Chapter 11 IMPLEMENTING FILE SYSTEMS

FILE SYSTEM STRUCTURE

Disks provide the bulk of secondary-storage on which a file-system is maintained.
The disk is a suitable medium for storing multiple files. This is because of two characteristics

1. A disk can be rewritten in place; it is possible to read a block from the disk, modify the block, and write it back into the same place.
2. A disk can access directly any block of information it contains. Thus, it is simple to access any file either sequentially or randomly, and switching from one file to another requires only moving the read-write heads and waiting for the disk to rotate.

To improve I/O efficiency, I/O transfers between memory and disk are performed in units of blocks. Each block has one or more sectors. Depending on the disk drive, sector-size varies from 32 bytes to 4096 bytes. The usual size is 512 bytes.

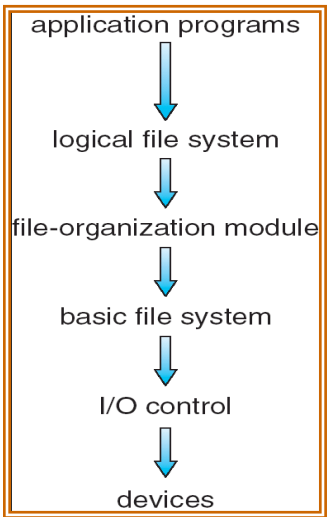
File-systems provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily

Design problems of file-systems:

1. Defining how the file-system should look to the user.
2. Creating algorithms & data-structures to map the logical file-system onto the physical secondary-storage devices.

Layered File Systems:

The file-system itself is generally composed of many different levels. Every level in design uses features of lower levels to create new features for use by higher levels.



File system 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 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 interrupt handlers to transfer information between the main memory and the disk system.

A device driver can be thought of as a translator. Its input consists of high-level commands such as "retrieve block 123."
Its output consists of low level, hardware-specific instructions that are used by the hardware controller, which interfaces the I/0 device to the rest of the system.
The device driver usually writes specific bit patterns to special locations in the I/0 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.

- **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.

- **Logical file system** manages metadata information. Metadata includes all of 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(FCB)**.

FCB contains information about the file, including ownership, permissions, and location of the file contents.

File System Implementation

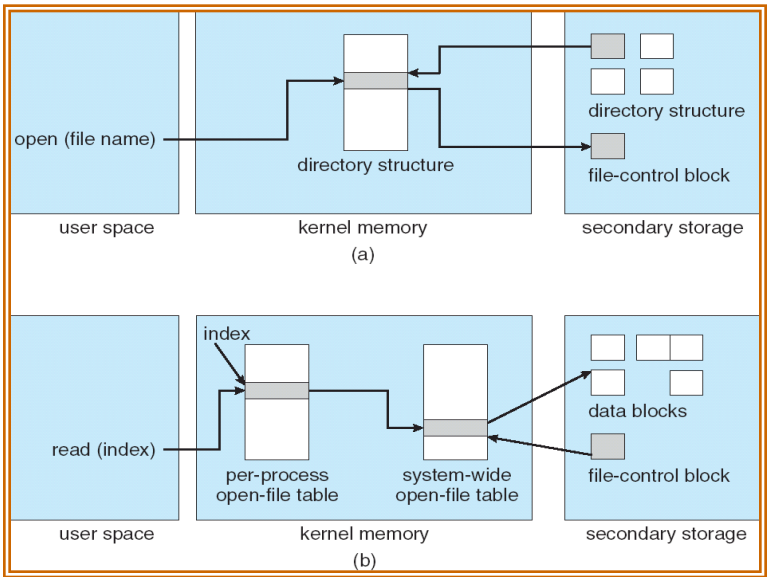
On disk, the file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files.

- Boot Control Block : 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.
- Volume Control Block : (per volume) contains volume (or partition) details, such as the number of blocks in the partition, the size of the blocks, a free-block count and free-block pointers and a free-FCB count and FCB pointers.

- A directory structure (per file system) is used to organize the files.
- A per-file FCB contains many details about the file. It has a unique identifier number to allow association with a directory entry.

