## File-System Structure

- Disks provide most of the secondary storage on which file systems are maintained. Two characteristics make them convenient for this purpose:

  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

- The file system itself is generally composed of many different levels.
- The **I/O control** level 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/O device to the rest of the system.

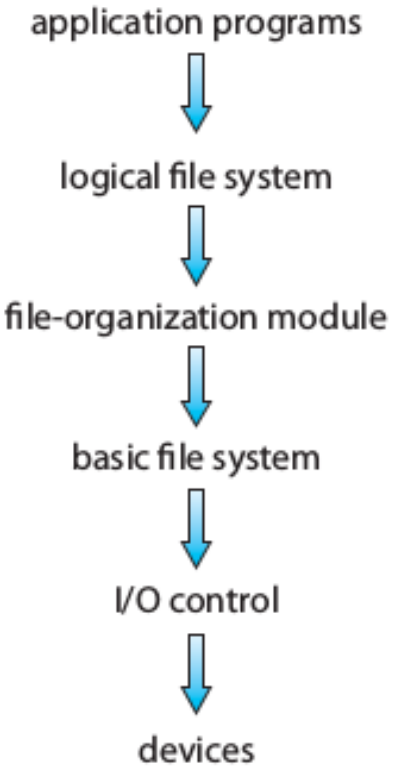


**Figure 12.1** Layered file system.

- The **basic file system** needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk.
- Each physical block is identified by its numeric disk address (for example, drive 1,  cylinder 73, track 2, sector 10).
- 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.

- 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.
- 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.

- Finally, the **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. A **file- control block ( FCB )** contains information about the file, including ownership, permissions, and location of the file contents.

## Allocation Methods

- Three major methods of allocating disk space are in wide use: contiguous, linked, and indexed.

  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.
- It 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.
- One difficulty is finding space for a new file.
- 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.

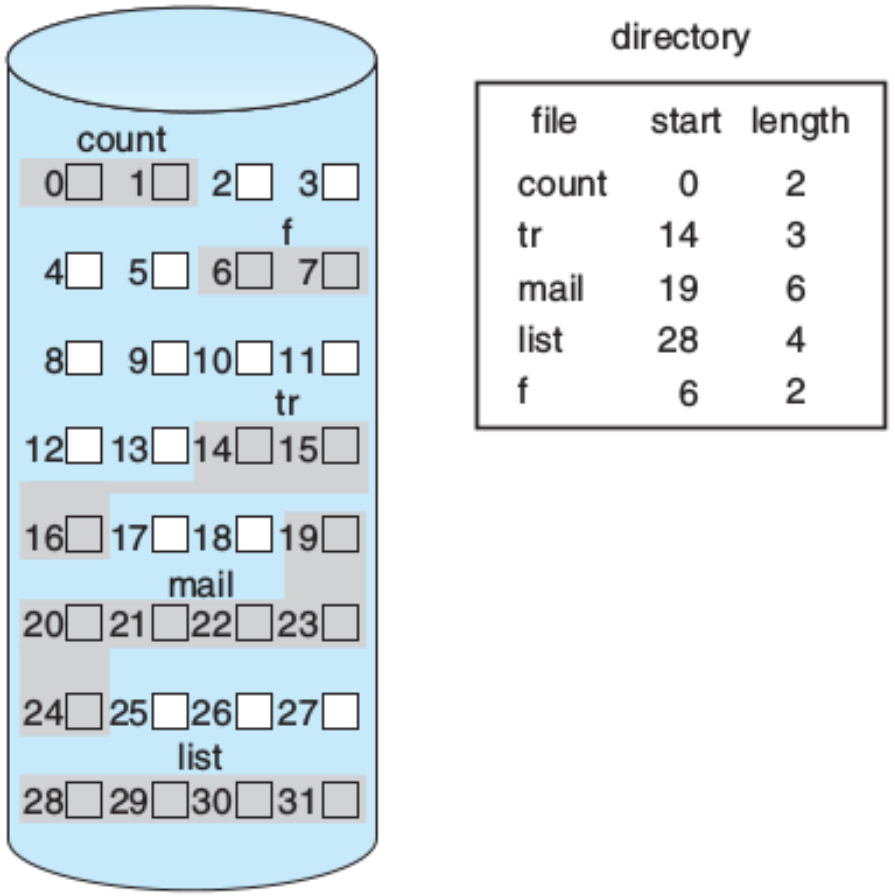


**Figure 12.5** Contiguous allocation of disk space.

- **Compaction**: Copy an entire file system onto another disk. 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. The cost of this can be particularly high for large hard disks.

