Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.033 Lab, Spring 1999 Handout 7 — Simple File System (SFS) March 31

Due date of Stage 1 (preliminary design): Friday, April 9
Due date of Stage 2 (working system): Friday, April 23
Due date of Stage 3 (tuned and robust): Wednesday, May 5
Due date of Stage 4 (design paper): Friday, May 7 (not April 29)

1 Introduction

The file system lab will provide you experience in designing complex systems, one of the major topics in 6.033. File systems, by their nature, incorporate aspects of naming, fault-tolerance, concurrency control, and storage management. In addition, issues of security and networking appear in more ambitious designs. This is a real-life file system. You will be able to use it from any program just like the system's built-in file system.

To help you design your file system, this project is organized into several stages. We provide a block I/O interface and implementation that will simulate a disk on top of a single file. Its interface includes functions to read and write disk blocks. You will build a simple file system on top of this interface. This file system should provide efficient storage. In particular, access to file blocks should be efficient and the amount of metadata needed to implement the file system should be small. In the third stage, you will make your file system robust in the face of failure (i.e., it should not corrupt data if it crashes). To test robustness, we will crash your file system at arbitrary points in its code: your file system should be able to recover to an internally consistent state irrespective of where these crashes occur.

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 7, not April 29 as in 6.033.

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 6.033-lab-tas. Tell us the names of your teammates. Make sure to do this right away. The first part of the project is due very soon (April 9).

The rest of this handout touches on some of the design and implementation issues, and provides some suggestions on a possible approach. We suggest you read the whole document before starting to work on Stage 1 so that you have a good idea how the whole project fits together and can plan your time carefully.

2 Stage 1: Preliminary Design Paper

The preliminary design paper describes your group's file system. You can learn more about the issues of file systems by reading the packet we handed out. There are three main challenges to consider: (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. Create diagrams to elucidate. You may also discuss the algorithms for allocating resources in the file system. You may want to discuss issues of robustness and performance in your preliminary design since you will have to deal with this anyway in the final paper. As always, you should include a rationale for your all design decisions. Keep track of your design alternatives too. While you do not need to describe each SFS function, you should keep in mind the interfaces given in subsection 3.1.

Turn in this paper during recitation April 9. This is a very important recitation because (1) you need to turn it in, (2) this is our last recitation, and (3) HKN will handout course evaluations.

3 Stage 2: File system

The next assignment is to implement your file system using the block I/O interface defined in subsection 3.1. The file system should be fully functional, but need not survive random crashes of the file system. This assignment revolves around three of the features provided by a file system:

1. Naming. Human-readable file names must map 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 explicitly deleted. 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 freelist. 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 persistently map human-readable names to inode numbers. Since directories are hierarchical, a single directory need only contain mappings that are in their context.

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

The actions to create a 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 freelist).
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.

We evaluate your implementation on how efficient it is, the storage overhead on disk, and how well it recovers from failures. You do not have to implement block caching. If you implement caching correctly, however, we will give you bonus points.

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!). Another option would be to have you implement a library of file system functions and then implement some test applications against that library. The advantage of this approach is that is significantly easier to debug. The disadvantage is that you cannot run unmodified applications against it. A third option, the one we chose, is to have you implement your file system as an NFS server on your local machine. You then mount that NFS server and the file system becomes part of your file system name space.

The source code for the NFS server skeleton (without a complete file system) will be posted in /mit/6.033/lab/src/three/. It should run under Linux and Solaris. We will post this code after you turn in the preliminary design paper. A further handout will talk about the NFS server issues in more detail. For the most part, you will not need to understand the RPC code used in NFS. We've abstracted out much of these issues in the following interfaces.

Block I/O Interface

A file system must be able to store its data in a persistent manner. The persistent medium in your assignment will be a virtual disk. Your file system will be built on top of this interface.

Below we suggest a block I/O interface (and we later provide the implementation). This interface will handle reading/writing blocks. You are free to create your own interface or implementation.

The interface we provide is similar to what the raw hardware provides. As a result we should be able to incorporate your file system code into a real OS with minimal fuss. OS software is frequently developed at user-level using emulation. It allows ease of debugging and use. It also removes the need for us to buy each student a disk.

