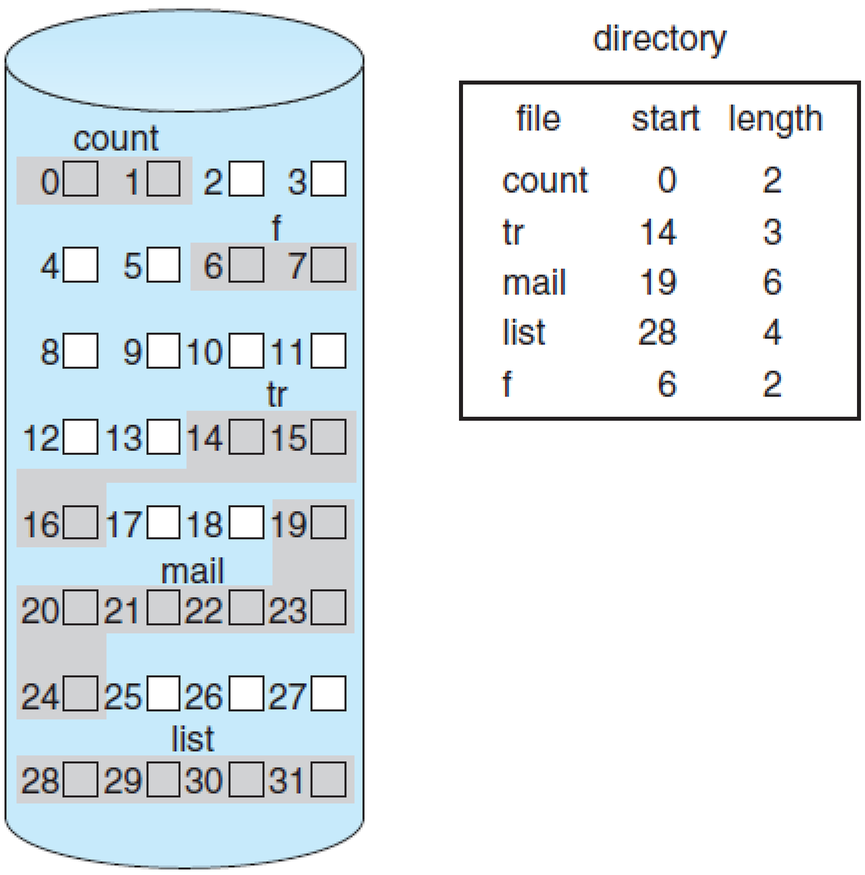
FILE ALLOCATION METHODS

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

Contiguous Allocation

- **Contiguous allocation** requires that each file occupy a set of contiguous blocks on the disk.
- Disk addresses define a linear ordering on the disk.
- Accessing block $b + 1$ after block b normally requires no head movement.
- When head movement is needed (from the last sector of one cylinder to the first sector of the next cylinder), the head need only move from one track to the next.
- Thus, the number of disk seeks required for accessing contiguously allocated files is minimal.
- Contiguous allocation of a file is defined by the disk address of the first block and length.
- If the file is n blocks long and starts at location b, then it occupies blocks $b, b + 1, b + 2, ..., b + n - 1$.



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

Figure 12.5 Contiguous allocation of disk space.

- 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 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.

Drawbacks:

- One difficulty is finding space for a new file. It suffers from **dynamic storage-allocation** problem which involves how to satisfy a request of size *n* from a list of free holes.

- First fit and best fit are the most common strategies used to select a free hole from the set of available holes. Simulations have shown that both first fit and best fit are more efficient than worst fit in terms of both time and storage utilization.
- Neither first fit nor best fit is clearly best in terms of storage utilization, but first fit is generally faster.
- All these algorithms suffer from the problem of **external fragmentation**. As files are allocated and deleted, the free disk space is broken into little pieces. External fragmentation exists whenever free space is broken into chunks. It becomes a problem when the largest contiguous chunk is insufficient for a request; 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.
- **Solution is compaction, which compact** all free space into one contiguous space, solving the fragmentation problem.
- The cost of this compaction is time and the cost can be particularly high for large hard disks. Compacting these disks may take hours and may be necessary on a weekly basis.
- Some systems require that this function be done **off-line**, with the file system un-mounted. During this **down time**, normal system operation generally cannot be permitted.