- Another problem is 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.
- The other possibility is to find a larger hole, copy the contents of the file to the new space, and release the previous space.
- Even if the total amount of space needed for a file is known in advance, preallocation 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.

## 2. Linked 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. 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. The pointer is initialized to null (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 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.
- 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.
- Another disadvantage is the space required for the pointers. 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. Pointers then use a much smaller percentage of the file’s disk space.
- The cost of this approach is an increase in internal fragmentation, because more space is wasted when a cluster is partially full than when a block is partially full.
- If a pointer were lost or damaged, it might result in picking up the wrong pointer.

## File-allocation table ( FAT )

- The table has one entry for each disk block and is indexed by block number.
- 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.
- An unused block is indicated by a table value of 0. 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

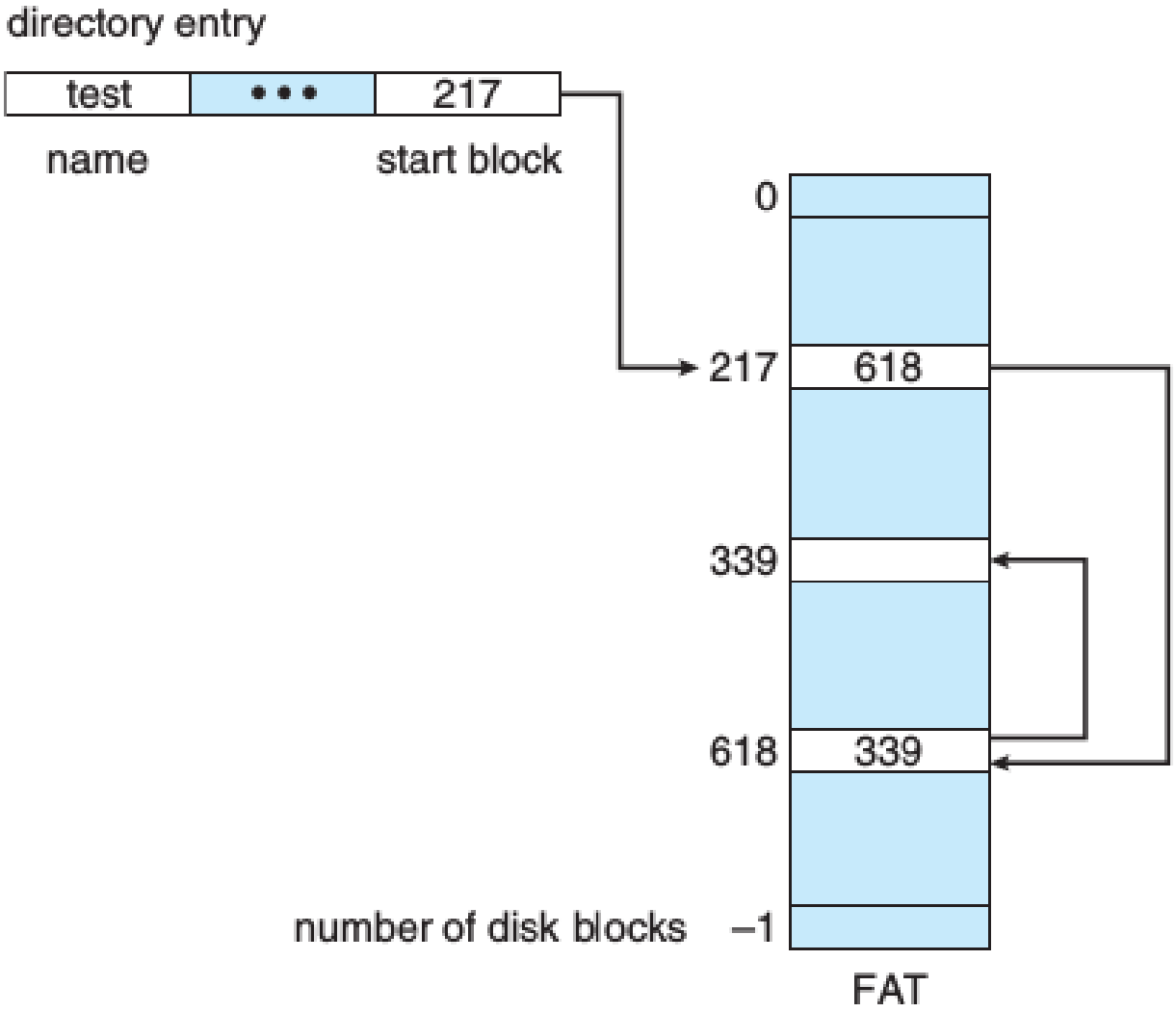


**Figure 12.7** File-allocation table.

## 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 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 i th index-block entry.

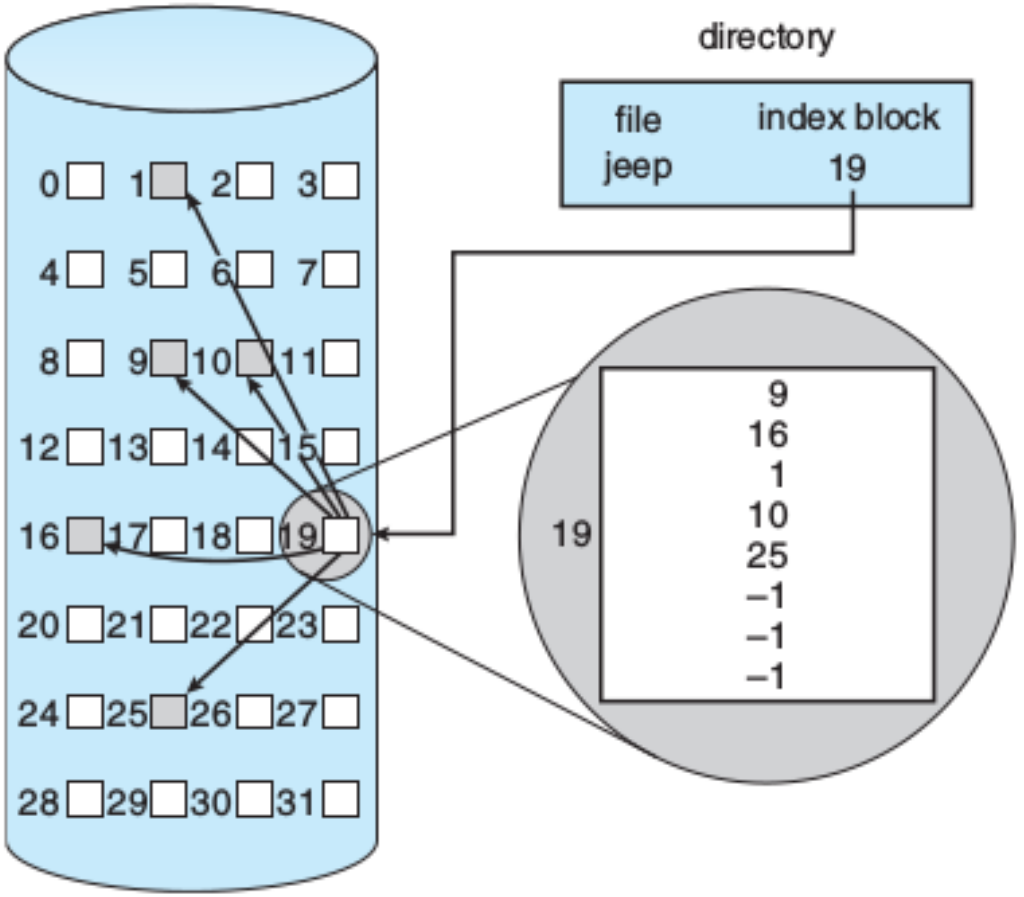


**Figure 12.8** Indexed allocation of disk space.

- When the file is created, all pointers in the index block are set to null. When the i th 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.

