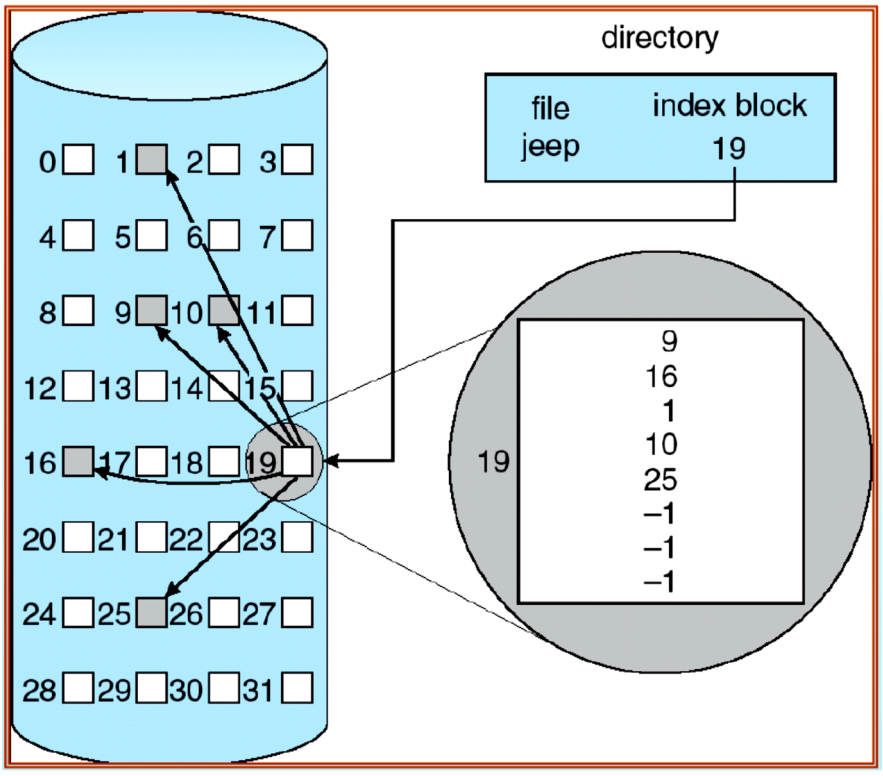
Chapter 4
File System Implementation

This chapter will discuss the following concepts:

4.1 File-System Implementation
4.2 Directory Implementation
4.3 Allocation Methods
4.4 Free-Space Management
4.5 Recovery



4.1 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. Here 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.

- **A volume control block** (per volume) contains volume (or partition) details, such as the number of blocks in the partition, size of the blocks, free block count and free-block pointers, and free **file control block (FCB)** count and FCB pointers.
- A directory structure per file system is used to organize the files.

4.2 Directory implementation

4.2.1 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. **To create** a new file, we must first search the directory to be sure that no existing file has the same

name. Then, we add a new entry at the end of the directory. **To delete** a file, we search the directory for the named file, then release the space allocated to it.

The real disadvantage of a linear list of directory entries is that finding a file requires a **linear search**. Directory information is used frequently, and users will notice if access to it is slow. In fact, many operating systems implement a software **cache** to store the most recently used directory information. A cache hit avoids the need to constantly reread the information from disk. A sorted list allows a **binary search** and decreases the average search time.

4.2.2 Hash Table

Another data structure used for a file directory is a **hash table**. With 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 deletion are also fairly straightforward.

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 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.

4.3 Allocation Methods

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

4.3.1 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. With this ordering, assuming that only one job is accessing 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** (in block units). 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 (Figure 4.1).