- Most modern systems that need defragmentation can perform it **on-line** during normal system operations, but the performance penalty can be substantial.
- **Another problem** with contiguous allocation is determining how much space is needed for a file. When the file is created, the total amount of space it will need must be found and allocated.
- How does the creator (program or person) know the size of the file to be created? In some cases, this determination may be fairly simple. In general, however, the size of an output file may be difficult to estimate.
- If we allocate too little space to a file, we may find that the file cannot be extended. Especially with a best-fit allocation strategy, the space on both sides of the file may be in use. Hence, we cannot make the file larger in place.
- Two possibilities then exist. First, the user program can be terminated, with an appropriate error message. The user must then allocate more space and run the program again. These repeated runs may be costly. To prevent them, the user will normally overestimate the amount of space needed, resulting in considerable wasted space.
- The other possibility is to find a larger hole, copy the contents of the file to the new space, and release the previous space. This series of actions can be repeated as long as space exists, although it can be time consuming. The user need never be informed explicitly about what is happening,

however; 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 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. The file therefore has a large amount of internal fragmentation.
- To minimize these drawbacks, some OS 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.
- Internal fragmentation can still be a problem if the extents are too large, and external fragmentation can become a problem as extents of varying sizes are allocated and deallocated.

Linked Allocation

- **Linked allocation** solves all problems of contiguous allocation.
- With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk.

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

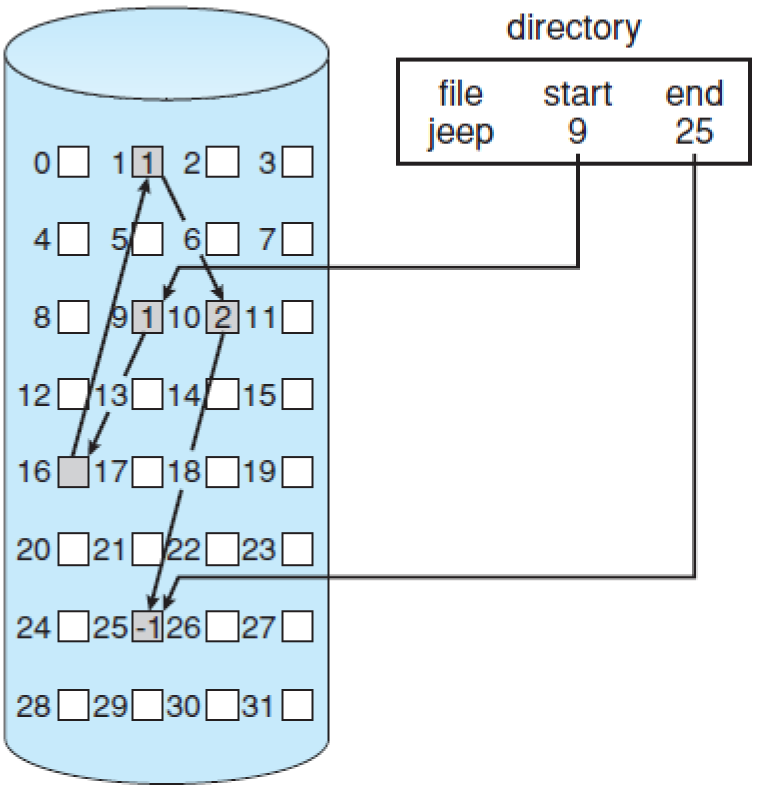


Figure 12.6 Linked allocation of disk space.

- Each block contains a pointer to the next block. If each block is 512 bytes in size, and the pointer requires 4 bytes, then the user sees blocks of 508 bytes.
- To create a new file, we simply create a new entry in the directory. The first pointer is initialized to null 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 the 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.

