6.826 Principles of Computer Systems	Fall Semester, 1992

Lecture 3: September 21, 1992

Lecturers: Bill Weihl and Butler Lampson — *Scribe: Ken Mackenzie*

1 Handouts

This lecture covered two topics. First was a short overview of proving implementations correct. This section was accompanied by Handout 8: Abstraction Functions and Rep Invariants. Second was the first of two lectures on file systems accompanied by Handout 9: Simple File System Implementation.

2 Correctness

An approach to proving that an implementation of a specification is correct is to view both the implementation and the specifications as descriptions of state machines and then to prove that the state machines are equivalent under (1) an abstraction function mapping implementation states to specification states and (2) a representation invariant that characterizes the set of reachable states, i.e. a predicate that is true for all valid states of the implementation state machine.

The abstraction function allows one to prove inductively that the implementation matches the specification. The base case requires showing that all the initial states of the implementation map via the abstraction function to initial states of the specification. Then, inductively, show that transitions (invocation/response pairs) between the states of the implementation map to transitions between the corresponding states in the specification.

The representation invariant serves to limit the domain of the abstraction function to sensible states of the implementation. The example given in lecture was of an implementation of a memory using a hash table. The hash table lookup fails if a location is stored in the wrong hash bucket. An invariant function for a hash table implementation would be true only for states in which all addresses are in the correct bucket since it is known that the store operation for a hash table implementation will never store an address in the wrong bucket. The invariant simplifies the correctness proof by limiting the implementation states that must be considered.

Another advantage of using abstraction and representation invariant functions as part of a correctness proof is that it may be possible to make slight alterations to the implementation that require only slight alterations in the proof. If the abstraction function and the rep-invariant are preserved, the addition or modification of an operation requires proof only for the corresponding transitions. On the other hand, if (say) the rep-invariant is changed, the proof must be redone. The cost of the technique is that it requires applied creativity to invent the functions. Inventing the functions should be part of the design process.

3 File Systems

The second part of the lecture dealt with the specification and partial implementation of a simple file system, and the issues surrounding the modeling of such a system. The file system maps names to storage, much like a memory. The names in a file system, however, are typically variable-length strings or pathnames and the storage is a variable-length sequence of bytes. Several introductory topics were dealt with in this lecture and the bulk of implementation issues left to the next lecture. Topics include:

- **Physical device characteristics –** How the underlying technology required for a file system affects the approach to forming a file system abstraction. Performance and cost of a typical hard disk and cache were examined.
- **Blocks –** Disk data is stored in fixed-sized sequences of bytes for performance reasons. The file system interface allows the user to ignore this blocking.
- **Stability –** The notion that data blocks on the disk are expected to persist across system crashes and thus file systems can be built to remain consistent across crashes.
- **Caching –** Basic principles of cache use in a file system, and ways of specifying its operation.
- **File representation –** Choosing a representation that will balance good performance and the functionality desired by the client.

4 Disk Characteristics

Disks store data like RAM but have quite different characteristics. The main differences are that bytes on disk do not require power for persistence and that, on a disk, sequential access is much faster than random access. Disks are also about an order of magnitude cheaper per byte than current DRAMs.

A disk drive stores data as magnetic domains on the surface of spinning disks. The domains are read and written by a movable head. The head positions itself by (1) moving radially across the disk to the selected track and (2) waiting for the appropriate section of a track to rotate into position under the head. Once positioned, a sequence of data bytes may be read from the disk quite rapidly. Disk data is usually read and written in blocks because of these properties. A typical block size is 1K bytes.

The access latency for a random block of data is high. A typical average seek time might be 20mS, about 10mS to move the head on average and about 10mS in rotational latency. On the other hand, the bandwidth of data read sequentially reaches 2-3MB/sec, only about an order of magnitude slower the bandwidth of main memory RAMs. This disparity between sequential and random access of disk storages implies that the file system should attempt to access data sequentially whenever possible.

Disk storage costs about \$2/MB vs. DRAMs at about \$24-30/MB today.[1] RAM is thus about

[1]Judging by Alewife group purchases in the last few months

15 times more expensive now and the technology trend is that this factor will decrease in the future. This only moderate cost disparity suggests that caching disk data in RAM for performance improvement may not be very costly.

Finally, disks storage is stable, i.e. persistent without the application of power. The mean time between failures (MTBF) of disks is also very high. For modern disks it is as much as 250K hours. We are not considering at this point how to deal with disks that fail.

5 Blocks and Stability

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.

The example file system implementation in Handout #9 allows the user to ignore blocks and thus contains special cases for situations in which write operations overlap block boundaries. In these cases, the blocks to be partially written must first be read in their entirety, modified and finally written back.

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. Less desirable but still usable is the behavior that in addition to those two cases, the block may also end up with detectably 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, e.g. power failures. Exploiting the persistence of disk data to build a stable file system is not trivial, however. In order to be stable across crashes the file system must contrive to update files and their associated directory entries together atomically. Further, writing entire files atomically from the point of view of the client is impractical because the files may be very large, but at least the file system can assure that the blocks that comprise a file are written in order.

The implementation in Handout #9 gives an `UpdateFile` operation that is non-atomic. This `proc` only guarantees that the blocks of a file are written in order.

6 Caching

Caching attempts reduces the average random access latency to a storage device by storing copies of frequently used data in storage that may be accessed faster. In the case of disk storage, the cache is kept in RAM. Generally disk data is cached at the granularity of disk blocks. Caching improves the average performance of read operations. The latency to find the first byte of a block (i.e. not including transfer time) with caching becomes, on average,

$$T_{cache} + (1 - h) \cdot T_{seek}$$

where T_{cache} is the time to search for the block in the cache, T_{seek} is the time to position the head for a block on disk and h is the hit rate of the cache.

A similar strategy to speed up write operations is to buffer blocks in the cache or separately and then to write the blocks to the disk later and in order. This is called a *write-behind* strategy. 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.

Caching 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* operation to force remaining buffers to be written back. The general problem of maintaining a consistent file system that uses volatile state is solved with logging, to be described in the next lecture.

7 File Representation

The representation of files on the disk is a significant factor in implementing a file system, since performance of the file system, efficient use of disk space, and disk space allocation all depend on the representation used. What is required is a mapping of a file identifier (usually a character string) to disk storage units (disk blocks in this example).

The most straightforward representation to consider is the mapping of the filename to a series of contiguous disk blocks. While this obviously maximizes sequentiality it makes the allocation problem quite difficult. Essentially the file system would have to be completely reorganized after only a few modifications. Apparently this technique is nevertheless still used in some mainframe database applications where the file sizes change only rarely.

A general solution to the allocation problem is to partition the file into *extents*, i.e. sequences of disk blocks, each at different disk addresses. In a hierarchical setup, each extent contains either (1) blocks of data or (2) a sequence of disk addresses for other extents. This arrangement allows a file to be represented as a tree of extents where the leaf nodes contain the data.

Old UNIX file system implementations essentially fixed the size of an extent at a single data block. This approach is wasteful because it keeps a pointer for every data block even though we expect to place data blocks sequentially for performance reasons. A better scheme is described by the `Dsk` module in Handout #9. This version defines an extent as a pair with a disk address and block count. This scheme saves space whenever consecutive blocks are arranged sequentially on the disk.