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



Steps for creating a file :

1) An application program calls the logical file system, which knows the format of the directory structures
2) The logical file system allocates a new file control block (FCB)
    - If all FCBs are created at file-system creation time, an FCB is allocated from the free list
3) The logical file system then
    - Reads the appropriate directory into memory
    - Updates the directory with the new file name and FCB
    - Writes the directory back to the disk

UNIX treats a directory exactly the same as a file by means of a type field in the inode
Windows NT implements separate system calls for files and directories and treats directories as entities separate from files .

Steps for opening a file:

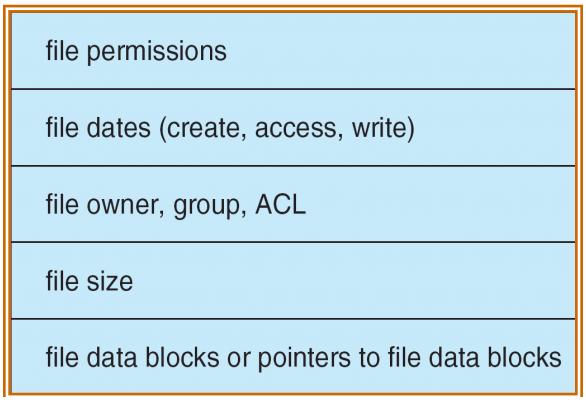
1) The function first searches the system-wide open-file table to see if the file is already in use by another process

- If it is, a per-process open-file table entry is created pointing to the existing system-wide open-file table
- This algorithm can have substantial overhead; consequently, parts of the directory structure are usually cached in memory to speed operations

2) Once the file is found, the FCB is copied into a system-wide open-file table in memory
   - This table also tracks the number of processes that have the file open
3) 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
4) The function then returns a pointer/index to the appropriate entry in the per-process file-system table
   - All subsequent file operations are then performed via this pointer
   - UNIX refers to this pointer as the file descriptor
   - Windows refers to it as the file handle

Steps for closing a file:

1) The per-process table entry is removed
2) The system-wide entry's open count is decremented
3) When all processes that have opened the file eventually close it
   Any updated metadata is copied back to the disk-based directory structure
   The system-wide open-file table entry is removed



Partitions and Mounting

- Each partition can be either "raw," containing no file system, or "cooked," containing a file system.
- Raw disk is used where no file system is appropriate.
- UNIX swap space can use a raw partition, for example, as it uses its own format on disk and does not use a file system.
- Boot information can be stored in a separate partition. Again, it has its own format, because at boot time the system does not have the file-system code loaded and therefore cannot interpret the file-system format.
- Boot information is a sequential series of blocks, loaded as an image into memory.
- Execution of the image starts at a predefined location, such as the first byte. This boot loader in turn knows about the file-system structure to be able to find and load the kernel and start it executing.
- The root partition which contains the operating-system kernel and sometimes other system files, is mounted at boot time.
- Other volumes can be automatically mounted at boot or manually mounted later, depending on the operating system.