<u>Drawbacks</u>

- The major problem is that it can be used **effectively only for sequential-access files and not suitable for direct access files**.
- To find the *i*th block of a file, we must start at the beginning of that file and follow the pointers until we get to the *i*th block.
- Each access to a pointer requires a disk read, and some require a disk seek. Consequently, it is inefficient to support a direct-access capability for linked-allocation files.
- Another disadvantage is the **space required for the pointers**.
- If a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information.
- Each file requires slightly more space than it would otherwise. The usual solution to this problem is to collect

blocks into multiples, called **clusters**, and to allocate clusters rather than blocks.

- For instance, the file system may define a cluster as four blocks and operate on the disk only in cluster units.
- Pointers then use a much smaller percentage of the file's disk space.
- It may increase in internal fragmentation, because more space is wasted when a cluster is partially full
- Clusters can be used to improve the disk-access time for many other algorithms as well, so they are used in most file systems.
- Yet another problem of linked allocation is **reliability**. Recall that the files are linked together by pointers scattered all over the disk, and consider what would happen if a pointer were lost or damaged.
- A bug in the OS software or a disk hardware failure might result in picking up the wrong pointer.
- This error could in turn result in linking into the free-space list or into another file.
- One partial solution is to use doubly linked lists, and another is to store the file name and relative block number in each block.
- However, these schemes require even more overhead for each file.

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

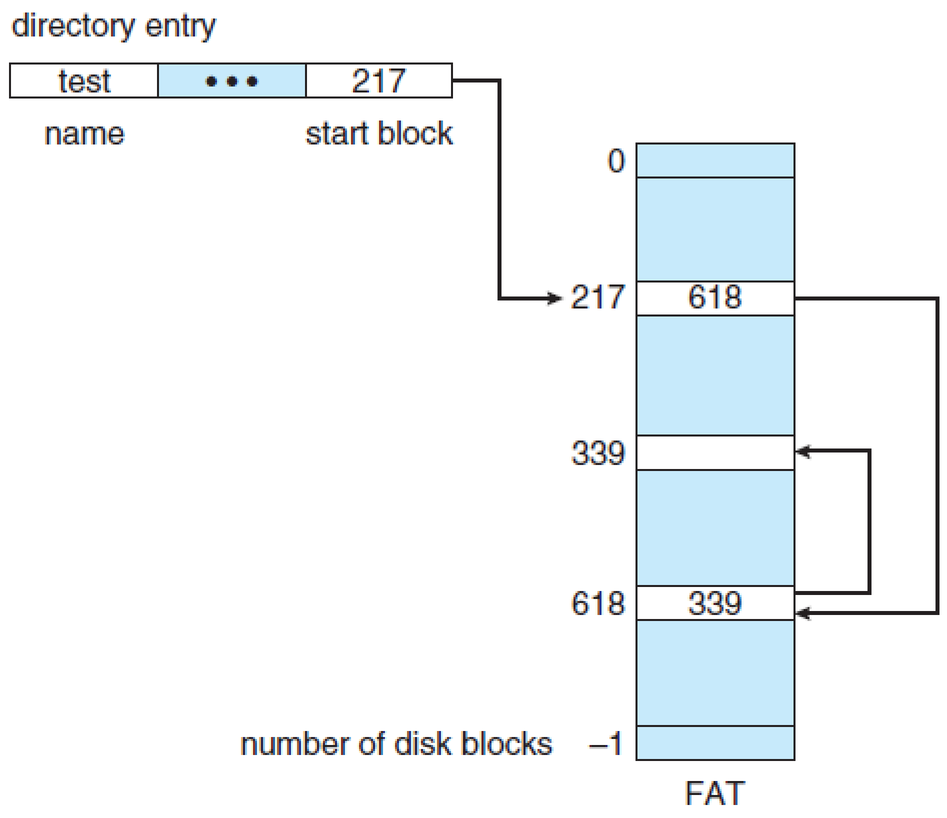


Figure 12.7 File-allocation table.