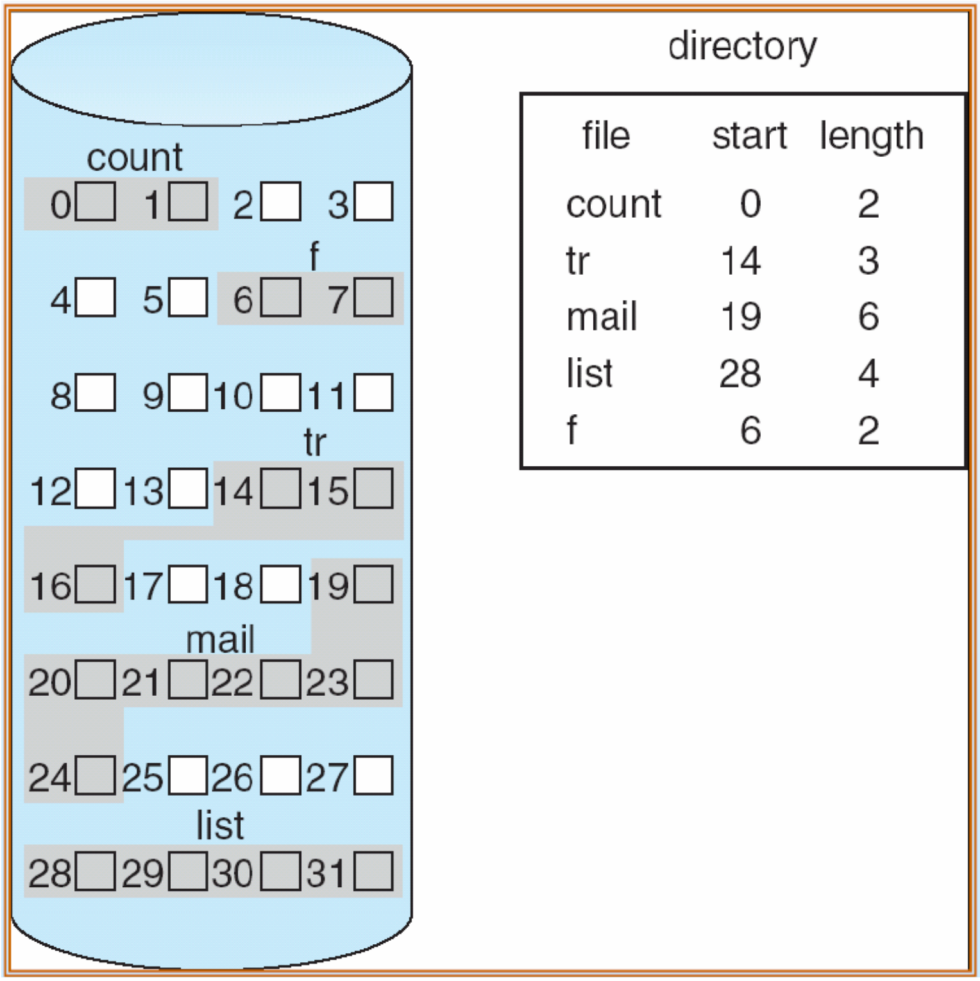


Figure 4.1 Contiguous allocation of disk space.

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, 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.

Contiguous allocation has some problems, however. One difficulty is finding space for a new file. Another problem with contiguous allocation is determining how much space is needed for a file.

4.3.2 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 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 (Figure 4.2). Each block contains a pointer to the next block. These pointers are not made available to the user. Thus, if each block is 512 bytes in size, and a disk address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes.

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

Linked allocation does have disadvantages, however. The major problem is that it can be used effectively **only for sequential-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.

