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

Note:

- because some systems interrupt after each sector operation, "consecutive" sectors may mean "every other physical sector" to allow time for CPU to start the next transfer before the head moves over the desired sector
- some systems allow transferring of many sectors between interrupts

Disk Block

The size of transfer convenient for operating system is a *disk block*. It may be the same size of 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.

Can also look at allocating blocks in a contiguous manner, but may not know the total needed for a file at creation time. Also can lead to fragmentation with too many small block runs.

Details of policies for reducing latencies for retrieval of blocks from disk discussed in previous course.

Free-Space Block Management

Approaches for keeping track of free blocks:

1. Bit map—devote a single bit to each block
2. Linked List—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.

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

- database applications that must insure that the database is in a known state.

  One solution is for the operating system to provide a "flush" system call, that writes to disk the file blocks associated with the file descriptor. The system might also flush the cache when the file is closed.
- machine crashes, where the disk must be in a consistent state. For instance, it wouldn't be good to have a directory entry point to a file that hadn't been written to disk (or vice versa).

File System Reliability

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 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 14.5 of SGG

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.