Indexed 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 must 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.

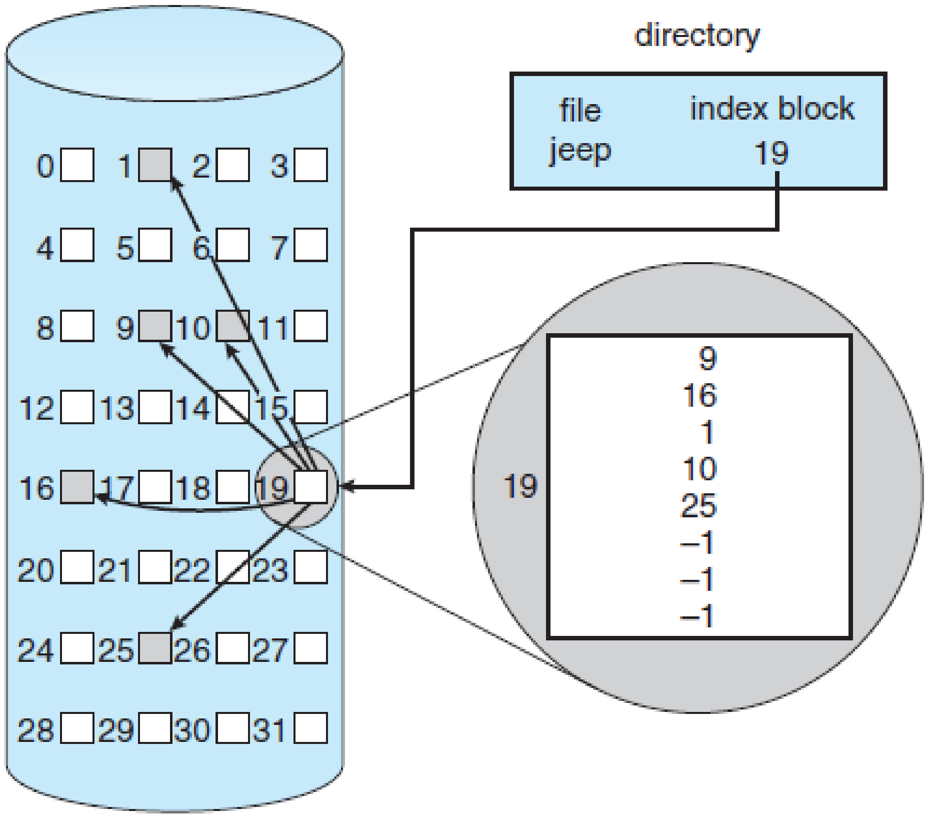


Figure 12.8 Indexed allocation of disk space.

- 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 find and read the *ith* block, we use the pointer in the *ith* index-block entry.
- This scheme is similar to the paging scheme
- Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk can satisfy a request for more space.
- Indexed allocation does suffer from wasted space, however.

- The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation.
- Consider a common case in which we have a file of only one or two blocks. With linked allocation, we lose the space of only one pointer per block. With indexed allocation, an entire index block must be allocated, even if only one or two pointers will be non-null.
- 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:
    1. Linked Scheme
    2. Multi-level index
    3. Combined Scheme

Linked scheme

- An index block is normally one disk block. 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 null (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, OS uses the first-level index to find a second-level index block and then uses that block to find the desired data block.
- This approach could be continued to a third or fourth level, depending on the desired maximum file size

Combined scheme

- Used in UNIX-based file systems
- It keeps the first 15 pointers of the index block in the file's inode. The first 12 of these pointers point to **direct blocks**; that is, they contain addresses of blocks that contain data of the file.
- Thus, the data for small files (of no more than 12 blocks) do not need a separate index block.
- The next three pointers point to **indirect blocks**. The first points to a **single indirect block**, which is an index block containing not data but the addresses of blocks that do contain data. The second points to a **double indirect block**, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual

data blocks. The last pointer contains the address of a **triple indirect block**.

