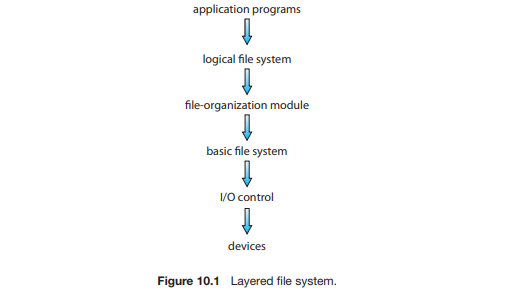
**FILE SYSTEM IMPLEMENTATION**

**File-System Structure:**

File systems provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily. A file system poses two quite different design problems. The first problem is defining how the file system should look to the user. This task involves defining a file and its attributes, the operations allowed on a file, and the directory structure for organizing files. The second problem is creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices. The file system itself is generally composed of many different levels. The structure shown in Figure 10.1 is an example of a layered design. Each level in the design uses the features of lower levels to create new features for use by higher levels. The lowest level, the I/O control, consists of device drivers and interrupts handlers to transfer information between the main memory and the disk system. A device driver can be thought of as a translator. Its input consists of high-level commands such as “retrieve block 123.” Its output consists of low level, hardware-specific instructions that are used by the hardware controller, which interfaces the I/O device to the rest of the system. The device driver usually writes specific bit patterns to special locations in the I/O controller’s memory to tell the controller which device location to act on and what actions to take.



**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). This layer also manages the memory buffers and caches that hold various file-system, directory, and data blocks. A block in the buffer is allocated before the transfer of a disk block can occur. When the buffer is full, the buffer manager must find more buffer memory or free up buffer space to allow a requested I/O to complete. Caches are used to hold frequently used file-system metadata to improve performance, so managing their contents is critical for optimum system performance.

**The file-organization module** knows about files and their logical blocks, as well as physical blocks. By knowing the type of file allocation used and the location of the file, the file-organization module can translate logical block addresses to physical block addresses for the basic file system to transfer. Each file’s logical blocks are numbered from 0 (or 1) through N. Since the physical blocks containing the data usually do not match the logical numbers, a translation is needed to locate each block. 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 the file-system structure except 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)** (an inode in most UNIX file systems) 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. These structures vary depending on the operating system and the file system, but some general principles apply.

On disk, the file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files. We describe them briefly

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

• A **volume-control block** (per volume) contains volume (or partition) details, such as the number of blocks in the partition, the size of the blocks, a free-block count and free-block pointers, and a free-FCB count and FCB pointers. In UFS, this is called a superblock; in NTFS, it is stored in the master file table.

• A directory structure (per file system) is used to organize the files. In UFS, this includes file names and associated inode numbers. In NTFS, it is stored in the master file table.

• A per-file FCB contains many details about the file. It has a unique identifier number to allow association with a directory entry. In NTFS, this information is actually stored within the master file table, which uses a relational database structure, with a row per file.

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

An in-memory **mount table** contains information about each mounted volume.

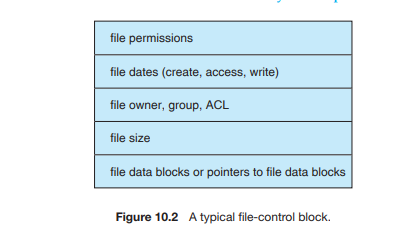
• An in-memory directory-structure cache holds the directory information of recently accessed directories. (For directories at which volumes are mounted, it can contain a pointer to the volume table.)

• The **system-wide open-file table** contains a copy of the FCB of each open file, as well as other information.

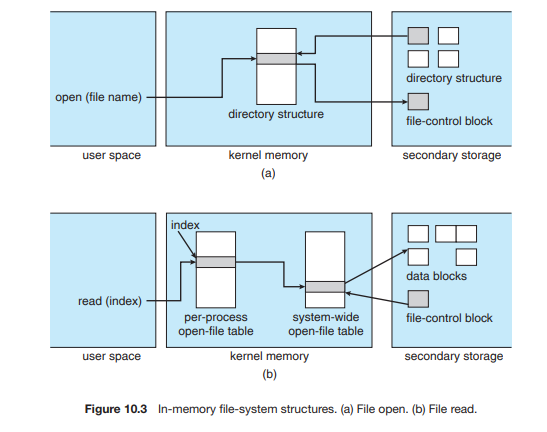
• The **per-process open-file table** contains a pointer to the appropriate entry in the system-wide open-file table, as well as other information.

