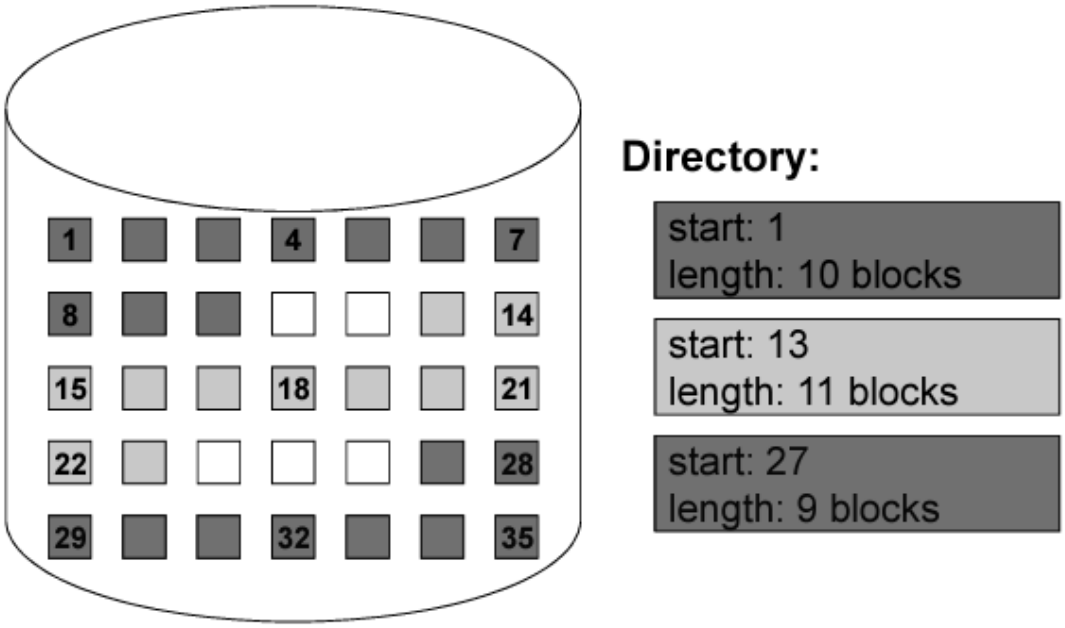
File system — physical layer

Allocation methods

- There are different ways to implement files in the file system.
- The main problem is how to allocate space to these files so that storage space is utilized effectively and files can be accessed quickly.
- There are three general methods for allocation:

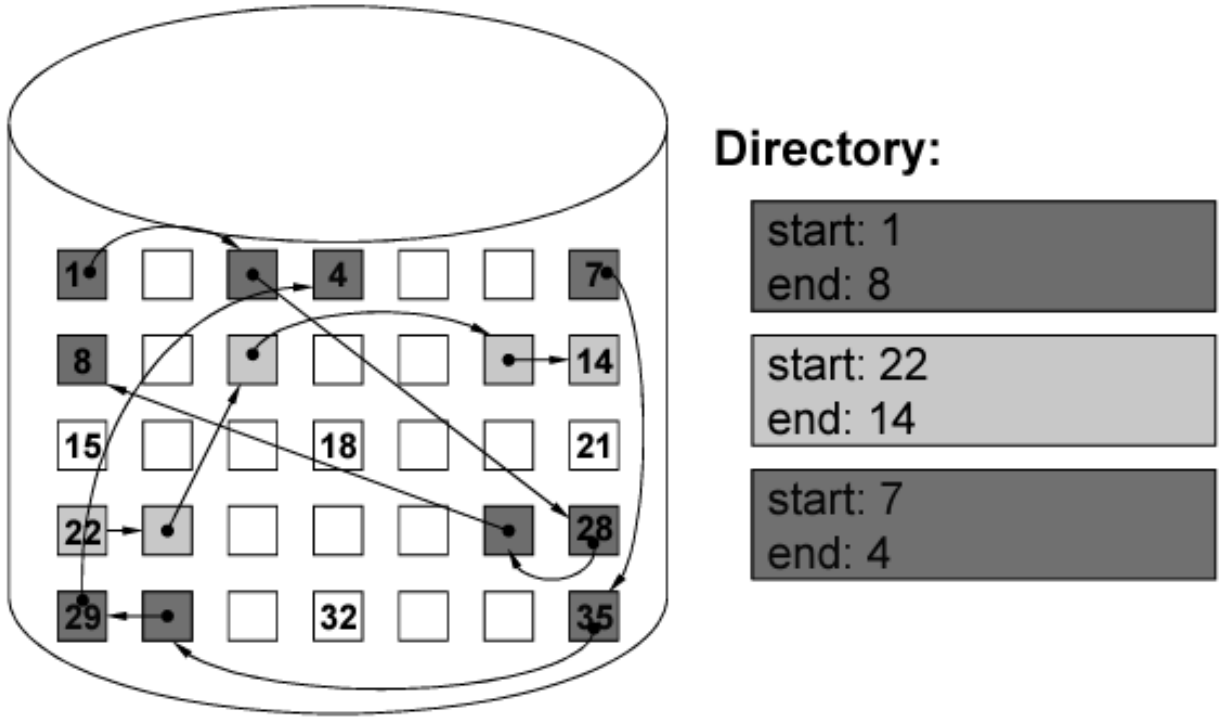
Contiguous allocation

- Contiguous allocation requires that each file occupy a set of contiguous blocks on the device.
- This method is really fast since the HDD head doesn't need to move far.
- Contiguous allocation of a file is defined by the address of the first block and length (in block units) of the file.
- The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.
- This method is easy to implement, however it has many problem, thus it is not used in practice. One such problem is how to find space for a new file. Another problem is that we need to reallocate a file if we want to enlarge it, but there is not enough space.
- External fragmentation — a problem with contiguous allocation when files are reallocated or deleted and our storage has many small holes of free space that is wasted.