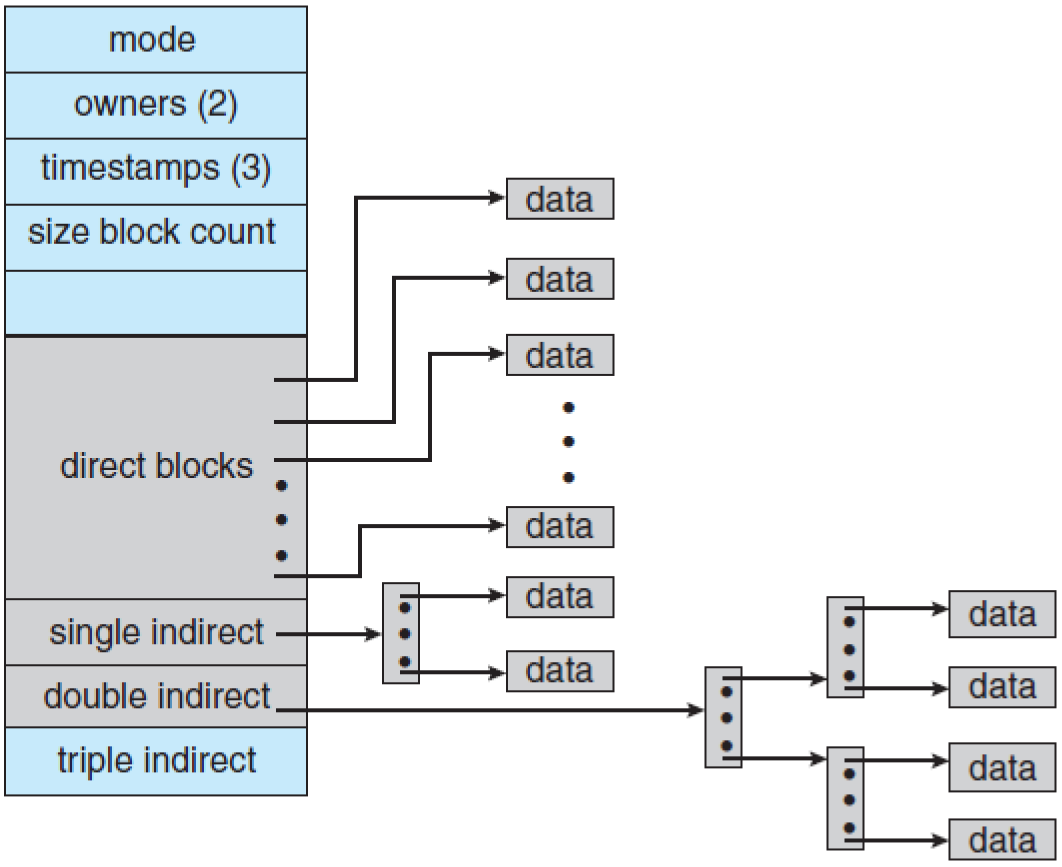


Figure 12.9 The UNIX inode.

<u>Drawback</u>

- 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.
- Some systems combine contiguous allocation with indexed allocation by using contiguous allocation for

small files (up to three or four blocks) and automatically switching to an indexed allocation if the file grows large.

- Since most files are small, and contiguous allocation is efficient for small files, average performance can be quite good.

STORAGE MANAGEMENT

MAGNETIC DISKS

- **Magnetic disks** provide the bulk of secondary storage for modern computer systems. Conceptually, disks are relatively simple.
- Each disk **platter** has a flat circular shape, like a CD. Common platter diameters range from 1.8 to 3.5 inches. The two surfaces of a platter are covered with a magnetic material. We store information by recording it magnetically on the platters.