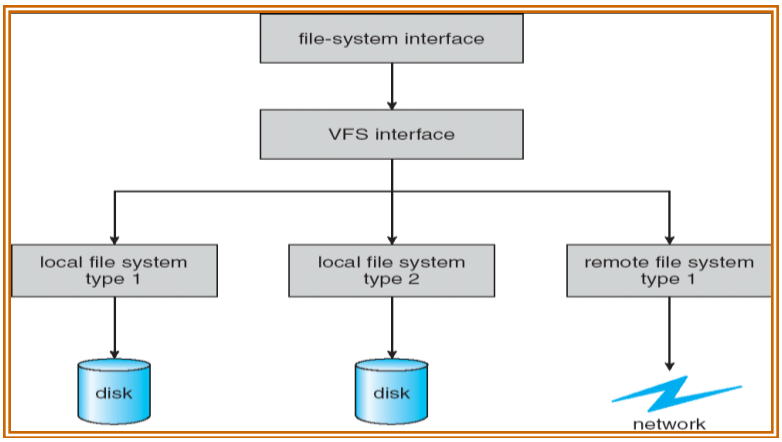
- After successful mount operation, the operating system verifies that the device contains a valid file system.
- It is done by asking the device driver to read the device directory and verifying that the directory has the expected format.
- If the format is invalid, the partition must have its consistency checked and possibly corrected, either with or without user intervention.
- Finally, the operating system notes in its in-memory mount table that a file system is mounted, along with the type of the file system.
- The root partition is mounted at boot time
    - It contains the operating-system kernel and possibly other system files
- Other volumes can be automatically mounted at boot time or manually mounted later
- As part of a successful mount operation, the operating system verifies that the storage device contains a valid file system
    - It asks the device driver to read the device directory and verify that the directory has the expected format
    - If the format is invalid, the partition must have its consistency checked and possibly corrected
    - Finally, the operating system notes in its in-memory mount table structure that a file system is mounted along with the type of the file system
        - Microsoft Windows-based systems mount each volume in a separate name space denoted by a letter and a colon (e.g., C:)
            - The operating system places a pointer to the file system in the field of the device structure corresponding to the drive letter
        - UNIX allows file systems to be mounted at any directory
            - Mounting is implemented by setting a flag in the in-memory copy of the inode for the directory

Virtual file Systems

The file-system implementation consists of three major layers, as depicted schematically in Figure. 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:

- 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.
- It provides a mechanism for uniquely representing a file throughout a network. The VFS is based on a file-representation structure, called a vnode that contains a numerical designator for a network-wide unique file. This network-wide uniqueness is required for support of network file systems.

The kernel maintains one vnode structure for each active node.
Thus, the VFS distinguishes local files from remote ones, and local files are further distinguished according to their file-system types.



Directory Implementation

Selection of directory allocation and directory management algorithms significantly affects the efficiency, performance, and reliability of the file system

One Approach: Direct indexing of a linear list

- Consists of a list of file names with pointers to the data blocks
- Simple to program
- Time-consuming to search because it is a linear search.
- Sorting the list allows for a binary search; however, this may complicate creating and deleting files
- To create a new file, we must first search the directory to be sure that no existing file has the same name.
- Add a new entry at the end of the directory.
- To delete a file, we search the directory for the named file and then release the space allocated to it.
- To reuse the directory entry,we can do one of several things. Mark the entry as unused .
- An alternative is to copy the last entry in the directory into the freed location and to decrease the length of the directory.
- Directory information is used frequently, and users will notice if access to it is slow.

Another Approach: List indexing via a hash function

- Takes a value computed from the file name and returns a pointer to the file name in the linear list
- Greatly reduces the directory search time
- **Can result in collisions** – situations where two file names hash to the same location
- A hash table are its generally fixed size and the dependence of the hash function on that size. (i.e., fixed number of entries).
- 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.

Allocation methods

Allocation methods address the problem of allocating space to files so that disk space is utilized effectively and files can be accessed quickly.

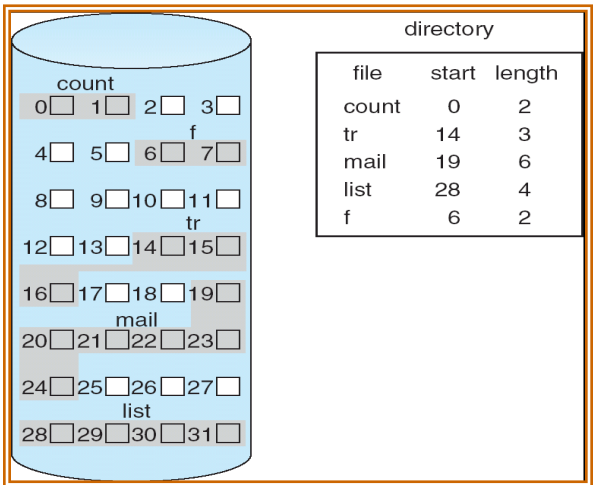
Three methods exist for allocating disk space

- **Contiguous allocation**
- **Linked allocation**
- **Indexed allocation**

Contiguous allocation :

- Requires that each file occupy a set of contiguous blocks on the disk
- Accessing a file is easy – only need the starting location (block #) and length (number of blocks)

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.