• Buffers hold file-system blocks when they are being read from disk or written to disk.

To create a new file, an application program calls the logical file system. The logical file system knows the format of the directory structures. To create a new file, it allocates a new FCB. The system then reads the appropriate directory into memory, updates it with the new file name and FCB, and writes it back to the disk. A typical FCB is shown in Figure 10.2.

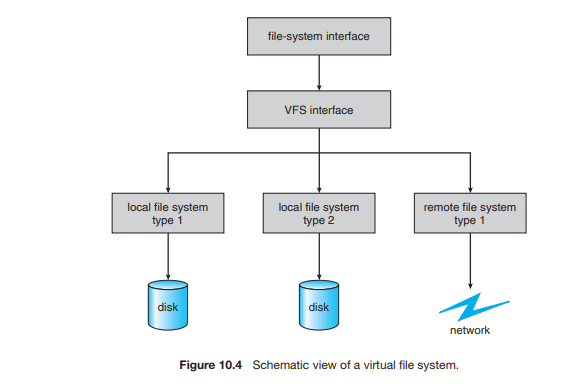


Now that a file has been created, it can be used for I/O. First, though, it must be opened. The open() call passes a file name to the logical file system. The open() system call first searches the system-wide open-file table to see if the file is already in use by another process. If it is, a per-process open-file table entry is created pointing to the existing system-wide open-file table. This algorithm can save substantial overhead. If the file is not already open, the directory structure is searched for the given file name. Parts of the directory structure are usually cached in memory to speed directory operations. Once the file is found, the FCB is copied into a system-wide open-file table in memory. This table not only stores the FCB but also tracks the number of processes that have the file open. Next, an entry is made in the per-process open-file table, with a pointer to the entry in the system-wide open-file table and some other fields. These other fields may include a pointer to the current location in the file (for the next read() or write() operation) and the access mode in which the file is open. The open() call returns a pointer to the appropriate entry in the per-process file-system table. All file operations are then performed via this pointer. The file name may not be part of the open-file table, as the system has no use for it once the appropriate FCB is located on disk. It may be cached, though, to save time on subsequent opens of the same file. The name given to the entry varies. UNIX systems refer to it as a file descriptor; Windows refers to it as a file handle. The operating structures of a file-system implementation are summarized in Figure 10.3



**Virtual File Systems:**

The file-system implementation consists of three major layers, as depicted schematically in Figure 10.4. The first layer is the file-system interface, based on the open(), read(), write(), and close() calls and on file descriptors. The second layer is called the virtual file system (VFS) layer. The VFS layer serves two important functions: 1. It separates file-system-generic operations from their implementation by defining a clean VFS interface. Several implementations for the VFS interface may coexist on the same machine, allowing transparent access to different types of file systems mounted locally. 2. It provides a mechanism for uniquely representing a file throughout a network. The VFS is based on a file-representation structure, called a vnode, that contains a numerical designator for a network-wide unique file. (UNIX inodes are unique within only a single file system.) This network-wide uniqueness is required for support of network file systems. The kernel maintains one vnode structure for each active node (file or directory). Thus, the VFS distinguishes local files from remote ones, and local files are further distinguished according to their file-system types. The VFS activates file-system-specific operations to handle local requests according to their file-system types and calls the NFS protocol procedures for remote requests. File handles are constructed from the relevant vnodes and are passed as arguments to these procedures. The layer implementing the file-system type or the remote-file-system protocol is the third layer of the architecture.



**Directory Implementation:**

**Linear List:** The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks. This method is simple to program but time-consuming to execute. The real disadvantage of a linear list of directory entries is that finding a file requires a linear search. A sorted list allows a binary search and decreases the average search time. So a more sophisticated tree data structure, such as a B-tree, might help here.

**Hash Table:** Another data structure used for a file directory is a hash table. 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 deletion are also fairly straightforward, although some provision must be made for collisions—situations in which two file names hash to the same location.

