Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.033 Lab, Spring 1998 Handout 7 — Simple File System (SFS) April 3

Due date of preliminary design: April 10
Due date of Stage 1 (working system): April 24
Due date of Stage 2 (tuned and robust system): May 6
Due date of design paper: May 8 (not April 30)

1 Introduction

The goal of this project is to build a file system. A file system, by its nature, incorporates many of the issues we discuss in 6.033, including naming, fault-tolerance, concurrency control, and storage management. In addition, issues in security and networking appear in some designs. The file system you will be building for this project is a real file system - you will be able to use it from any program just like the system's built-in file system.

Unlike the previous projects, you are strongly encouraged to execute this project in teams of three. Find someone else to work with and send mail to the TAs (6.033-lab-tas@mit.edu), telling them who your teammates are. Make sure to do this right away. The first part of the project is due very quickly (April 10).

2 Preliminary Design Paper

The first stage of the file system project is a preliminary design paper describing your group's file system. In the paper, you should discuss the on-disk layout the file system as well as the in-memory data structures. You may also discuss the algorithms you use for updating the file system. You may also wish to treat issues of robustness and performance in your preliminary design document as you will have to deal with them in your final design paper. As always, you should include rationale for all your design decisions.

3 Stage 1: File system

In the first stage, you will implement a working file system, though it need not be robust in the face of sudden shutdowns.

There are several challenges in implementing a file system; three of the main ones are: (1) the main-memory and on-disk data structures to map files names onto disk blocks and to keep track of free blocks; (2) the disk layout; and (3) the recovery strategy. We discuss each challenge in turn.

This assignment revolves around three of the functions provided by a file system:

1. Naming. Human-readable file names must be mapped to the disk blocks that they are associated with.

2. Organization. File systems (typically) have mechanisms humans can use to organize information. Computer scientists like hierarchies, as a result, the tree directory structure has become the dominant mechanism for organization.

3. Persistence. Data is entrusted to the file system and is expected to remain there (uncorrupted and undeleted) until an explicit request is made to delete it. This aspect includes support for allocation and deletion of files. File systems also provide mechanisms to modify existing files.

Important issues that we will treat lightly include protection (ensuring only authorized programs can read/write/delete/create files) and performance (disks are slow: most research on file systems concentrates solely on performance).

To make the assignment concrete, we discuss a common file system organization below: you are encouraged to construct your own ideas, and to simply use this discussion to understand the issues involved.

There are three main data structures file systems manage:

1. A disk block free list. This is a simple data structure (for instance, a bitmap) that tracks what blocks on the disk have been allocated. This data structure is, for obvious reasons, persistent (i.e., on disk). You will have to establish some convention for locating it across reboots.

2. Inodes. An inode is persistent memory that contains pointers to disk blocks or to "indirect blocks" (you can think of these as other inodes). Inodes are used to map a file to the disk representation of a file: each file has its own inode, the inode has pointers to all disk blocks that make up that file.

3. Directories. Directories are persistent mappings of names to inode numbers for all the files and directories that the directory is responsible for.

Given this class of objects, the following algorithms are used to create, grow, and delete a file.

The actions to create file:

1. Allocate and initialize an inode.

2. Write the mapping between the inode and file name in the file's directory (usually the current working directory).

3. Write this information to disk.

To grow a file:

1. Load the file's associated inode into memory.

2. Allocate disk blocks (mark them as allocated in your free list).

3. Modify the file's inode to point to these blocks.

4. Write the data the user gives to these blocks.
5. Flush all modifications to disk.

To shrink a file:

1. Load the file's associated inode into memory.
2. Remove pointers to the disk blocks from the inode.
3. Mark the disk blocks as free.
4. Flush all modifications to disk.

Modifying directories is similar.

Note that we are careful about when modifications to file system state are made persistent. For instance, in the case of creating a file, if we flush modifications to disk after stage (1), a crash that happened before (2) would cause us to have a disk block allocated that was not used. If we did not flush modifications to disk at all, then a crash would cause the file to be lost. You will have to pay particular attention to this and similar problems in the third part of the assignment.

While your file system does not have to survive random crashes at this stage, it does have to maintain state persistently. In particular, it should be able to run, be powered down, and then be "rebooted" in the state that it was in when it was shutdown (i.e., all files that were written at shutdown time should still be there, and no garbage files should suddenly appear).

In addition to correctness, you should consider space efficiency in this part of the assignment: your file system should have a low overhead per byte of file data.

3.1 Interface and Implementation

How will you implement this file system? In many modern operating systems, the file system is implemented in the kernel which is difficult to modify and debug (especially when you don't have the source code!). However, it turns out that many modern operating system have Network File System (NFS) client support in the kernel. The NFS client talks using RPC to an NFS server, which actually implements the file system. We will take advantage of this architecture by implementing our file systems as NFS servers.

The source code for the NFS server skeleton (without a complete file system) will be posted in /mit/6.033/lab/lab3/src. The skeleton will support NetBSD, OpenBSD, Linux, and Solaris.

The NFS server skeleton is multi-threaded. Each NFS request will be launched in a separate thread. The threading is non-preemptive, so there will be only a couple points where the code yields to another thread. This non-preemptive multi-threaded environment is very similar to that of the original UNIX and BSD UNIX kernels.

However, even non-preemptive threads can be difficult to deal with, so to simplify your initial implementation, we recommend protecting the file system with a mutex so that only one thread proceeds into the file system code at a time. Later, when your file system is stable, you can allow multiple concurrent operations in your file system for improved performance.

A later handout will describe the thread and NFS server package in more detail.

4 Stage 2: Robustness

File systems are different from most other parts of the computer system in that they are *persistent*: modifications to disk survive power cycles. In contrast, most other state (e.g., main memory, registers, TLBs, and, as a result, data structures) is not persistent. Since the file system is main repository of persistent system state, we'd like it to be at least recoverable to a consistent state after an abrupt, unclean shutdown. For example, the Berkeley Fast File System guarantees that the file system is at all times recoverable to a consistent state. In addition, it guarantees that files not touched by operations in the thirty seconds preceding the crash will survive a crash. Other file systems (like NFS version 2) provide stronger semantics: that once a file operation returns, its effects will survive a sudden power-down.

Much reasoning about robustness in a file system stems from the problem of moving state from the non-persistent realm to the persistent realm in a series of discrete steps in such a way that a failure can occur at any point in time and still leave the system in a usable state. For instance, SFS will commonly write data to a file. This will involve a number of steps: allocating space for the file on disk, modifying the file's inode, modifying the disk block free list, and finally, writing the data to memory. SFS must guarantee that a failure after any of these will not corrupt the system (e.g., will not lose disk blocks, delete existing disk blocks, lose inodes, etc.).

Operationally, this part of the assignment will involve working through your code so that a failure at any point in time will not corrupt the system. You may require that a program be run at boot time to reconstruct parts of SFS, but it should always be able to be recovered to a usable state.

Failures to guard against include:

- Use of allocated disk blocks. Blocks may have been allocated to a particular file but the free list was only modified in memory, not on disk.
- Loss of disk blocks. Blocks may have been removed from the free list but not yet had an inode persistently modified to point to them.
- Loss of data. Data that has been written to disk should stay there.
- Mixing data from different files

There are, of course, many other failures possible.

The due date of this part of the project is May 6.

5 Final design paper

The final result of your project is a complete document detailing the design and implementation of your simple file system (SFS), the code that implements the design, and a short evaluation of your file system in terms of performance and functionality. This document can be used as the paper for the second design project in 6.033. The paper is due on May 8.