18 File Structure, Disk Scheduling

File: a named collection of bytes stored on disk. From the OS' standpoint, the file consists of a bunch of blocks stored on the device. Programmer may actually see a different interface (e.g., records), but this doesn't matter to the file system (just pack bytes into disk blocks, unpack them again on reading).

Common addressing patterns:

- Sequential: information is processed in order, one piece after the other. This is a common mode: e.g. editor writes out new file, compiler compiles it, etc.
- Random Access: can address any block in the file directly without passing through its predecessors. E.g. the data set for demand paging, databases.
- Content based: search for blocks with particular values, e.g. hash table, associative database, dictionary. Not provided by all operating system.

Modern file systems must address several general problems:

- Disk Management: efficient use of disk space, fast access to files, sharing of space between several users.
- Naming: how do users select files?
- Protection: all users are not equal.
- Persistence: information must last safely for long periods of time.
  - across runs of programs.
  - across login sessions.
  - across power failures.
  - across hardware (e.g., disk) failures.
- Concurrency control: several programs/users may read/write a file at roughly the same time.

Disk Management: how should the disk sectors be used to represent the blocks of a file? The structure used to describe which sectors represent a file is called the *file descriptor*. You'll see later that the file descriptors are stored on disk along with the files.

Things to think about in designing a file structure:

- Most files are small.
- Much of the disk is allocated to large files.
- Many of the I/O operations are made to large files.

Thus, per-file cost must be low but large files must have good performance.

Contiguous allocation (also called "extent-based"): allocate files like segmented memory (give each disk sector a number from 0 up). Keep a free list of unused areas of the disk. When creating a file, make the user specify its length, allocate all the space at once. Descriptor contains location and size.

- Advantages: easy access, both sequential and random. Simple. Few seeks.
- Drawbacks: horrible fragmentation will make large files impossible. Hard to predict needs at file creation time.
- Example: IBM OS/360.

Block groups: descriptor is a table of base/size pairs ("block group" pointers). If run out of block groups pointers, set BIGFILE flag in descriptor—Each table entry now points to an indirect block group (a block group containing base/size pairs of block groups containing file data.)

- Advantages: easier to find free block groups.
- Drawbacks: when disk gets full, lots of small block groups, many seeks. External fragmentation.
- Example: DEMOS.

Linked files: keep a linked list of all free blocks. In file descriptor, just keep pointer to first block. In each block of file keep pointer to next block.

- Advantages? Files can be extended, no fragmentation problems. Sequential access is easy: just chase links.
- Drawbacks? Random access is virtually impossible. Lots of seeking, even in sequential access.

Example: TOPS-10, sort of. Alto, sort of.

File allocation table (FAT): table indexed by disk block number. each entry contains

- -1, if the block is free
- block number of next block, if block is allocated,
- -2, if block is the last block of a file.

File descriptor contains block number of first block.

- Advantages? If FAT is cached in main memory, random access is reasonable efficient.
- Drawbacks? FAT takes lots of space.

Example: MS-DOS, OS/2.