Disadvantages :

1.Finding space for a new file is difficult. The system chosen to manage free space determines how this task is accomplished.Any management system can be used, but some are slower than others.
2.Satisfying a request of size *n* from a list of free holes is a problem. First fit and best fit are the most common strategies used to select a free hole from the set of available holes.
3.The above algorithms suffer from the problem of external fragmentation.

- As files are allocated and deleted, the free disk space is broken into pieces.
- External fragmentation exists whenever free space is broken into chunks.
- It becomes a problem when the largest contiguous chunk is insufficient for a request; storage is fragmented into a number of holes, none of which is large enough to store the data.
- Depending on the total amount of disk storage and the average file size, external fragmentation may be a minor or a major problem.

Compaction :
One strategy for preventing loss of significant amounts of disk space to external fragmentation is to copy an entire file system onto another disk or tape. The original disk is then freed completely, creating one large contiguous free space. We then copy the files back onto the original disk by allocating contiguous space from this one large hole. This scheme effectively **compacts** all free space into one contiguous space, solving the fragmentation problem.
4.Determining space needed for a file. When the file is created, the total amount of space it will need must be found and allocated.

- If we allocate too little space to a file, we may find that the file cannot be extended.
- Solution :
  1. 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.
  2.Find a larger hole, copy the contents of the file to the new space, and release the previous space.

5. A file that will grow slowly over a long period(months or years) must be allocated enough space for its final size, even though much of that space will be unused for a long time.

Solution : Some operating systems use a modified contiguous-allocation scheme. Here, a contiguous chunk of space is allocated initially; then, if that amount proves not to be large enough, another chunk *of* contiguous space, known as an **extent** is added. The location of a file's blocks is then recorded as a location and a block count, plus a link to the first block of the next extent.

Linked Allocation :

- Solves the problems of contiguous allocation
- Each file is a linked list of disk blocks: blocks may be scattered anywhere on the disk
- The directory contains a pointer to the first and last blocks of a file
- Creating a new file requires only creation of a new entry in the directory
- Writing to a file causes the free-space management system to find a free block

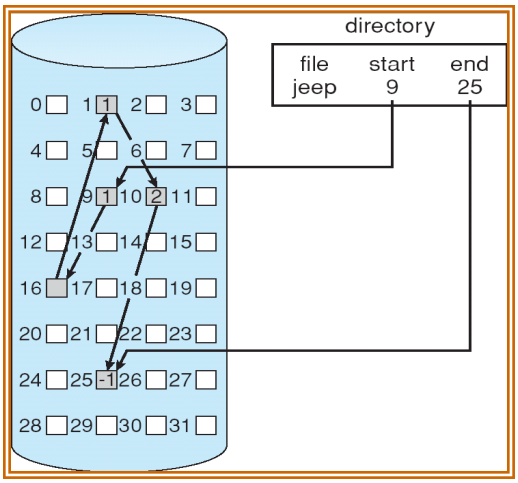
➢ This new block is written to and is linked to the end of the file

➢ Reading from a file requires only reading blocks by following the pointers from block to block

Advantages

- There is no external fragmentation
- Any free blocks on the free list can be used to satisfy a request for disk space
- The size of a file need not be declared when the file is created
- A file can continue to grow as long as free blocks are available

  - It is never necessary to compact disk space for the sake of linked allocation (however, file access efficiency may require it)



Each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. For example, a file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10, and finally block 25. Each block contains a pointer to the next block. These pointers are not made available to the user. A disk address (the pointer) requires 4 bytes in the disk.

To **create** a new file, we simply create a new entry ile 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 the free-space management system to filed 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.

Disadvantages :

1. The major problem is that it can be used effectively only for sequential-access files. To filed 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.