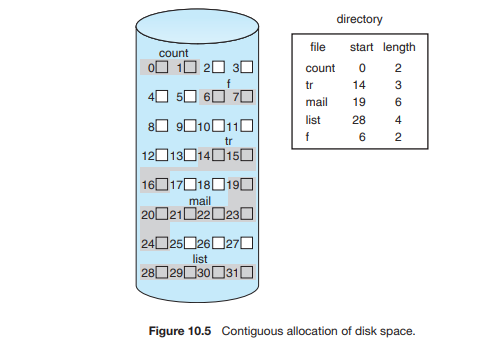
The major difficulties with a hash table are its generally fixed size and the dependence of the hash function on that size. For example, assume that we make a linear-probing hash table that holds 64 entries. The hash function converts file names into integers from 0 to 63, for instance, by using the remainder of a division by 64. If we later try to create a 65th file, we must enlarge the directory hash table—say, to 128 entries. As a result, we need a new hash function that must map file names to the range 0 to 127, and we must reorganize the existing directory entries to reflect their new hash-function values. Alternatively, a chained-overflow hash table can be used. Each hash entry can be a linked list instead of an individual value, and we can resolve collisions by adding the new entry to the linked list.

**File Allocation Methods:**

There are 3 major methods of allocating disk space.

**Contiguous allocation:-**

The contiguous allocation method requires each file occupy a set of contiguous blocks on the disk. Contiguous allocation of a file is defined by the disk address and length 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 length of the area allocated for this file shown in Fig 10.5. Contiguous allocation of file is very easy to access. For the 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 contagious allocation. Contiguous allocation has some problems. One difficulty with this method is finding space for a new file. Also there are many problems with this method a) external fragmentation:- files are allocated and deleted , the free disk space is broken in to little pieces. The external fragmentation exists when free space is broken in to chunks (large piece) and these chunks are not sufficient for a request of new file. There is a solution for external fragmentation i.e compaction. All free space compact in to one contiguous space. However, the cost of this compaction is time and it can be particularly severe for large hard disks that use contiguous allocation, where compacting all the space may take hours and may be necessary on a weekly basis. b) Another problem is determining how much space is needed for a file. When file is created the creator must specifies the size of that file. This becomes to big problem. Suppose if we allocate too little space to a file , sometimes it may not sufficient. Suppose if we allocate large space sometimes space is wasted. c) Another problem is if one large file is deleted, that large space is becomes to empty. Another file is loaded in to that space whose size is very small then some space is wasted. That wastage of space is called internal fragmentation.



**Linked Allocation:**

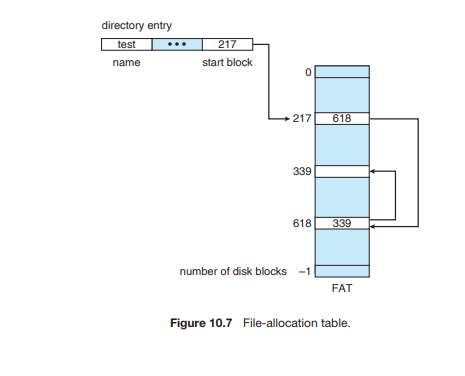
Linked allocation solves all the problems of contagious allocation. With linked allocation, each file is a linked list of disk blocks, the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. For Ex:- a file have five blocks start at block 9, continue at block 16,then block 1, block 10 and finally block 25 see in figure 10.6. Each block contains a ponter to the next block. These pointers are not made available to the user. To create a new file we simply create a new entry in 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 the free space-management system to find a free block, and this new block is written to and is linked to the end of the file. To read a file, we simply read blocks by following the pointers from block to block. There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request. The size of a file need not be declared when that file is created. A file can continue to grow as long as free blocks are available. Consequently, it is never necessary to compact disk space.

Linked allocation does have disadvantages, however. The major problem is that it can be 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 access to a pointer requires a disk read, and some require a disk seek. Consequently, it is inefficient to support a direct-access capability for linked-allocation files.

Another disadvantage is it requires space for the pointers. If a pointer requires 4 bytes out of 512 byte block, then 0.78% of disk is being used for pointers, rather than for information. The solution to this problem is to allocate blocks in to multiples, called clusters and to allocate the clusters rather than blocks.

another problem of linked allocation is reliability. Recall that the files are linked together by pointers scattered all over the disk, and consider what would happen if a pointer were lost or damaged.