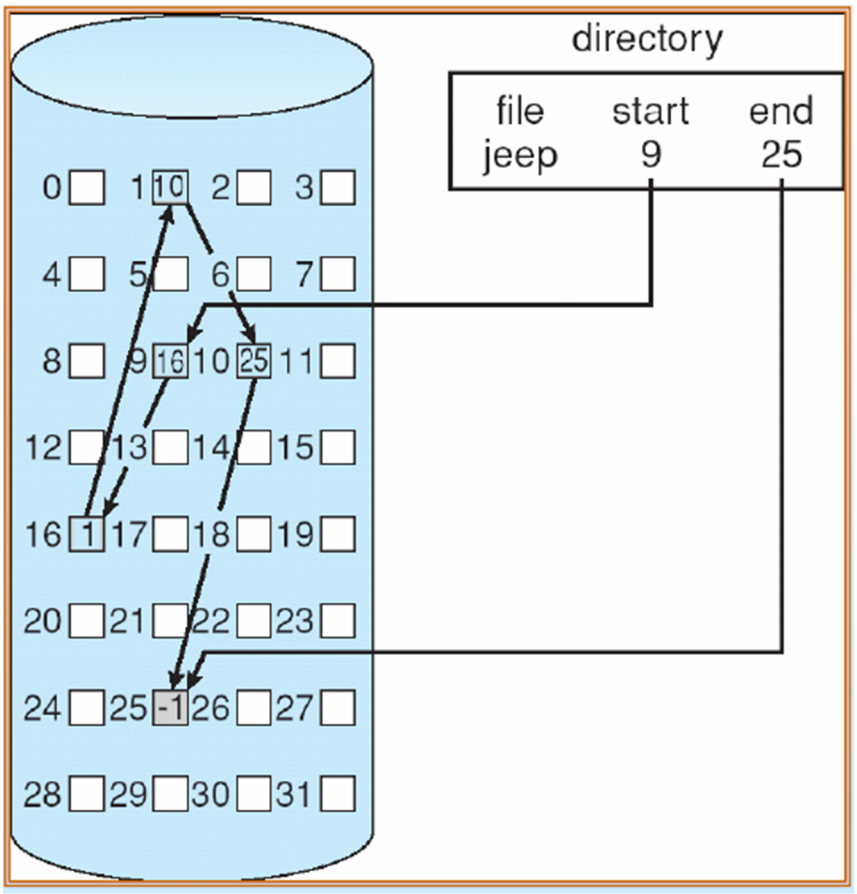


Figure 4.2 Linked allocation of disk space.

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.

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 operating-system 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 is used by the MS-DOS and OS/2 operating systems. 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 the last block, which has a special end-of-file value as the table entry. Unused blocks are indicated by a 0 table value. Allocating a new block to a file is a simple matter of finding the first 0-valued table entry and replacing the previous end-of-file value with the address of the new block. The 0 is then replaced with the end-of-file value. An illustrative example is the FAT structure shown in Figure 4.3 for a file consisting of disk blocks 217, 618, and 339.

