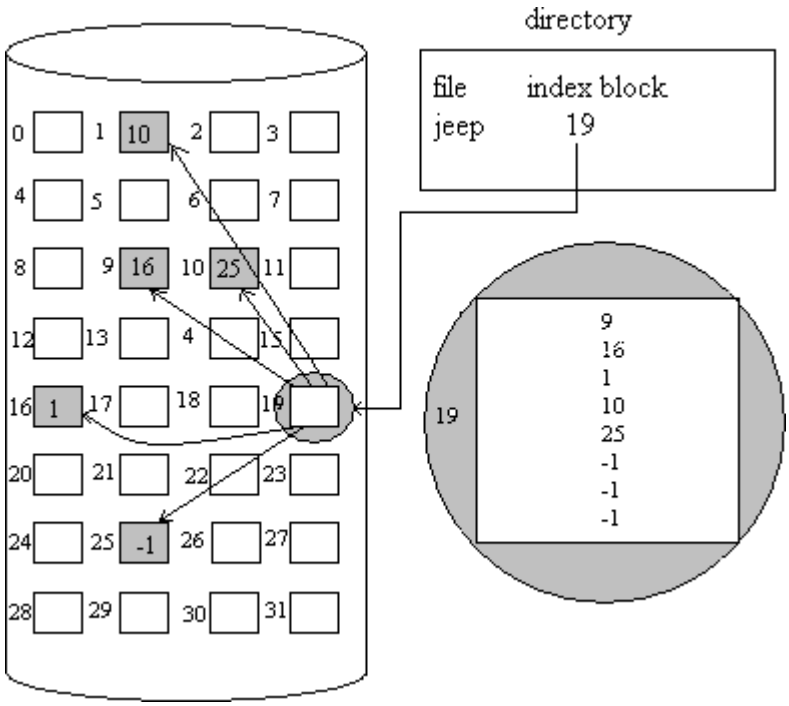
File Allocation Table(FAT)

- 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 beginning of each partition is set aside to contain the table.
- The table has entry for each disk block, and is indexed by block number.
- The FAT is much as is a linked list.
- The directory entry contains the block number 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 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 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.
- The 0 is replaced with the end – of – file value, an illustrative example is the FAT structure for a file consisting of disk blocks 217,618, and 339.



3. Indexed Allocation

- Linked allocation solves the external – fragmentation and size- declaration problems of contiguous allocation.
- Linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and need to be retrieved in order.
- 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 .
- To read the ith block, we use the pointer in the ith index – block entry to find and read the desired block this scheme is similar to the paging scheme .



- When the file is created, all pointers in the 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.

Disadvantages

1.Pointer Overhead

- Indexed allocation does suffer from wasted space. The pointer over head of the index block is generally greater than the pointer over head of linked allocation.

2. Size of Index block

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.
- **Multilevel index:** A variant of the linked representation is to use a first level index block to point to a set of second – level index blocks.
- **Combined scheme:**
    - Another alternative, used in the UFS, 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 for small (no more than 12 blocks) files do not need a separate index block
    - The next pointer is the address of a single indirect block.
        - ✓ The single indirect block is an index block, containing not data, but rather the addresses of blocks that do contain data.
    - Then there is a double indirect block pointer, which contains the address of a block that contain pointers to the actual data blocks. The last pointer would contain pointers to the actual data blocks.
    - The last pointer would contain the address of a triple indirect block.