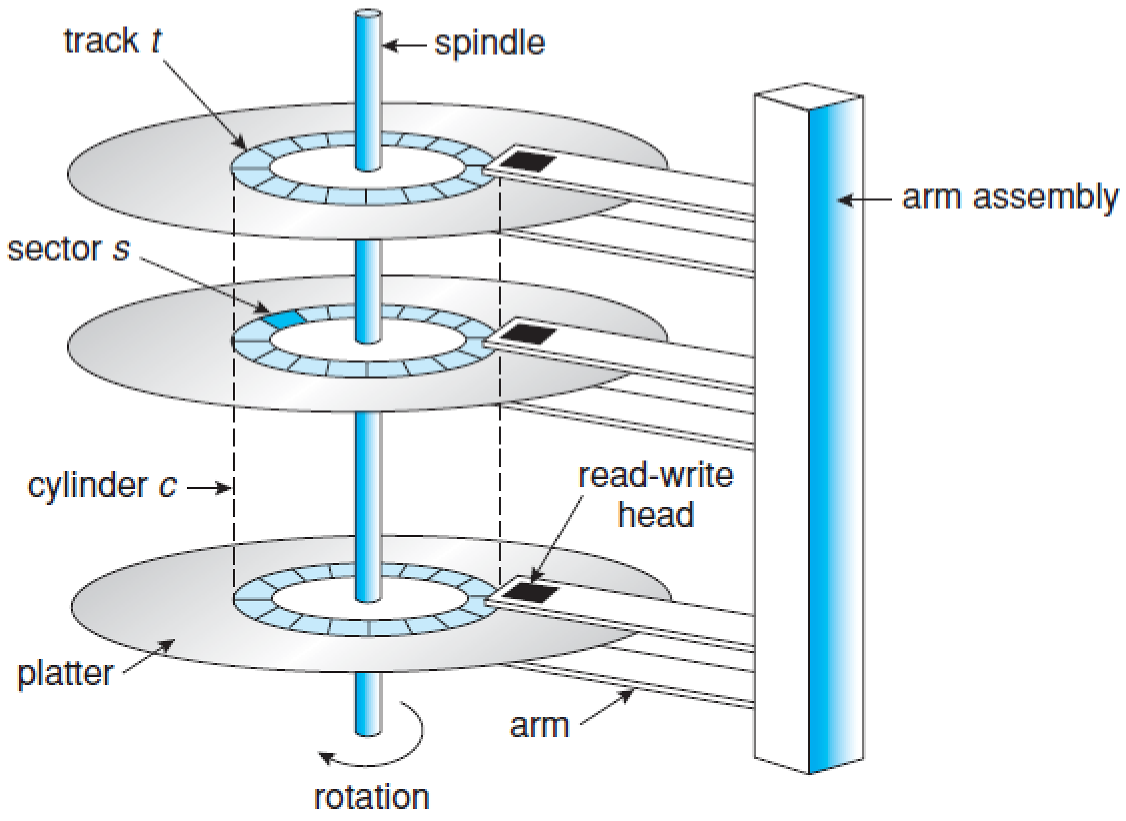


Figure 10.1 Moving-head disk mechanism.

- A read–write head "flies" just above each surface of every platter. The heads are attached to a **disk arm** that moves all the heads as a unit. The surface of a platter is logically divided into circular **tracks**, which are subdivided into **sectors**. The set of tracks that are at one arm position makes up a **cylinder**.
- There may be thousands of concentric cylinders in a disk drive, and each track may contain hundreds of sectors. The storage capacity of common disk drives is measured in gigabytes.
- When the disk is in use, a drive motor spins it at high speed. Most drives rotate 60 to 250 times per second, specified in

terms of rotations per minute **(RPM)**. Common drives spin at 5400, 7200, 10000, and 15000 RPM.

