6.826 Principles of Computer Systems **Fall Semester, 1993**

Lecture 3: September 20, 1993

Lecturer: Butler Lampson *Scribes: Anders Fornander and Cliff Young*[1]

1 File System

This lecture is the introduction to the specification and implementation topics of simple file systems (excluding concurrency issues). A file system maps names to storage, much like a memory. The names in a file system, however, are typically variable-length strings or path names and the storage is a variable-length sequence of bytes. In addition to the topic itself, Professor Lampson pointed out that the example of a specification and implementation of a simple file system (Handout 9) also enhances our understanding and use of the language Spec.

Real file systems usually deal with concurrency issues; most of these issues are deferred for later in the course. However, this lecture will handle a form of concurrency: behavior under failures. In the following discussions and references to examples, nontrivial atomicity brackets will enclose actions that must be atomic with respect to failures.

2 Disks

2.1 Physical Characteristics

Disks store data like RAM but have quite different characteristics. The main differences are that bytes on disk do not require power to persist and that, on a disk, sequential access is much faster than random access. Disk storage compared with RAM storage is much cheaper per byte.[2]

A magnetic disk stores data as magnetic domains in concentric rings called *tracks* on the surface of a spinning disk.[3] Data is read by examining the flux of a domain; it can be changed using an electromagnet. Tracks are further divided into *blocks* of 512 bytes.[4] To access data on the disk a read/write head is moved radially across the disk to the selected track (incurring *seek delay*), where it waits (incurring *rotational delay*) until the appropriate section of the track is rotated into position under the read/write head. Once positioned, a sequence of data bytes may be read from the disk quite rapidly.

The *access latency* (seek time plus rotational delay) for a random block of data is high. The average seek time is around 8 ms (worst case: 10 to 20 ms) and the average rotational delay about 5 to 10 ms (worst case: 25 to 40 ms). On the other hand, the bandwidth of sequential reads reaches 2 to 5 MB/s. Hence, once the block we want is under the read/write head, it can be transferred rapidly

[1] These notes are based on those written by Ken Mackenzie from Fall, 1992.

[2] \$1–2/MB for disks compared to \$20–30/MB for RAMs.

[3] In contrast, data on a CD-ROM is stored in a single spiral track that winds from the center to the outer edge of the disc (heritage from audio CD).

[4] The trend is toward larger block size.

(block size/bandwidth; here 0.5KB / 5000KB/sec = 100 ms). This disparity between sequential and random access of disk storage implies that the file system should attempt to access data sequentially whenever possible; *i.e.*, plan ahead and read a bunch of sequential blocks instead of addressing each block separately.

The mean time between failure (MTBF) of disks is very high. For modern disks it is about 300K hours. This does not mean that if you buy an individual disk, it will not fail for 300K hours. Rather, it means that in a large sample of disks, on the average one will fail every 300K disk-hours.

Disk storage is *stable*; data that has been written persists even without the application of power. Stable is the opposite of *volatile* storage, which can be erased during a crash or power failure.

2.2 Blocks and Crashes

Disk data is stored in blocks and disk drives only support data transfer in the size of blocks. Blocking is useful because it amortizes the high random access time across an entire block of data. On the other hand, blocking adds complexity to the file system implementation if it is expected to export an interface that allows the user to ignore block boundaries.

Page 4 of Handout 9 shows a disk specification that uses *extents*, which are (byte index, length) pairs. In the description of WriteBlocks, atomic brackets surround the disk update; this is difficult to do when failures are possible.

Disk block transfers are the unit of atomic operations supported by disk drives. Different disk drives provide different guarantees about the behavior of disk block writes at the time of a system crash. The most desirable behavior is that the write is atomic with respect to the failure: either the write occurs and the entire block is written properly, or the write does not occur and the block is unchanged. Such behavior requires buffering the block and sufficient power reserves to complete the block write in progress when the power fails. Less desirable but still usable is the behavior that in addition to those two cases, the block may also end up with detectable bad data, but that no other block will be affected. Other worse behaviors are possible.

Since data stored on a disk is persistent, it seems reasonable to expect that a file system can be built that will remain consistent across crashes.

