Physical Representation of Files

A *disk drive* consists of a *disk pack* containing one or more *platters* stacked like phonograph records. Information is stored on both sides of the platter.

Each platter is divided into concentric rings called *tracks*, and each track is divided into *sectors*. All transfers to and from the disk are performed at the sector level.

For example, to modify a single byte, the system reads the sector containing the byte off the disk, modifies the byte, and rewrites the entire sector.

The disk uses a *read/write head* to transfer data to and from the platter. The operation of moving the head from one track to another is called *seeking*, and the heads generally move together as a unit.

Because the disk heads move together as unit, it is useful to group the tracks at the same head position into logical units called *cylinders*. Sectors in the same cylinder can be read without an intervening seek.

Formatting

Before a disk can be used, the raw disk must be *formatted*. Formatting creates sectors of the size appropriate for the target operating system. Some systems allow disks to be formatted by user programs; usually, the manufacturer formats a disk before sending it to a customer.

One or more disk drives connects to a *disk controller*, which handles the details of moving the heads, etc. The controller communicates with the CPU through a *host interface*. Moreover, through *direct memory access* (DMA), the controller can access main memory directly, transferring entire sectors without interrupting the CPU.

Look at picture. Fig 5-3 from Tanenbaum.

Disk Latency

Three factors influence the delay, or *latency*, that occurs in transferring data to/from a disk:

1. *seek latency* — time needed to move the head to the desired track
2. *rotational delay* — time needed to wait for the sector of a track to move under the head
3. *transfer delay* — the amount of time required to read/write the sector

File Implementation

An important (experimental) observation is that:

1. the majority of files are small
2. a few files are large

We want to handle both file types efficiently.

The operating system may choose to use a larger *block size* than the sector size of the physical disk. Each *block* consists of consecutive sectors. Motivation:

- a larger block size increases the transfer efficiency
- might be more convenient to have block size match the machine's page size

Disk Block

The size of transfer convenient for operating system is a *disk block*. It may be the same size as a sector or larger. Generally moving to larger block size. NTFS uses 4K block size for disks larger than 2GB. FAT-32 uses 4K up to 8GB, 8K up to 16GB, 16K up to 32GB and 32K above 32GB.

Ideally want to locate blocks of a file in near proximity on disk to avoid excessive head movement. Can lead to fragmentation with too many small block runs because the size of a file may not be known a priori.

Will look at some specific policies later.

Free-Space Block Management

Approaches for keeping track of free blocks:

1. Bit map—devote a single bit to each block
2. Linked List (in groups)—each element of the list contains the number of a block. This block contains a group of free blocks. Thus for 16-bit block numbers (FAT-16) and 1K block, can fit 512 block numbers in the block. First 511 are really free and the last points to the next block of free blocks.

   Also can groups set of consecutive free blocks using an address/count approach.

Can look at space/time tradeoffs for each approach.

Caching

Caching is crucial to improve performance. Why?

- many file blocks will be accessed again soon
- consider the cycle of editing, compiling, and running a program
- consider frequently used commands such as *ls*, *vi*, etc.

Using memory-mapped files integrates the cache for files with the virtual memory system.

Otherwise, must maintain a separate file system cache in Operating System.

Unfortunately, caching may cause data in memory to become out of step with data on disk. This is known as the *cache coherency problem.*

Log-Structured File System

Traditional files systems maintain a number of data structures on disk (directory structures, free-block pointers, indoes ...). These structures may be cached in memory, but ...

A crash in the middle of changes to files can leave these structures in an inconsistent state.

One solution is to run a consistency check on system reboot to ensure that file system structures are in a consistent state—time consuming!

Becoming more common to maintain a log of file system metadata changes. Changes are *committed* as a *transaction* once written to the log. Completed transactions are removed from log. Incomplete transactions can be replayed on system reboot.

Also faster performance for file requests as only log file needs to be written before committing transaction versus waiting for all data structure updates to be completed in critical path of the request.

File System Reliability

File Recovery

To guard against complete file loss, most systems recommend that operators *dump* the file system to archival storage (floppy disks, Zip drives, tape, etc). Thus, some or all files could be restored after a disk failure.

Because of the expense of dumping the entire file system, most dumps are *incremental*, meaning that only files modified (or created) within the last N days are saved. Once a week (month) a complete dump of the file system is performed.

Operators cycle through a sequence of dump tapes, reusing a tape only after the files stored on the tape have been archived to a another tape. For instance, daily dump tapes might be recycled only after the weekend dump has saved all files modified since the last weekly dump.

Thus, it is not always possible to restore every file from a dump, but chances increase the longer the file has been in existence.

For additional safety, backups are often stored off-site. A back might store backups in a different region for increased reliability.

Redundant Array of Independent Disks (RAID)

Section 12.7 of SGG

Improve reliability by introducing file system *redundancy*.

There are several *levels* of RAID disks. Simplest is to use *mirroring* (RAID level 1) where each disk is shadowed by a copy. Requires double the disk space, but reads can be faster by reading from both disks.

Another level is *block interleaved parity* (RAID level 4). Fewer redundant disks. Assume nine disk blocks. Use one parity block for every eight data blocks. This is called *disk striping* or *interleaving*. If one of the disks fails then the data can be recovered using the remaining eight disks and the parity computation. Reads are quick because the data is spread amongst eight disks. Writes are expensive because parity must be recomputed.

Problems with RAID

RAID protects against physical media errors, but not necessarily software and hardware bugs.

Other file systems also introduce checksums for all blocks to verify the stored data is consistent with checksum. Used by Google File System for example.