- Indexed allocation does suffer from wasted space, however. 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.
- How large the index block should be?

  1. **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.

  2. **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.

3. **Combined scheme**. Another alternative, used in UNIX -based file systems, 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.
- 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.

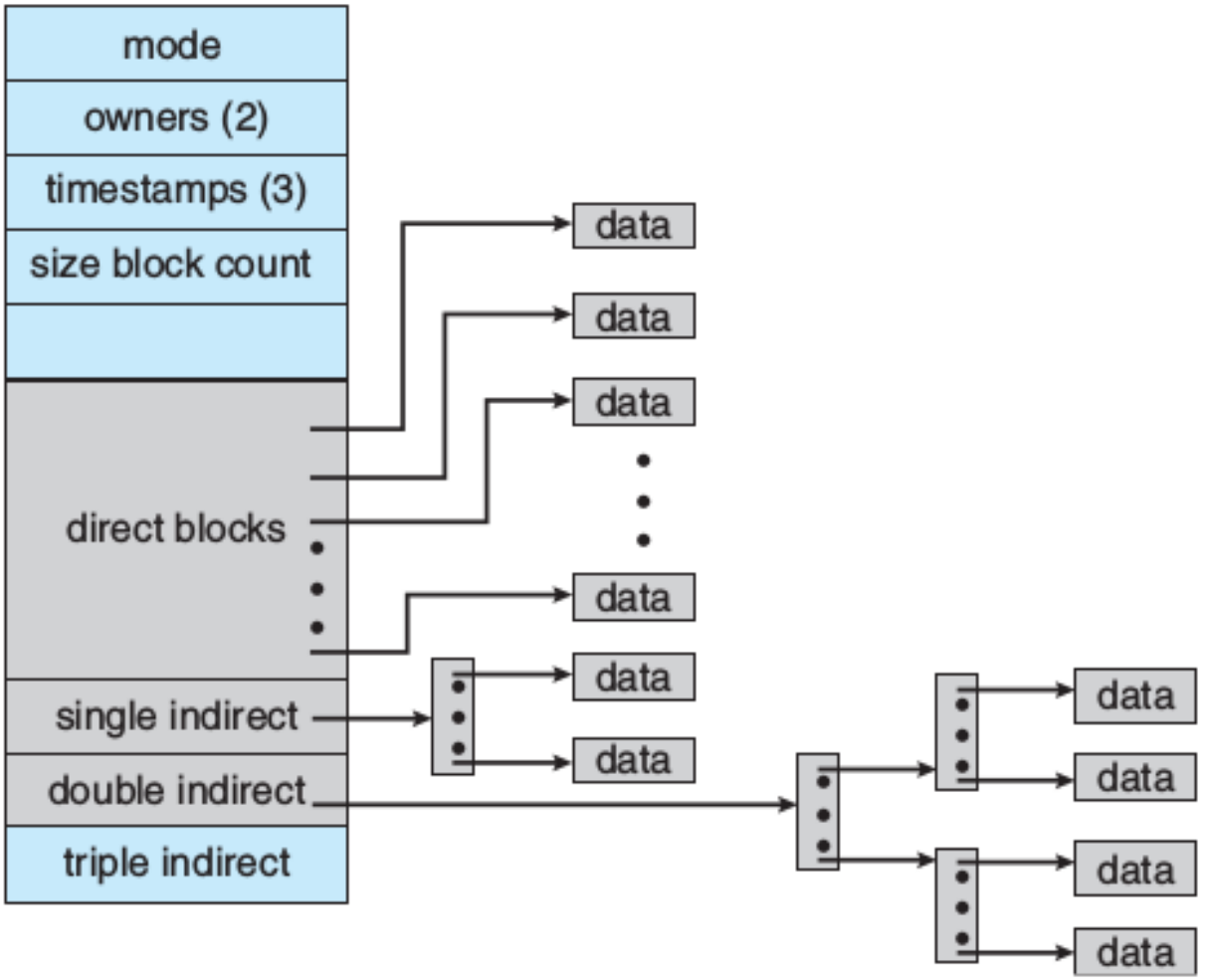


**Figure 12.9** The UNIX inode.