Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.033 Lab, Spring 1995–6 Handout 4—Simple File System (SFS) April 4

1 Due dates

Stage 1 .... April 12
Lab recitation section ..... April 12
Stage 2 .... April 29
Design paper .... May 2
Stage 3 .... May 10

2 Introduction

The goal of this project is to obtain experience with file systems. To this end, you are asked to write both a simple file system (SFS), a core component of any computer system, and a disk emulation library that it uses to run on top of UNIX. Implementation of this system will provide you with experience designing complex systems (one of the major topics in 6.033). While we give you a specification for the external interfaces for SFS, you are free to design the internals as you wish.

To help you in designing your file system, this project is organized into three stages. In Stage 1, you are asked to build a disk emulation library that will emulate a disk on top of a single file. Its interface includes functions to read, write, and flush "disk blocks". In Stage 2, you will build a simple file system on top of the disk interface you built in Stage 1. This file system should provide efficient storage. In particular, access to all the disk blocks which are part of a single file should be efficient, and the amount of metadata (i.e., data on the disk which is not part of any file, but is used by the file system itself) should be small. In Stage 3, you are asked to make your file system robust in the face of failure (i.e., it should not corrupt data if it crashes). To test its 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 regardless of where these crashes occur.

The final result of your project is a complete document detailing the design and implementation of your 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 2.

Unlike the STCP project, you are strongly encouraged to execute this project in teams of two. Find someone else to work with and send mail to Dawson Engler (engler@lcs.mit.edu), telling him who your teammate is. Make sure to do this right away. The first part of the project is due very soon (April 12).

The rest of this handout touches on some of the design and implementation issues, and suggests 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.

3 Stage 1: Disk library

A file system must be able to store its data in a persistent manner. The persistent medium in your assignment will be a disk. Your task for this stage is to design a simple disk library that reads and writes 4-kilobyte disk blocks from this disk. Your file system will be built on top of this interface.

Since we do not have a raw disk for each of you, your disk library will use a single (large) UNIX file to emulate a disk. This technique is similar in spirit to the network emulation layer you built for STCP. Your disk controller will divide the file into 4-kilobyte blocks. Blocks will be written using a "block number." You will use this number to compute the offset into the UNIX file. For instance, disk block 10 will be at byte offset $10 \times 4096 = 40960$.

The disk library's interface (which we define) is similar to that of raw hardware provides: as a result, just as in STCP, we should be able to incorporate your file system code into a real OS with minimum fuss. OS software is frequently developed at user-level using emulation. This allows ease of use and debugging. Importantly, it also means we don't have to buy each student a disk.

The disk interface is as follows:

```
int openDisk(char *filename, int nbytes)
int readBlock(int disk, int block_num, char *block)
int writeBlock(int disk, int block_num, char *block)
void syncDisk()
```

The calls return an error if the underlying UNIX system calls fail.

`openDisk` opens a virtual disk file named `filename` for reading and writing, and returns a descriptor used to access the virtual disk. The size of the "disk" is fixed at `nbytes`. If `filename` does not already exist, `openDisk` creates it.

`readBlock` reads disk block number `block_num` from the virtual disk `disk` into a buffer pointed to by `block`.

`writeBlock` writes the data in `block` to disk block `block_num` in `disk`.

`syncDisk` forces all outstanding writes to disk.

Your task is to implement this interface. While this task should be simple, you have only a single week to do it. The implementation will require that you understand how to open, read, write, and seek files under UNIX. Section 7.4 in Tanenbaum contains a short description of the relevant UNIX system calls. The ones on Athena are mostly identical, but verify their semantics using the man pages (for instance, try "`man 2 open`").

3.1 What to hand in?

The due date of this part of the project is April 12. Since the rest of the project builds on the disk library, make sure you make the deadline! 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 disk library.

4 Stage 2: File system

The next assignment is to implement a file system using your disk library. The file system must support the core UNIX file system calls. Your main task is to design and implement a disk layout, the main-memory and disk data structures that map filenames to disk blocks, and a recovery strategy to boot the file system using on-disk data structures. To get some idea what the issues are, read Tanenbaum's Chapter 7. This part of the project is the hard part, so do not wait until one night before the deadline!

The following interface should be supported by your file system:

```
int sf_create(char *name)
int sf_open(char *name)
int sf_close(int fd)
int sf_read(int fd, char *buffer, nbytes)
int sf_write(int fd, char *buffer, int nbytes)
int sf_lseek(int fd, int offset, int whence)
int sf_mkdir(char *name)
int sf_rmdir(char *name)
int sf_chdir(char *name)
int sf_shutdown(void)
int sf_sync(void)
```

These functions map closely to their UNIX namesakes. All functions should return -1 on failure; most also return 0 on success. All file and directory names can be either *relative* or *absolute*. A relative pathname is given with respect to the current working directory (a variable you must maintain). An absolute pathname is given with respect to the root of the file system (by convention, the root is "`/`"). For instance, "`/tmp/foo.c`" is an absolute pathname. As with Unix, the special directory name "." names the current working directory, ".." names the one above.