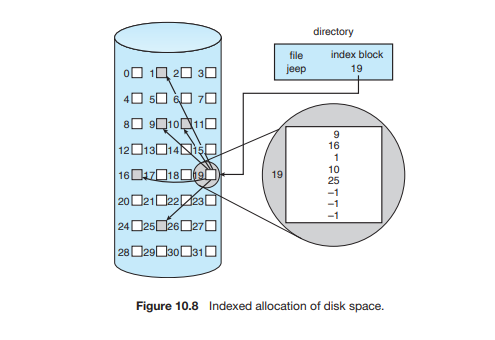
FAT( file allocation table):- An important variation on the linked allocation method is the use of a file allocation table. The table has one entry for each disk block, and is indexed by block number. The FAT is used much as is a linked list. The directory entry contains the block number of the first block of the file. The table entry contains the block number then contains the block number of the next block in the file. This chain continuous until the last block, which has a special end of file values as the table entry. Unused blocks are indicated by a ‘0’ table value. Allocation a new block to a file is a simple. First finding the first 0-value table entry, and replacing the previously end of file value with the address of the new block. The 0 is then replaced with end of file value. An illustrative example is the FAT structure shown in Figure 10.7 for a file consisting of disk blocks 217, 618, and 339.



**Indexed Allocation:**

Linked allocation solves the external fragmentation and size declaration problems of contagious allocation. However in the absence of a FAT, linked allocation cannot support efficient direct access. Since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order. Indexed allocation solves this problem by bringing all the pointers together in to one location i.e the index block.

Each file has its own index block ,which is an array of disk block addresses. The i th entry in the index block points to the ith block of the file. The directory contains the address of the index block shown in Fig 10.8 . To read the ith block we use the pointer in the ith index block entry. 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. It supports the direct access without suffering from external fragmentation because any free block on the disk can satisfy a request for more space. It suffers from the wasted space. The pointer overhead of the index block is generally greater than the pointer over head 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. With indexed allocation, an entire index block must be allocated, even if only one or two pointers will be non-nil. 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. Mechanisms for this purpose include the following:

• Linked scheme. An index block is normally one disk block. Thus, it can be read and written directly by itself. To allow for large files, we can link together several index blocks. 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).

• Multilevel index. A variant of linked representation uses a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks. To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block. This approach could be continued to a third or fourth level, depending on the desired maximum file size. With 4,096-byte blocks, we could store 1,024 four-byte pointers in an index block. Two levels of indexes allow 1,048,576 data blocks and a file size of up to 4 GB.

• 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, they contain addresses of blocks that contain data of the file. Thus, the data for small files (of no more than 12 blocks) do not need a separate index block. If the block size is 4 KB, then up to 48 KB of data can be accessed directly. The next three pointers point to indirect blocks. The first points to a single indirect block, which is an index block containing not data but the addresses of blocks that do contain data. The second points to a double indirect block, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. The last pointer contains the address of a triple indirect block.

**Free Space Management:**

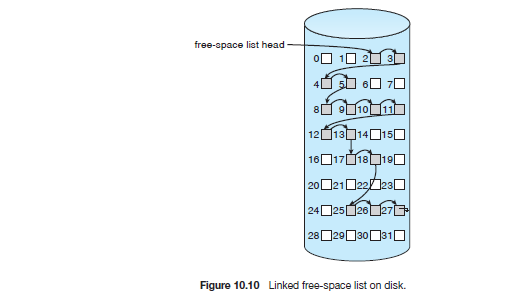
To keep track of free disk space, the system maintains a free space list. The free space list records all free disk blocks — those not allocated to some file or directory. To create a file we search the free space list for the required amount of space, and allocate that space to the new file. This space is then removed from the free space list. When the file is deleted, its disk space is added to the free space list. There are many methods to find the free space.

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.

Ex:- consider a disk where blocks 2,3,4,5,8,9,10,11,12,13,17,18,25, 26 and 27 are free and rest of 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

1. **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 contain a pointer to the next free disk block, and so on. In this approach, 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 (Figure 10.10). However this scheme is not efficient to traverse the list, we must read each block, which requires I/O time. Disk space is also wasted to maintain the pointer to next free space.



1. **Grouping:-** Another method is 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 main advantage of this approach is that the addresses of a large no.of blocks can be found quickly.
2. **Counting:-** Another approach is counting. Generally several contiguous blocks may be allocated or freed simultaneously. Particularly when space is allocated with the contiguous allocation algorithm or through clustering. Thus, rather than keeping a list of ‘n’ free disk address, we can keep the address of 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.