2. Space required for the pointers. Solution is clusters. Collect blocks into multiples and allocate clusters rather than blocks

3. Reliability - the files are linked together by pointers scattered all over the disk and if a pointer were lost or damaged then all the links are lost.

File Allocation Table :

A section of disk at the beginning of each volume is set aside to contain the table. The table has one entry for each disk block and is indexed by block number.

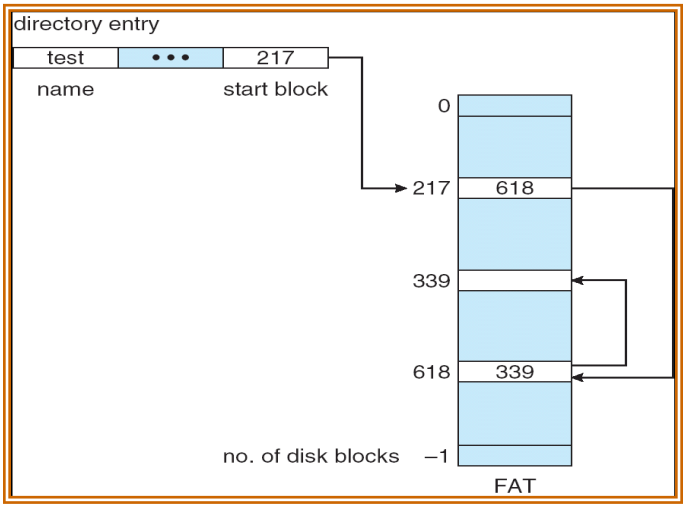
The FAT is used in much the same way as a linked list. The directory entry contains the block number of the first block of the file.

The table entry indexed by that block number contains the block number of the next block in the file.

The chain continues until it reaches the last block, which has a special end-of-file value as the table entry.

An unused block is indicated by a table value of 0.

Consider a FAT with a file consisting of disk blocks 217, 618, and 339.



Indexed allocation :

Brings all the pointers together into one location called 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 . To find and 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.

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

Disadvantages :

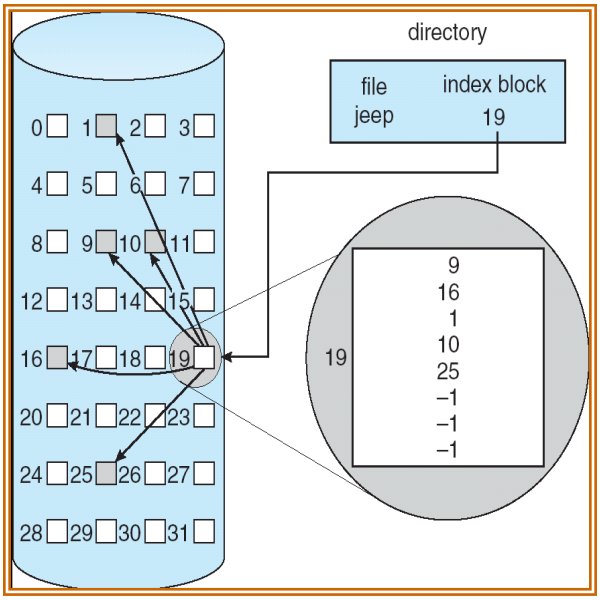
Suffers from some of the same performance problems as linked allocation

Index blocks can be cached in memory; however, data blocks may be spread all over the disk volume

Indexed allocation does suffer from wasted space.

The pointer overhead of the index block is generally greater than the pointer overhead of

linked allocation.