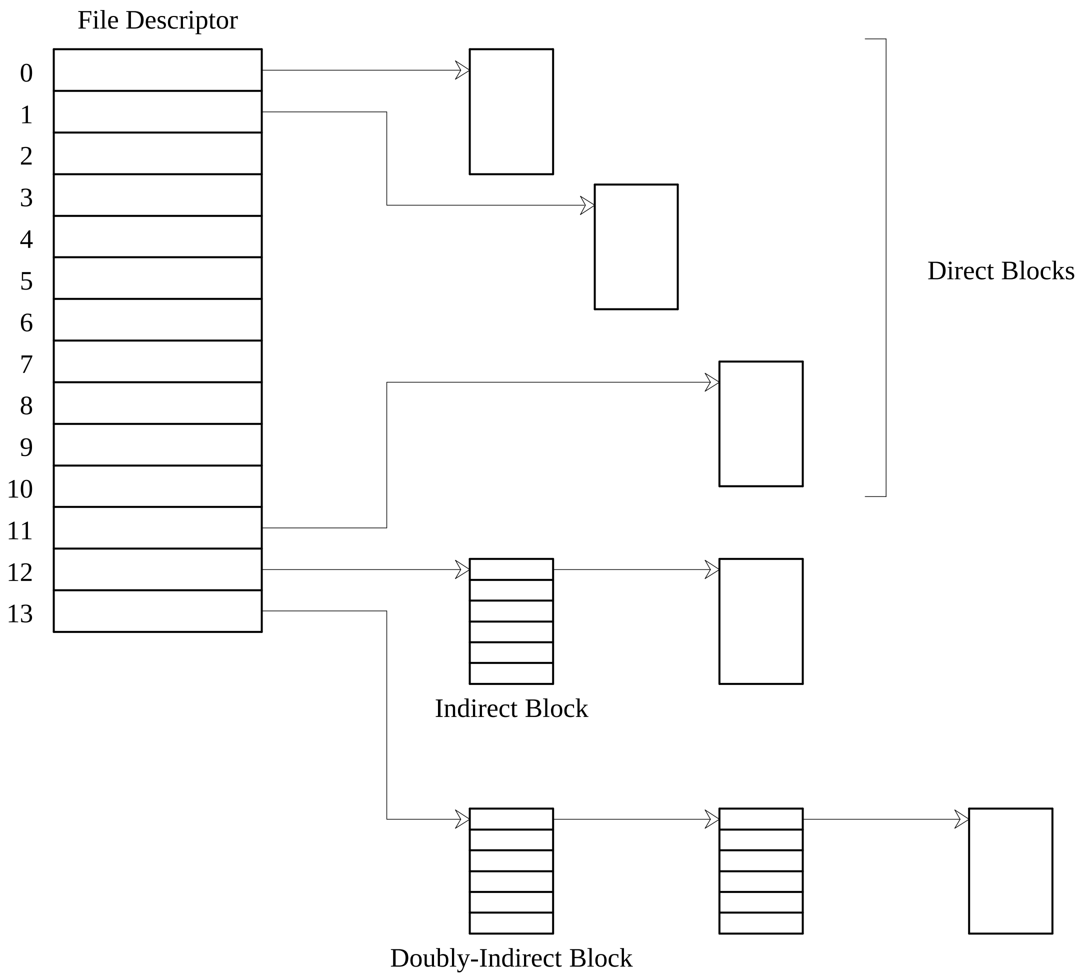
Indexed files: keep an array of block pointers for each file. File maximum length must be declared when it is created. Allocate an array to hold pointers to all the blocks, but don't allocate the blocks. Then fill in the pointers dynamically using a free list.

- Advantages? Not as much space wasted by overpredicting, both sequential and random access are easy.
- Drawbacks? Still have to set maximum file size, and there will be lots of seeks.

Multi-level indexed files: 4.3 BSD solution:

- File descriptor = 14 block pointers. The first 12 point to data blocks, the next one to an indirect block (contains 1024 more block pointers), last one to a doubly-indirect block. Maximum file length is fixed, but large. Descriptor space isn't allocated until needed.
- Examples: block 23, block 5, block 1040
- Advantages: simple, easy to implement, incremental expansion, easy access to small files.
- Potential drawbacks:
  - Indirect mechanism doesn't provide very efficient access to large files: up to 2 descriptor ops for each real operation.
  - Block-by-block organization of free list may mean that file data gets spread around the disk.

UNIX File Descriptors: Multi-Level Indexed



Block cache: a pool of recently-accessed blocks is kept in main memory. If the same blocks are referenced over and over (such as indirect blocks for large files), there's no need ever to read them from disk. This solves the problem of slow access to large files.

Keeping track of free blocks: use a *bit map*.

- Just an array of bits, one per block. 1 means block free, 0 means block allocated. For a 2.4 Gbyte drive there are about 615,000 4kbyte blocks, so bit map takes up 75 KBytes. Nowadays, can usually keep entire bit map in memory.

- During allocation, try to allocate next block of file close to previous block of file. If disk isn't full, this will usually work well.
- If disk becomes full, this becomes VERY expensive, and doesn't get much in the way of adjacency. Solution: keep a reserve (e.g. 10% of disk), and don't even tell users about the reserve. Never let the disk get more than 90% full.
- With multiple surfaces on disk, there are multiple optimal next blocks; with 10% of disk free, can almost always use one of them.

Disk scheduling: in timesharing systems, it may sometimes be the case that there are several disk I/O's requested at the same time.

First come first served (FIFO, FCFS): may result in a lot of unnecessary disk arm motion under heavy loads.

Shortest seek time first (SSTF): handle nearest request first. This can reduce arm movement and result in greater overall disk efficiency, but some requests may have to wait a long time.

Scan: like an elevator. Move arm back and forth, handling requests as they are passed. This algorithm doesn't get hung up in any one place for very long. It works well under heavy load, but not as well in the middle (about 1/2 the time it won't get the shortest seek).

Most of the time there aren't very many disk requests in the queue, so this isn't a terribly important decision.