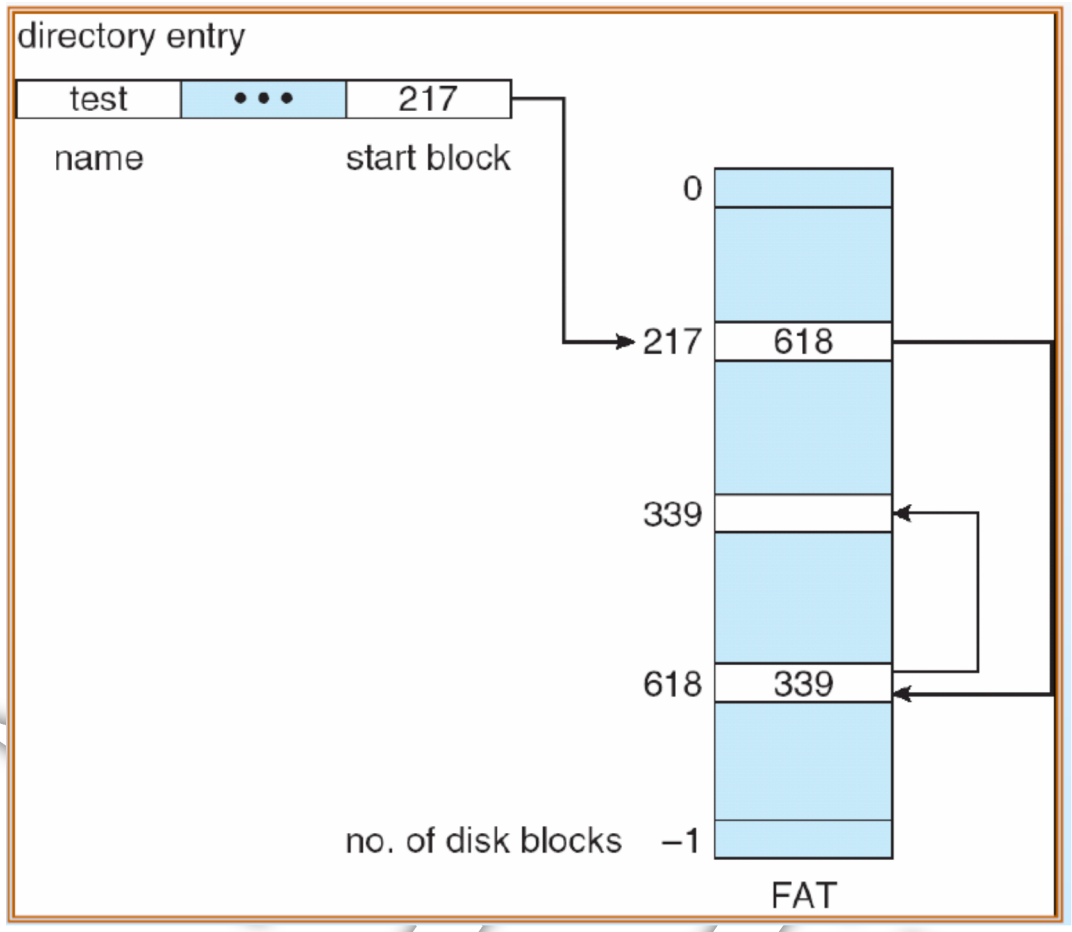


Figure 4.3 File-allocation table.

4.3.3 Indexed Allocation

Linked allocation solves the external-fragmentation and size-declaration problems of contiguous 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 into one location: 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 i_{th} block of the

file. The directory contains the address of the index block (Figure 4.4). To find and read the i_{th} block, we use the pointer in the i_{th} index-block entry.

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

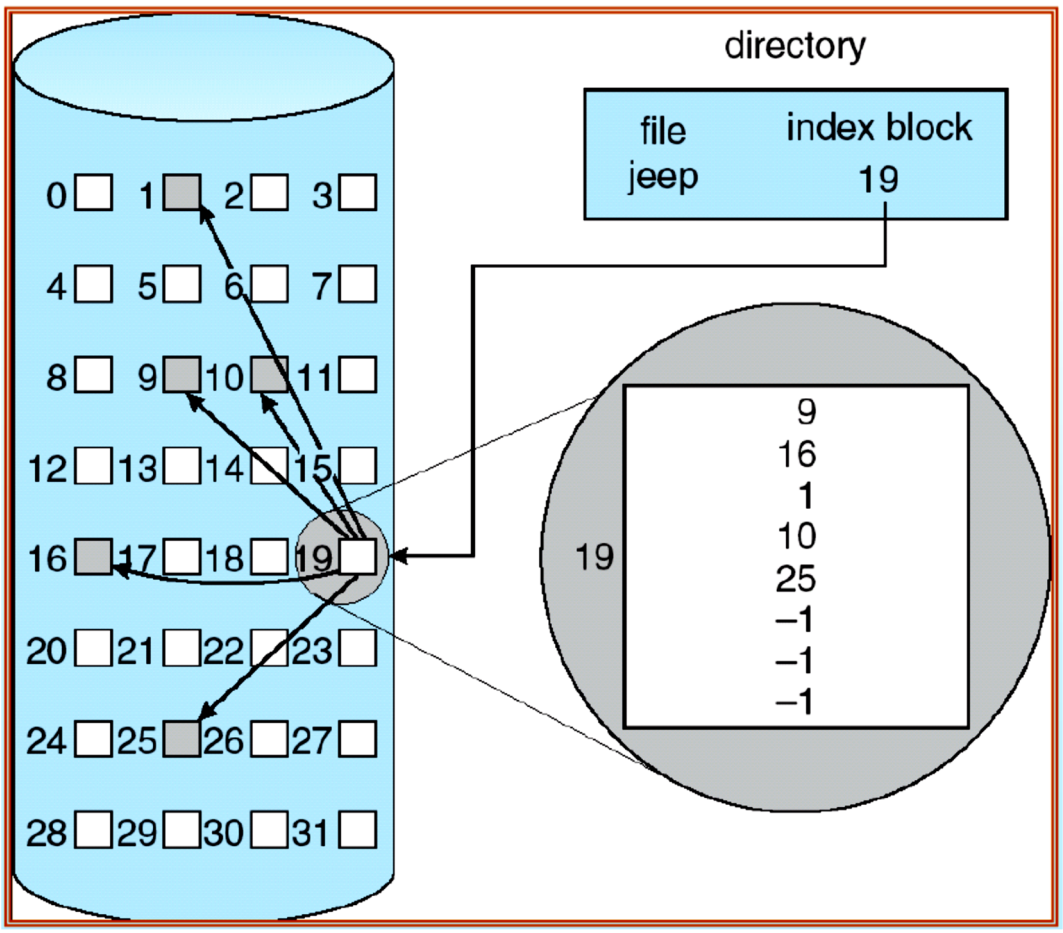


Figure 4.4 Indexed allocation of disk space.