We will provide an implementation for the following disk interface:

- `bread(int secno, int size)`: Return buffers which correspond to data starting at offset `secno` and extending `size` bytes.
- `bwrite(struct buf *bp)`: Given a pointer to a buffer, write that buffer to disk. The buffer contains information to describe the proper offset and size on the disk.
- `brelse(struct buf *bp)`: Free a buffer from memory without writing to disk. You would never call this function on a buffer waiting to be written to disk.

If you implement a disk cache, you should do so in this interface. Add a `bflush` operation to flush the dirty blocks to disk.

File System Interface

The file system interface we provide is as follows:

- `sfs_getattr`: get file attributes

- `sfs_setattr`: set files attributes
- `sfs_lookup`: look up file name
- `sfs_readlink`: read from symbolic link
- `sfs_read`: read from file
- `sfs_write`: write to file
- `sfs_create`: create file
- `sfs_remove`: remove file
- `sfs_rename`: rename file
- `sfs_link`: create link to file
- `sfs_symlink`: create symbolic link
- `sfs_mkdir`: create directory
- `sfs_rmdir`: remove directory
- `sfs_readdir`: read from directory
- `sfs_statfs`: get file system attributes

Each function takes a file handle as one of the arguments. A file handle is a name that uniquely identifies a version of a file. A file handle also acts as a context. For instance, a file name makes sense only in the context of a directory's file handle. A simple file handle might be:

```
struct fh {
  int padding[6];
  int inumber;          /* inode number */
  int gen;              /* generation number */
};
```

There is also a `sfs_mount` function which takes the name of your virtual disk (another filename). Your `sfs` server will automatically mount the disk, then wait for the kernel to request the following functions which you must implement:

Notice that the NFS loopback server does not have the same semantics of an in-kernel file system. For instance, NFS version 2 has no way to detect a file close operation. Keep this in mind for your implementation.

Along with these functions, you will have to implement a collection of functions to manipulate disk and memory structures (e.g., inodes, freemap, superblock, etc.). To keep the task manageable, you may assume that there is a single client using the file system; the single client may have multiple files open, though.

3.2 What to hand in?

The due date of this part of the project is April 23. Send us a pathname to a directory in your Athena locker that contains the source code (including Makefile) and documentation. Also, make sure you have tested your file system extensively. What is the ultimate test? Can you copy your code to a partition mounted in your file system, then compile your file system again?

4 Stage 3: 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, TLB's, and, as a result, data structures) is not persistent. Since the file system is the main repository of persistent 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 such as NFS version 2 provide stronger semantics: that once a file operation returns, its effects will survive a sudden power-down.

In reasoning about robustness in your file system, you should think about the order in which your operations occur. 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 freelist, 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.). This struggle involves mechanical operations (such as writing transaction software by hand) and semantic struggles (such as defining exactly what "usable" means). In talking about failure, more so than other domains, precise definitions are not semantic quibbles.

The first part of this assignment will involve defining the semantics of failure: at what point is the modified state of a file made persistent? For instance, if I write a single byte to disk, am I guaranteed that when the `bwrite()` call returns the byte has been written? Or do I have to perform a sync by hand? If I delete a file and the system crashes, will the file remain deleted? You should define a non-trivial (i.e., useful) recovery model that balances the utility of consistent state on disk with the expense of synchronous disk updates.

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:

- Use of allocated disk blocks. Blocks may have been allocated to a particular file but the freelist was only modified in memory, not on disk.
- Loss of disk blocks. Blocks may have been removed from the freelist 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. This list should give their flavor.

We will test this part of the assignment by crashing your program at random points in time and then rerunning it (repeatedly).

4.1 What to hand in?

The due date of this part of the project is May 5. Send us a pathname to a directory in your Athena locker that contains the source code (including Makefile) and documentation. In this part, we crash your file system and check whether it recovers correctly. The file system is a 60 point project which is half of your 6.033 lab grade. For each day late, you lose 2 points.

5 Final paper

The due date for the final design paper is May 7 so that your design paper and your software are not due on the same date. Your paper should follow the generic design project II criteria. The paper grade will reflect your merits in creativity, critical thinking, and technical writing. The same penalty from 6.033 for lateness applies. For each missed 6.033 recitation, your grade goes down one letter grade.