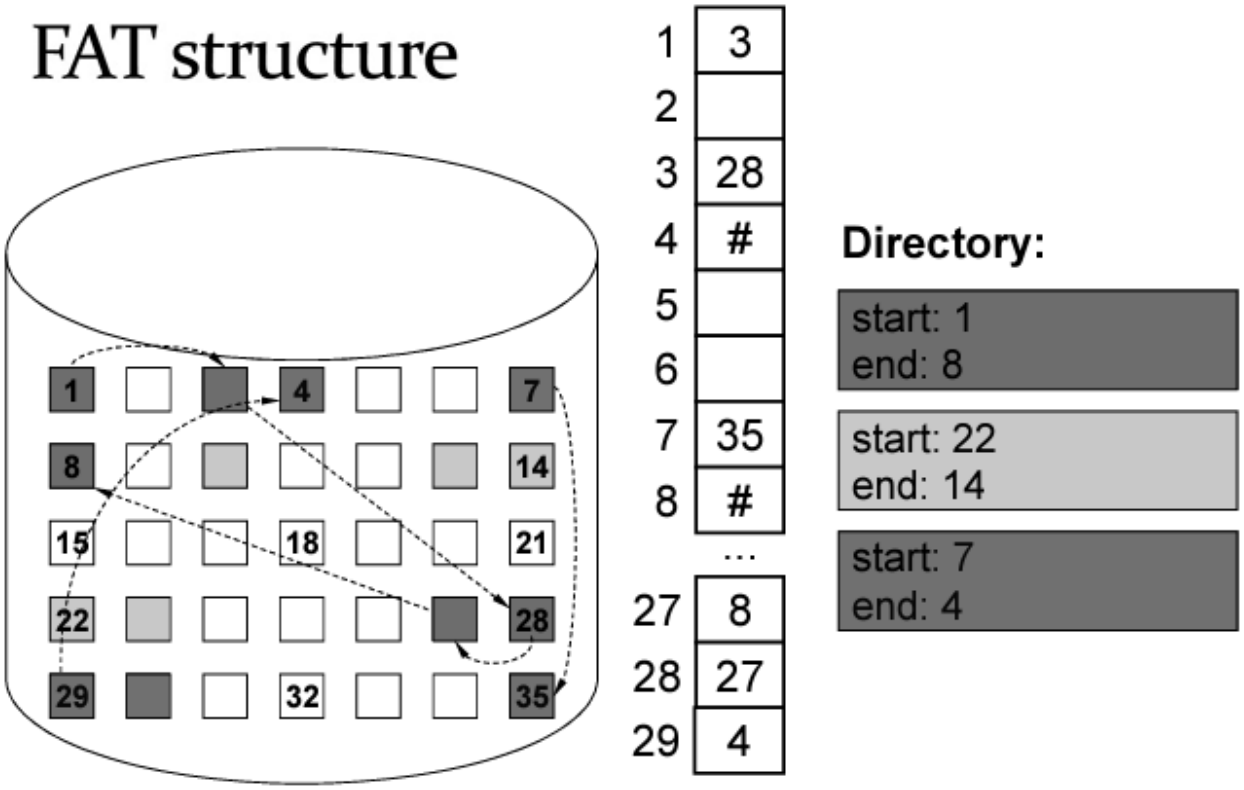
Linked (chained) allocation

- With linked allocation, each file is a linked list of storage blocks; the blocks may be scattered anywhere on the device.
- The directory contains a pointer to the first and last blocks of the file.
- Each block contains a pointer to the next block (thus a block usually has 4 bytes less available, because the pointer takes up 4 bytes).
- There is no external fragmentation with linked allocation, however it is not suitable for direct access, only for sequential access.
- There is also a reliability issue — losing a block means losing all subsequent blocks.



File Allocation Table (FAT)

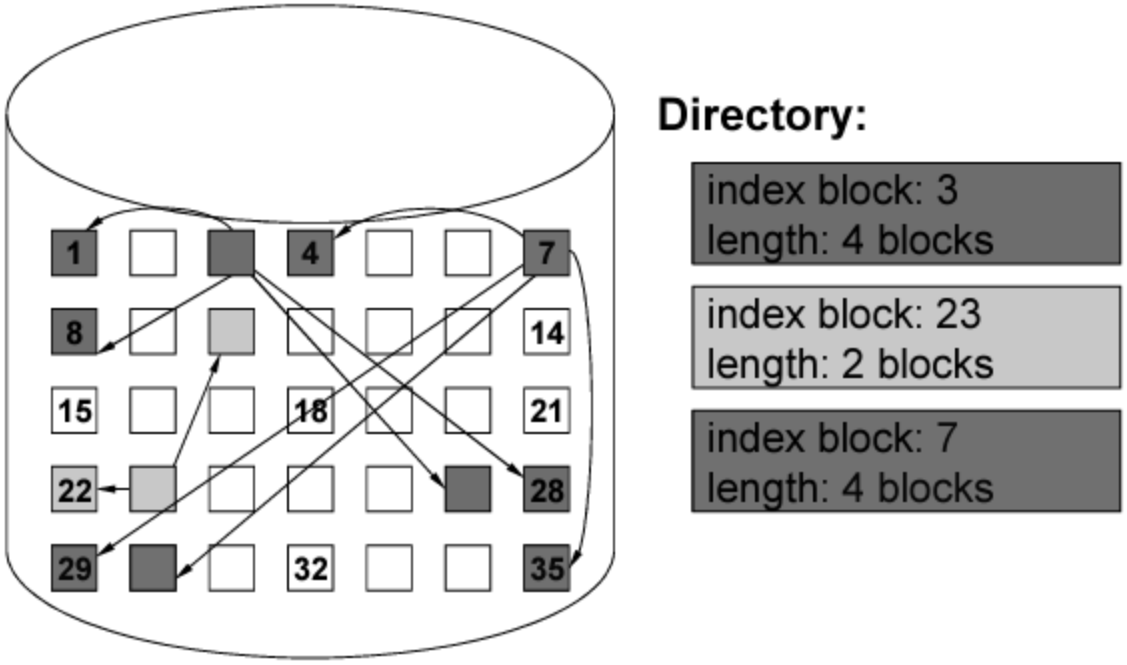
- It is a variation on linked allocation.
- A section of storage at the beginning of each volume is set aside to contain the file allocation table.
- The table has one entry for each block and is indexed by block number. The pointers to the next blocks are saved in this table, instead of the end of each block.
- The FAT entry contains the number of the next block in the chain or a special value denoting either a free block, corrupted sector, or the end of the chain.
- The FAT table must be cached, otherwise the number of disk head movements is significant. But this method improves the random-access time from the linked allocation method.



Indexed allocation

- Indexed allocation solves the problem of inefficient direct access of linked allocation with no FAT table.
- Each file has its own index block, which is an array of storage-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 and the length of the file in blocks.
- When the file is created, all pointers in the index block are set to null. 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. However it is wasteful when a file only needs a few blocks, because we always need to allocate a whole block for the indices.