2.3 Alternate Disk Technologies

Besides caching (treated in the next section), Professor Lampson described two other disk abstractions that are less close to the physical device. *Disk Mirroring* keeps two duplicate disks in order to increase reliability; the homework explores disk mirroring in more detail. A *RAID*, which stands for *Redundant Arrays of Inexpensive Disks*, attempts to increase both reliability and the data transfer rate using a collection of small inexpensive disks that function as one large disk. RAIDs allow the use of several separate I/O controllers operating in parallel. These parallel accesses can be coordinated to satisfy a single logical I/O request, or can service several independent I/O requests simultaneously. In addition, RAIDs can be made more reliable, by *striping* data bit-wise across disks, and including a parity disk which holds the exclusive-or of the other disks. If any single disk fails, its contents can be reconstructed using the remaining disks.

3 Caching

Caching reduces the average random access read latency to a storage device by storing copies of frequently used data in storage that may be accessed faster. In the cache of disk storage, the cache is kept in RAM. Generally disk data is cached at the granularity of a disk block. The average access time of a block in a cache implementation is: $T(\mathrm{hit}) + m * T(\mathrm{miss})$, where m is the miss rate of the cache. It is easy to improve over disk performance using a cache, but it is hard to get a performance close to that of RAM. For example, with $T(\mathrm{hit})$ about 10 us, $T(\mathrm{miss})$ about 10 ms and m equal 0.1 performance will be 10 times better than the raw disk. But to achieve performance 10 times worse than RAM we need m equal 0.01.

Caches do not help speed up writes; instead a technique called *write-behind* can be used. Write-behind buffers blocks in fast memory, then writes the blocks to the disk later. With this technique, from the point of view of the client, writes complete without waiting for the disk as long as the space for buffering writes is not full. Write-behind introduces an additional complexity in keeping the file system consistent through crashes because it adds volatile state. The user interface may require an added sync or flush operation, both to force remaining buffers to be written back, and to ensure the order of writes to the stable disk.

Similar kinds of consistency issues arise at the file system level, and are generated because of the friction between fast, volatile storage and slow, stable storage.

As in disks, alternate technologies can help ensure more consistent writes even with write-behind. Such technologies include stable RAM, UPS (for Uninterruptible Power Supply), battery-backed RAM, and Flash RAM. The notebook industry uses Flash RAM because it consumes less power than disk storage. However, these technologies introduce problems of their own. UPSs and batteries can fail. Without protection systems, a bad program might randomly write to stable memory, destroying the supposedly stable state.

4 Files

Files are specified on pp. 2–3 of Handout 9. A file is modeled by a sequence of bytes; a directory maps path names (which are just strings) to files.

The major operations on files are read and write. Reads and writes move variable amounts of data. Rather than reading and rewriting entire files, it makes more sense to take apart the component data types in describing how to read and write parts of files. While doing this, read requires range checking. Write is complicated by writes that extend beyond the end of the file. In such cases, the specification can do one of: wreak `HAVOC`, return an error, or perform some useful behavior. The specification does the third, extending the file to include the written area and filling any intervening area with 0's. Note that the specification is written inefficiently, using a VAR declaration and a large `ALL` statement. This is acceptable in a specification, because specifications are not meant to be run: this write construction is executed only in "the eye of God."

There are other operations to create, remove, and rename files; p. 3 also includes an alternative write specification.

The disk specification is designed to mimic the physical specifications of disks; the file system specification masks many of these properties.

5 File System Implementation

Professor Lampson briefly introduced file system implementations. In addition to supplying an abstraction of variable-sized files built on top of fixed size disk blocks, the file system must also supply a directory. In the specification, the directory is just a map from strings to files, but the directory implementation is more complex, since it must also be stable. See Handout 9 for a file implementation example.

Professor Lampson briefly summarized three fundamental problems to address in building a file system implementation:

- Indexing. Some sort of data structure must be able to map a (filename, index) pair into disk block location; this is complicated by the fact that files are of variable size, while disk blocks are fixed size.
- Atomicity. The index data structures need to be stable, so they can be used after a crash. The only atomic operation provided by most disks is writing a single disk block.
- Performance.