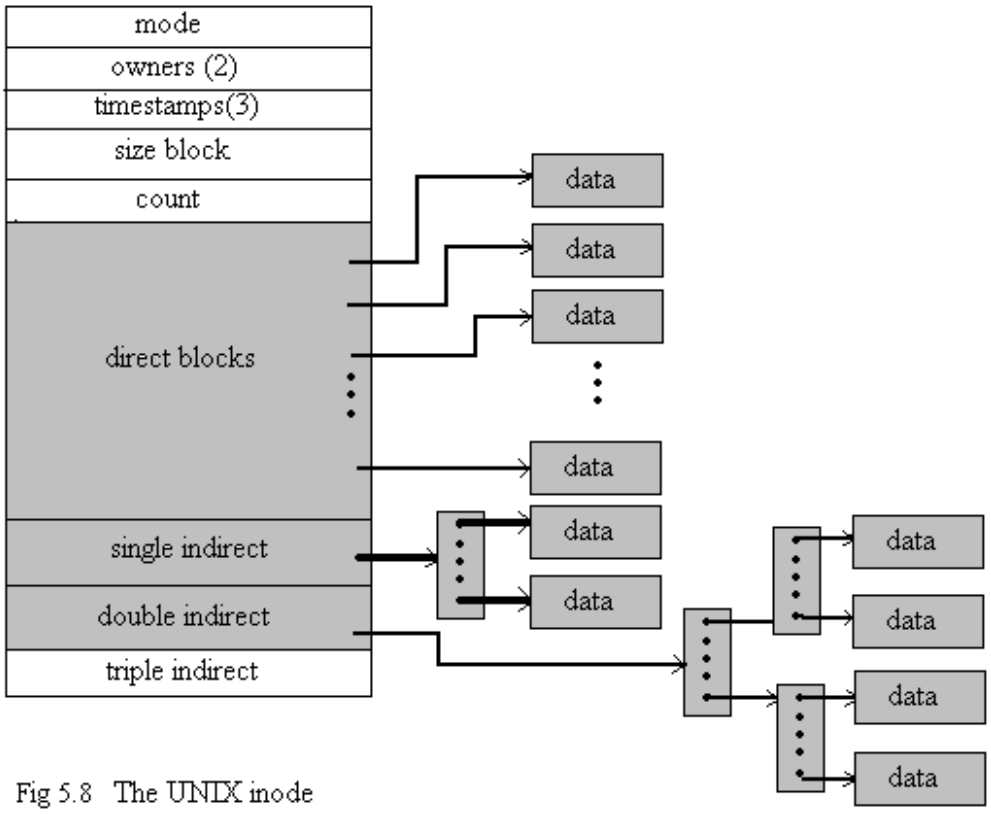


Fig 5.8 The UNIX inode

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.
- When a file is deleted, its disk space is added to the free-space list.

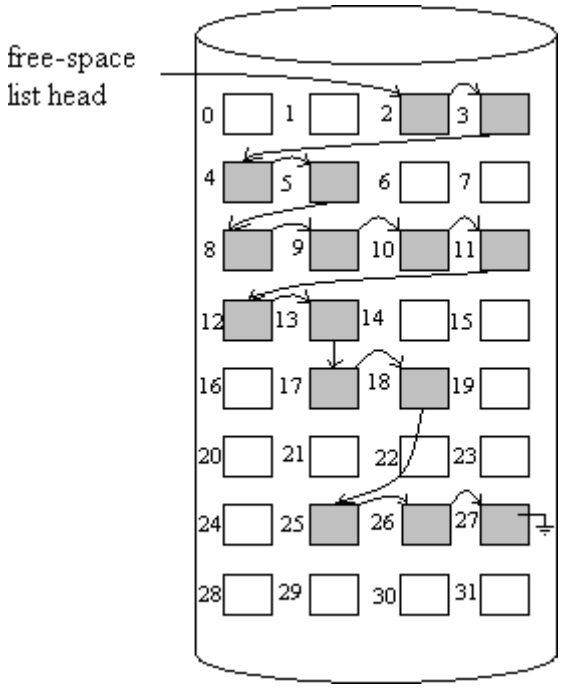
1. Bit Vector

- 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 block 2,3,4,5,8,9,10,11,12,13,17,18,25,26 and 27 are free, and the rest of the block are allocated. The free space bit map would be

  0011110011111100011000000011100000 …
- The main **advantage** of this approach is its relatively simplicity and efficiency in finding the first free block, or n consecutive free blocks on the disk.

2. 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.
- In our example, 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.
- The FAT method incorporates free-block accounting data structure. No separate method is needed.



3. 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-1 of 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.

4. Counting

- 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.

Recovery

- Files and directories are kept both in main memory and on disk, and care must be taken to ensure that system failure does not result in loss of data or in data inconsistency.

1. Consistency Checking

- The directory information in main memory is generally more up to date than is the corresponding information on the disk, because cached directory information is not necessarily written to disk as soon as the update takes place.
- Frequently, a special program is run at reboot time to check for and correct disk inconsistencies.
- The consistency checker—a systems program such as
- chkdsk in MS-DOS—compares the data in the directory structure with the data blocks on disk and tries to fix any inconsistencies it finds. The allocation and free-space-management algorithms dictate what types of problems the checker can find and how successful it will be in fixing them.

2. Backup and Restore

- Magnetic disks sometimes fail, and care must be taken to ensure that the data lost in such a failure are not lost forever. To this end, system programs can be used to **back** up data from disk to another storage device, such as a floppy disk, magnetic tape, optical disk, or other hard disk.
- Recovery from the loss of an individual file, or of an entire disk, may then be a matter of **restoring** the data from backup.

A **typical backup schedule** may then be as follows:

Day 1: Copy to a backup medium all files from the disk. This is called a **full backup.**

Day 2: Copy to another medium all files changed since day 1. This is an **incremental backup.**

Day 3: Copy to another medium all files changed since day 2.

Day N: Copy to another medium all files changed since day N— 1. Then go back to Day 1.

Log-Structured File Systems

- Computer scientists often find that algorithms and technologies originally used in one area are equally useful in other areas.
- These logging algorithms have been applied successfully to the problem of consistency checking.
- The resulting implementations are known as **log-based transaction-oriented** (or **journaling)** file systems.
- Fundamentally, all metadata changes are written sequentially to a log.
- Each set of operations for performing a specific task is a transaction.
- Once the changes are written to this log, they are considered to be committed, and the system call can return to the user process, allowing it to continue execution.
- As the changes are made, a pointer is updated to indicate which actions have completed and which are still incomplete.
- When an entire committed transaction is completed, it is removed from the log file, which is actually a circular buffer.
- A circular buffer writes to the end of its space and then continues at the beginning, overwriting older values as it goes. If the system crashes, the log file will contain zero or more transactions.