Indexed allocation



- To minimize the waste of allocating the whole index block when we only need a few blocks of memory we usually allocate as little space for the index block as possible. Then when we want to expand our index block we can use a few different techniques:
  - Linked scheme:

    The index block might contain a header with the name of the file, a list of addresses to the blocks of memory, and then the last entry in the index block is either null or it is a pointer to another index block (for large files).
  - Multilevel index:

    The first-level index block holds a list of addresses to the second-level index blocks which can either hold the lists of addresses to the file blocks, or addresses to another level of index blocks. With 4096 byte blocks we could store 1024 pointers in an index block, which means with two levels of index blocks we could store a file of up to 4GB.
  - Combined scheme:

This alternative is used in UNIX-based file systems. This approach combines the previous schemes by storing different types of pointers along with some metadata in the file's inode. An inode is a block of memory that, like in previous schemes stores some metadata for the file, as well as pointers to first memory blocks. In this scheme the inode stores four different different types of pointers along with the metadata:

- Pointer to direct blocks: they point directly to the first few data blocks of the file.
- Pointer to single indirect blocks: it points to blocks that contain lists of addresses to the next data blocks.
- Pointer to double indirect blocks: it points to blocks that contain lists of addresses to blocks that contain lists of addresses to the next data blocks.
- Pointer to triple indirect blocks: it points to blocks that contain lists of addresses to blocks that contain lists of addresses to blocks that contain lists of addresses to the next data blocks.

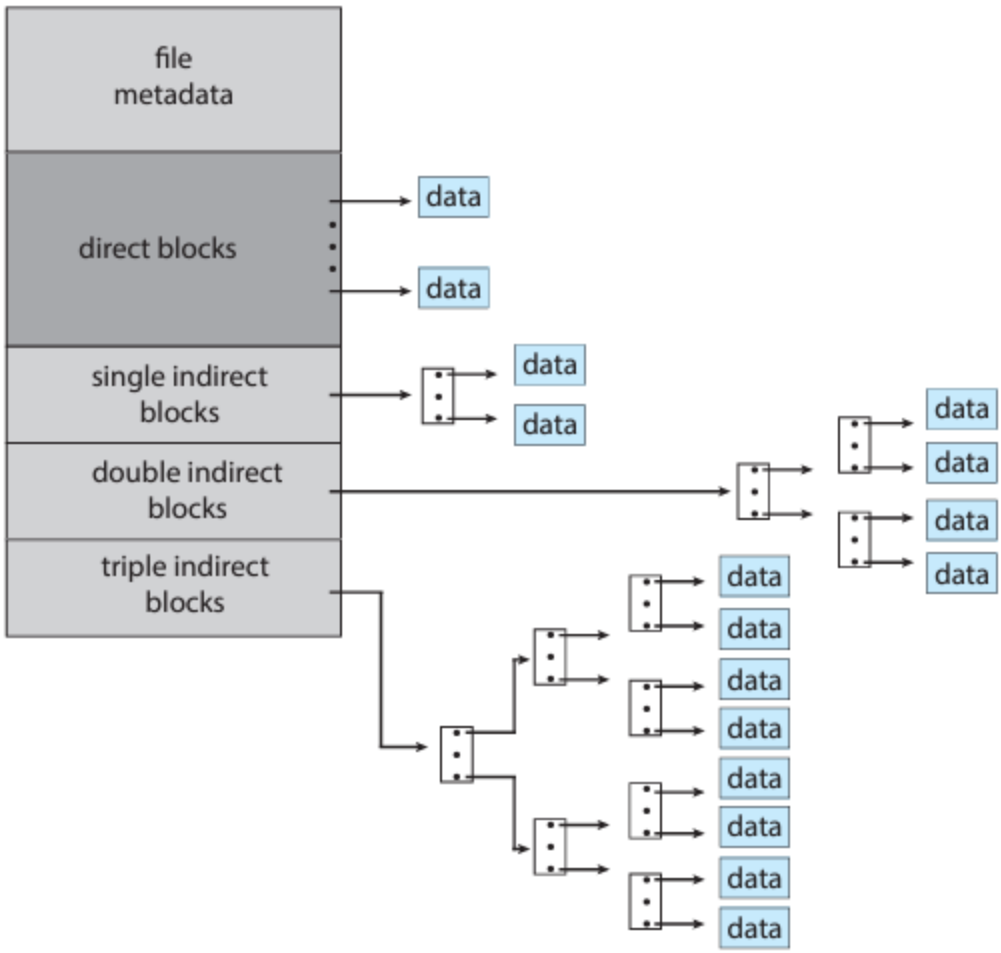


Figure 14.8 The UNIX inode.