- Disk speed has two parts.
    - The **transfer rate** is the rate at which data flow between the drive and the computer.
    - The **positioning time**, or **random-access time**, consists of again two parts:
        - The time necessary to move the disk arm to the desired cylinder, called the **seek time**
        - The time necessary for the desired sector to rotate to the disk head, called the **rotational latency**.
- Typical disks can transfer several megabytes of data per second, and they have seek times and rotational latencies of several milliseconds.
- Because the disk head flies on an extremely thin cushion of air (measured in microns), there is a danger that the head will make contact with the disk surface
- Although the disk platters are coated with a thin protective layer, the head will sometimes damage the magnetic surface. This accident is called a **head crash**. A head crash normally cannot be repaired; the entire disk must be replaced.
- A disk can be **removable**, allowing different disks to be mounted as needed. Removable magnetic disks generally consist of one platter, held in a plastic case to prevent damage while not in the disk drive.

- Other forms of removable disks include CDs, DVDs, and Blu-ray discs as well as removable flash-memory devices known as **flash drives** (which are a type of solid-state drive).
- A disk drive is attached to a computer by a set of wires called an **I/O bus**. Several kinds of buses are available, including **advanced technology attachment (ATA)**, **serial ATA (SATA)**, **universal serial bus (USB)**, and **fibre channel (FC)**.
- The data transfers on a bus are carried out by special electronic processors called **controllers**. The **host controller** is the controller at the computer end of the bus. A **disk controller** is built into each disk drive.
- To perform a disk I/O operation, the computer places a command into the host controller. The host controller then sends the command via messages to the disk controller, and the disk controller operates the disk-drive hardware to carry out the command.
- Disk controllers usually have a built-in cache. Data transfer at the disk drive happens between the cache and the disk surface, and data transfer to the host, at fast electronic speeds, occurs between the cache and the host controller.

SOLID STATE DISKS (SSD)

- SSD is non-volatile memory that is used like a hard drive. There are many variations of this technology, from DRAM with a battery to allow it to maintain its state in a power failure through flash-memory technologies

- SSDs have the same characteristics as traditional hard disks but can be more reliable because they have no moving parts and faster because they have no seek time or latency. In addition, they consume less power.
- However, they are more expensive per megabyte than traditional hard disks, have less capacity than the larger hard disks, and may have shorter life spans than hard disks, so their uses are somewhat limited.
- One use for SSDs is in storage arrays, where they hold file-system metadata that require high performance.
- SSDs are also used in some laptop computers to make them smaller, faster, and more energy-efficient. Because SSDs can be much faster than magnetic disk drives.
- Some SSDs are designed to connect directly to the system bus.
- Some systems use SSD as a direct replacement for disk drives, while others use them as a new cache tier, moving data between magnetic disks, SSDs, and memory to optimize performance.

DISK STRUCTURE

- Modern magnetic disk drives are addressed as large one-dimensional arrays of **logical blocks**, where the logical block is the smallest unit of transfer.

- The one-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially. Sector 0 is the first sector of the first track on the outermost cylinder. The mapping proceeds in order through that track, then through the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.
- Converting a logical block number into an old-style disk address consists of a cylinder number, a track number within that cylinder, and a sector number within that track.
- In practice, it is difficult to perform this translation, for two reasons.
    - ➢ Most disks have some defective sectors, but the mapping hides this by substituting spare sectors from elsewhere on the disk.
    - ➢ The number of sectors per track is not a constant on some drives.
- Let’s look more closely at the second reason.
- On media that use **constant linear velocity (CLV)**, the density of bits per track is uniform.
- The farther a track is from the center of the disk, the greater its length, so the more sectors it can hold. As we move from outer zones to inner zones, the number of sectors per track decreases. Tracks in the outermost zone typically hold 40 percent more sectors than do tracks in the innermost zone. The drive increases its rotation speed as the head moves from the outer to the inner tracks to keep the same rate of data

moving under the head. This method is used in CD-ROM and DVD-ROM drives.

- Alternatively, the disk rotation speed can stay constant; in this case, the density of bits decreases from inner tracks to outer tracks to keep the data rate constant. This method is used in hard disks and is known as **constant angular velocity (CAV)**.
- The number of sectors per track has been increasing as disk technology improves, and the outer zone of a disk usually has several hundred sectors per track.
- Similarly, the number of cylinders per disk has been increasing; large disks have tens of thousands of cylinders.