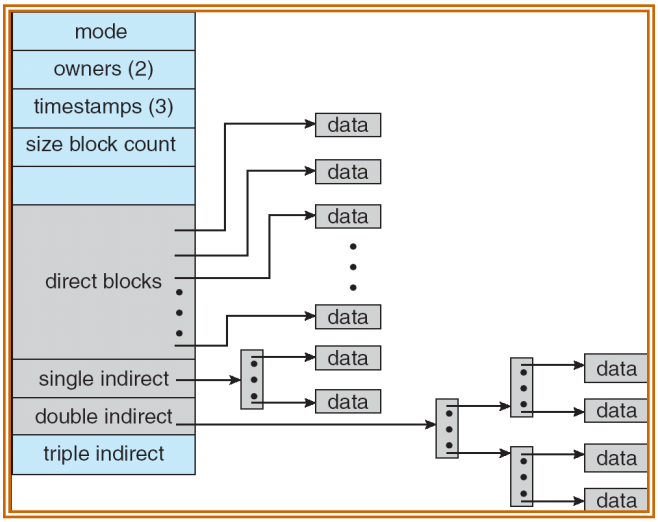
If the index block is too small, however, it will not be able to hold enough pointers for a large file, and a mechanism will have to be available to deal with this issue. Mechanisms for this purpose include the following:

a) **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).

b) **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

c) **Combined scheme.** For eg. 15 pointers of the index block is maintained 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**.



Performance

Contiguous allocation requires only one access to get a disk block. Since we can easily keep the initial address of the file in memory, we can calculate immediately the disk address of the *ith* block and read it directly.

For linked allocation, we can also keep the address of the next block in memory and read it directly. This method is fine for sequential access.:Linked allocation should not be used for an application requiring direct access.

Indexed allocation is more complex. If the index block is already in memory, then the access can be made directly. However, keeping the index block in memory requires considerable space. If this memory space is not available, then we may have to read first the index block and then the desired data block.

Free Space Management

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.

Bit Vector

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

For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free-space bit map would be 001111001111110001100000011100000 ...

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

A technique for finding the first free block on a system that uses a bit-vector to allocate disk space is to sequentially check each word in the bit map to see whether that value is not 0, since a 0-valued word contains only 0 bits and represents a set of allocated blocks. The first non-0 word is scanned for the first 1 bit, which is the location of the first free block. The calculation of the block number is

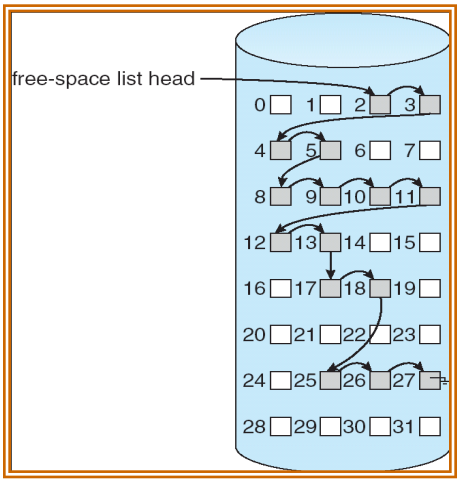
(number of bits per word) x (number of 0-value words) +offset of first 1 bit.

Linked List

Another approach to free-space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory. This first block contains a pointer to the next free disk block, and so on.

Blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 were free and the rest of the blocks were allocated. A pointer to block 2 is kept 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).

This scheme is not efficient to traverse the list, we must read each block, which requires substantial I/0 time. Traversing the free list is not a frequent action. Usually, the operating system simply needs a free block so that it can allocate that block to a file, so the first block in the free list is used. The FAT method incorporates free-block accounting into the allocation data structure.



Grouping

A modification of the free-list approach stores the addresses of *n* free blocks in the first free block. The first n-1 of these blocks are actually free. The last block contains the addresses of another *n* free blocks, and so on. The addresses of a large number of free blocks can now be found quickly, when the standard linked-list approach is used.

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

Space maps

- Sun's ZFS file system (hierarchical structure) uses a combination of techniques in its free-space management algorithm to control the size of data structures and minimize the I/0 needed to manage those structures.
- First, ZFS creates to divide the space on the device into chucks of manageable size. A given volume may contain hundreds of metaslabs. Each metaslab has an associated space map. ZFS uses the counting algorithm to store information about free blocks.
- Rather than write count structures to disk, it uses log-structured file- system techniques to record them. The space map is a log of all block activity (allocating and freeing), in time order, in counting format.
- When ZFS decides to allocate or free space from a metaslab, it loads the associated space map into memory in a balanced-tree structure (for very efficient operation), indexed by offset, and replays the log into that structure.