- Indexed allocation is not as efficient as contiguous allocation, but still fast. It doesn't have the external fragmentation problem, but it does have an reliability issue, since losing an index block means losing a couple of data blocks.

Free space management

- Since storage space is limited, we need to reuse the space from deleted files for new files, if possible.
- To keep track of free disk space, the system maintains a free-space list.
- 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 space is added to the free-space list.

- There are a few approaches for implementing the free-space list:

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.
- This approach is relatively simple and efficient if we store the bit vector in the main memory, and save it on the disk only for recovery. It is not possible to implement this approach for bigger devices that have a lot of memory.
- Hardware allows to effectively find free space in the bit vector by finding the first non-0 valued word and then finding the first non-0 bit.

Linked list

- Another approach to free-space management is to link together all the free blocks, keeping a pointer to the first free block in a special location in the file system and caching it in memory.
- This first block contains a pointer to the next free block, and so on.
- This scheme is not efficient, to traverse the list, we must read each block of secondary storage which is time consuming.

Grouping

- The first free block stores the addresses to the first n free blocks. The first $n-1$ of there blocks are actually free. The last block contains the addresses to another n free blocks, and so on.

Counting

- This approach takes advantage of the fact that usually there are several free contiguous blocks, instead of just one.
- Thus, in this method we store an address of the first block, along with the length of the free space segment starting in that block.

Directory implementation

- There are several ways to implement directory allocation and directory management, they all have different trade-offs.

Linear list (table)

- The simplest method of implementing a directory is to use a linear list of file names with other attributes.
- 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 can either change it’s name to all blank, assign an invalid inode number to it, or store a special bit that says whether this entry is used/unused.
- Linked list can also be used to speed up file deletion.
- The main disadvantage of this method is that finding a file takes linear time.
- To reduce the search time, the entries may be sorted, however, retaining the list sorted generates a cost.

Hash table

- Here, 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, 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.

Indexing structure

- A search tree is used to organise the directory entries (e.g. binary tree, B-tree, B+-tree).
- Following the rules of search tree organisation, the entries can be found more efficiently than in the case of linear list.
- The search tree structure is usually optimised to reduce the number of transfers between disk and memory.

Disk caching

- Memory access is much faster than disk access, thus many optimization techniques have been developed.
- In this context, a cache is a collection of blocks that logically belong on the disk but are being kept in memory for performance reasons. The memory area used for this purpose is called the buffer cache.
- Various algorithms can be used to manage the cache, but a common one is to check all read requests to see if the needed block is in the cache. If it is, the read request can be satisfied without a disk access. If the block is not in the cache, it is first read into the cache and then copied to wherever it is needed. Subsequent requests for the same block can be satisfied from the cache.
- To handle a write request, the data to be written are transferred to the buffer cache, then to the disk (even with delay — lazy caching).
- When writing to a disk, first we check whether this block is in buffer, if it’s not then we transfer it from the disk. Then we transfer the data to be written into the buffer. After the writing to the buffer is completed we either immediately update the disk memory, or with a delay depending on the urgency of the data.

File system integrity

- Many file systems read blocks, modify them, and write them out later. If the system crashes before all the modified blocks have been written out, the file system can be left in an inconsistent state.

- To deal with inconsistent file systems, most computers have a utility program that checks file-system consistency. This utility can be run whenever the system is booted, especially after a crash.
- This kind of utility program works by keeping two tables, each holding a count for each block in memory, initially set to 0. Then this program for each file increases the use count of each block in the first table. After this is goes through the list of free blocks and also increases the use count for each block, this time in the second table. If the file system is consistent, then each block will have a 1 in either table, but if it has a 1 in both tables, or any other number then this block is violated. There are a few cases of integrity violation:
    - Missing block: in this case a block doesn't appear in any of the tables. The simple solution is to add this block into the free list
    - Duplication of a free block: a block has a number greater than 1 in the second table. The solution is to rebuild the free list
    - Dual appearance of a block — a block is found both allocated and free.
    - Duplication of an allocated block — a block is found to be allocated more than once (it belongs to different files)
    - Inconsistency in directory entries.