6.826 Principles of Computer Systems **Fall Semester, 1992**

Lecture 4: September 23, 1992

Lecturer: Butler Lampson *Scribe: Anne Clark*

1 Overview

Today's lecture finished the discussion of simple file systems begun last time. Then we discussed encoding/decoding for stable storage, and broached the topic of crash recovery.

2 File System Representations

Assume that there is one directory mapping path names to file structures:

```
Dir = PN -> F
```

The most straightforward way to represent the file would be to store the data contiguously in a sequence of data blocks:

```
F = [start: DA, blocks: INT, bytes: INT]
```

where `start` is the disk block address of the first block, `blocks` is the number of blocks occupied by the file, and `bytes` is the exact size of the file in bytes. This scheme above provides sequentiality, which is good for performance but makes storage allocation impractical. Such a method involves constant reorganization as files grow and shrink and leads to extensive fragmentation.

Another solution uses a hierarchical approach to represent the file system. A tree structured index is built whose internal nodes are index blocks containing pointers to either data blocks or other index blocks.

```
F = INDEX
INDEX = UNION [block: DA, indexes: SEQ INDEX]
```

The problem with this method is that each data block has its own entry in the index, whether or not it is contiguous with other blocks. There is no obvious sequentiality. Even if the sequentiality is somehow extracted by noting that successive blocks are adjacent, this method wastes space by storing all block addresses.

A third method for representing a file system is actually a generalization of the simple sequential one outlined above. This solution partitions the file into a number of contiguous series of disk blocks. On page 7 of Handout 9, the concept of *extents* is explained. Extents are "references to contiguous sequences of disk blocks." F is a sequence of these extents that would be concatenated together to get the data.

```
F = [data: SEQ EXTENT, bytes: INT]
EXTENT = [start: DA, blocks : INT]
```

Problems with this implementation occur when F is big so that the sequence of extents does not fit into a block. This is usually solved by building a hierarchical structure like the one described earlier, except that the leaf nodes are variable sized extents of contiguous data blocks rather than individual blocks. The indexing nodes of the tree might themselves be variable sized sequence of blocks, but for the sake of simplicity, this is not employed in most file systems. Most implementations use B-trees to prevent the structure getting unbalanced and keep the index nodes fixed-size while the data is in variable-sized extents.

```
F = EXTENT
EXTENT = UNION [[start: DA, blocks : INT], extents: SEQ EXTENT]
```

3 Allocation

Allocation involves the assigning of free space to extend a current file or form a new one. The free space available for allocation comprises all blocks that are not reachable from the directory, and is constantly changing as data is added and deleted on the disk. A file system needs some method of keeping track of what space is free to prevent valid data from being overwritten. The most basic way of allocating space would be to define reachable data (disk addresses already in the file structure) as

```
Reachable (da) = {pn | da IN blocks (dir (pn))}.
```

where `blocks` returns a set of the disk addresses of all blocks in the file.

When a free disk block is needed, adding da to a file involves searching for an unreachable block:

```
VAR da | ~ Reachable (da)
```

This scheme is very inefficient, because it explores the entire file system each time a new disk address is needed. Its form is useful though, because when a more optimized solution is found, we can ask if it is a restriction of above. If it is, the new method need not be proved correct separately. A better solution to the free space problem would be to define a variable `free` which tracks whether a block is free:

```
VAR free: DA -> Bool
```

The invariant to be maintained for correctness is:

```
free (da) ==>  ~ Reachable(da)
```

The standard way of representing free is through a bit table with a bit per disk block. (If the table gets too big to fit in a block, the extent trick can be used.) Each update involves two writes: to the file itself and to the free table. The problem is that a crash could occur between these two atomic operations. If the system writes to a file and then crashes before it resets the free-bit in the table, the disk address could be allocated to another file on recovery, breaking our invariant.

On the other hand, the invariant permits updating the table before the file. If a crash occurs between the writes, the block is lost, but there is no loss of data. This can cause problems after a time if too many blocks are lost, so most systems now include a garbage collection system. Many systems also amortize the updates to bit table by taking a large batch of free block addresses in advance so that the system is not slowed down by having to do two writes at the time of each file update.

4 Encoding and Decoding

Programs are written using abstract types to define variables and relationships. This information must be preserved when the data is saved in a file. Encoding involves representing these types as sequences of bytes. Any system that is used to do this must (1) be total, that is, work for any type T, and (2) decoding the encoded form of a data structure must yield the same data structure:

```
(ALL t |  T.Decode (T.Encode (t)) = t)
```

The standard encoding/decoding technique is called *TLV*, for *Type-Length-Value*. This method encodes a data structure into the type (say, as the type identifier), the size of the encoded value, and the encoded value itself. This technique is completely self-describing, but often carries extra baggage and needs a naming scheme for the types.

5 Crash Recovery

A generic way is needed to deal with stable and volatile variables after a system has crashed. The stable variables are not affected by a crash, while the volatile ones are reset. We assume that writes to disk blocks are atomic and will either be completed or not done at all. Any disk write involving a number of blocks of data can be interrupted by a crash. This is not a bad assumption considering that a disk block write takes less than 1ms, which would not be affected by the system losing power, the most common form of crash. The basic idea is to build a system with both stable and volatile states:

```
S : state
ss : S := S$s0()                % initialization of the stable state
vs : S := S$s0()                % initialization of the volatile state
```

The system would then need to provide means of doing actions which would affect the volatile state, a procedure to commit the volatile state to the stable state, and a crash recovery procedure that copies the stable state into the volatile one.

```
A : Action = S -> (V,S)

Do (a : A) -> V = VAR v | (vs,v) := a (vs)
        RET v

Commit ( ) = ss := vs

Crash ( ) = vs := ss
```