4.4 Free-Space Management

Since disk space is limited, we need to reuse the space from deleted files for new files, if possible. (Write-once optical disks only allow one write to any given sector, and thus such reuse is not physically 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. The free-space list, despite its name, might not be implemented as a list, as we discuss next.

4.4.1 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 00111100111111000110000011100000 ...

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, indeed, many computers supply bit-manipulation instructions that can be used effectively for that purpose.

4.4.2 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 earlier 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 (Figure 4.5). However; this scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/O time.

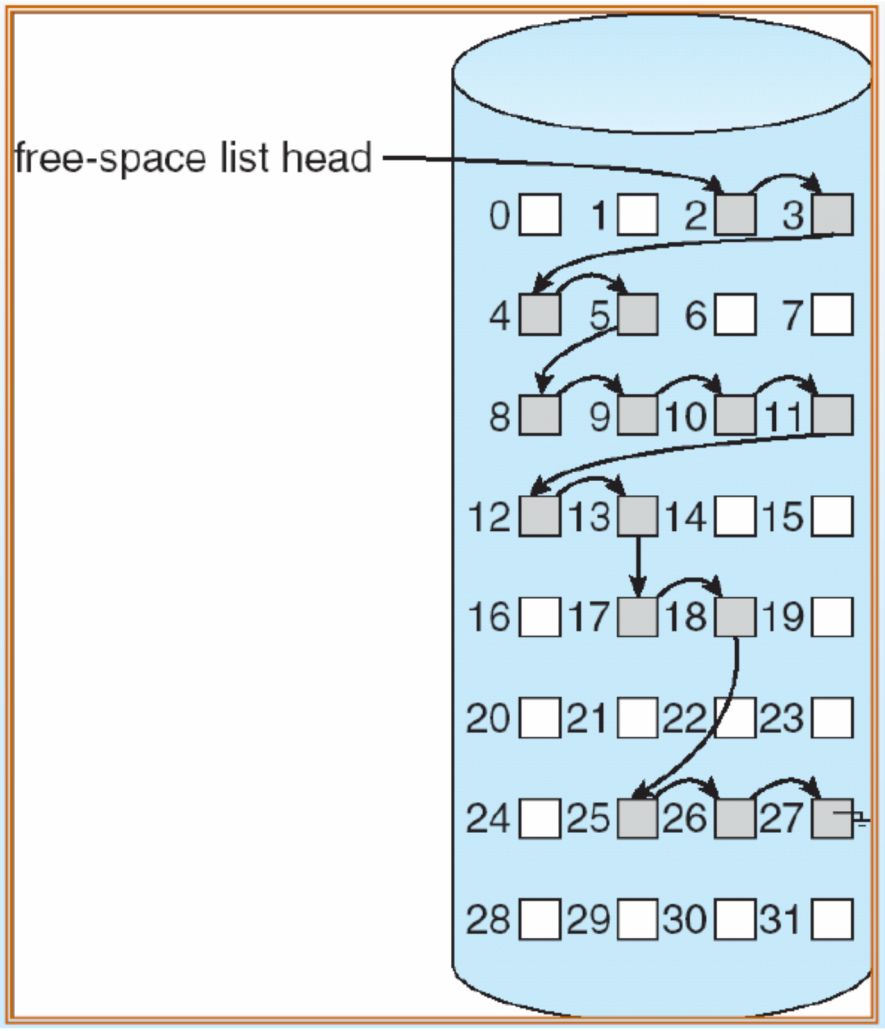


Figure 4.5 Linked free-space list on disk.

4.4.3 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 other n free blocks, and so on. The addresses of a large number of free blocks can now be

found quickly, unlike the situation when the standard linked-list approach is used.

4.4.4 Counting

Another approach is to take advantage of the fact that, generally, several contiguous blocks may be allocated or freed simultaneously, particularly when space is allocated with the contiguous-allocation algorithm 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**. 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.

4.5 Recovery

4.5.1 Consistency Checking

Some directory information is kept in main memory (or cache) to speed up access. 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.

Consider, then, the possible effect of a computer crash. Cache and buffer contents, as well as I/O operations in progress, can be lost, and with them any changes in the directories of opened files. Such an event

can leave the file system in an **inconsistent state**: The actual state of some files is not as described in the directory structure. Frequently, a special program is run at reboot time to check for and correct disk inconsistencies.

4.5.2 **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.

End of chapter 4