`sf_create` creates a file named `name`. The operation fails if the disk is full, if the file already exists, or if the given path does not exist.

`sf_open` opens the file named `name`. `sf_open` fails, returning -1, if the file does not exist or the user already has too many files open. On success it returns an integer that represents a file descriptor.

`sf_close` closes the file associated with the given file descriptor. `sf_close` fails if the file descriptor is invalid.

`sf_read` reads `nbytes` of data into `buffer` using file descriptor `fd`. `sf_read` fails if the file descriptor does not exist, or if `nbytes` is negative. It returns the actual number of bytes read on success.

`sf_write` reads `nbytes` of data from `buffer` into the file pointed to by `fd`. `sf_write` fails if the file descriptor does not exist, if `nbytes` is negative, or if the file system is full.

`sf_lseek` moves the current file pointerfor file `fd` by `offset` bytes. `whence` indicates whether this movement is in relation to the begining of the file (`SF_SEEK_START`), the end of the file (`SF_SEEK_END`), or to the current file pointer (`SF_SEEK_CUR`). You will define these constants in the interface header file you create for SFS. `sf_lseek` fails if `fd` is not a legal file descriptor, or `whence` is invalid (i.e., if it would move the file pointer to a negative offset). Seeking past the end of the file will extend the file; the newly allocated space will be initialized to zeros.

`sf_mkdir` creates a directory named `name`. `sf_mkdir` fails if the path does not exist, a file or directory named `name` already exists, or the file system is full.

`sf_rmdir` deletes the directory named `name`. `sf_rmdir` fails if `name` does not exist or is a file instead of a directory.

`sf_chdir` change the current working directory to `name`. `sf_chdir` fails if `name` does not exist.

`sf_shutdown` gracefully shuts down the file system. `sf_shutdown` should ensure that SFS has written all persistent state to disk; specifically, when the file system is powered up again, the set of files (and the file contents) should be identical to the logical set of files on disk at shutdown time.

`sf_sync` is used by applications to ensure that all data associated with their files has been written to disk. It always succeeds. Note that, for simple implementations of the Simple File System, `sf_sync` may not do anything at all.

These functions will be provided in a library that normal applications can link in. You should test your file system by running as many real applications as possible on it. To make this process feasible, you should write a program, `import`, that will copy a file from a UNIX file system onto your file system (the Unix file name is `import`'s first argument, the destination file name is `import`'s second argument).

There are three main challenges in implementing this interface: (1) the main-memory and on-disk data structures to map filenames onto disk blocks, and to keep track of free blocks; (2) the disk layout itself; and (3) the recovery strategy. To keep the task manageable, you may assume that there is at most a single client using the file system at a time; that client may have multiple files open, though. 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, and 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; most 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 current 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 a persistent structure that contains a list of 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 structures which map filenames to inodes, for all the files and directories the directory is responsible for.

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

To create a file, we:

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 and 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 you into 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 with the `sf_shutdown` call, 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. Specifically, your file system should have a low overhead per byte of file data.

When designing your system, you should think about the characteristics of the workloads it will run. A good workload to use as a model would be an Athena cluster. A useful measurement to take is the distribution of file sizes on the system. Use this measurement to derive the median file size, common outlying points, etc. Compute the overhead your file system would have under this style of usage.

We will evaluate your implementation mainly on correctness. Bonus points will be awarded for particularly space- and time-efficient implementations.

4.1 Extensions

This subsection lists a number of possible ways you can extend your system. *These extensions are not mandatory.*

The distribution of file sizes on your system will be non-uniform. For bonus points you should exploit this non-uniformity for improved performance (a favorite trick is treating both large files and small files specially). In characterizing your performance you should consider the following:

1. What percentage of time do you spend waiting for disk? This measurement can be approximated by running your file system without I/O (keeping all data in main memory) and comparing its performance with the overhead of performing explicit I/O and explicit syncs.

2. If disk access time is relatively constant, how will your file system scale with faster computers? For instance, if the CPU was ten times faster what speedup would applications actually get using your system?

3. What is your performance on very large files and very small files?

4. Develop a set of synthetic and real benchmarks (file system papers are a good starting point to find out about these). As you make each improvement to your system record its impact on performance. Also record the optimizations you did but abandoned because they were ineffectual.

5. Writing to disk is expensive. A common trick when going from one memory hierarchy to another is to use a buffer to hide their speed differential (this buffer is usually called a cache). Develop a file block caching strategy that hides disk latency. Note, however, that the larger your cache, the more memory you are taking from other uses of it (e.g., network buffers, program memory, etc.). How do you reconcile the need to have a large cache with the needs of other system services?

A particularly ambitious extension would be to handle concurrently open disk in order to implement RAID techniques to ensure reliability (i.e., if one disk crashs the file system should have enough redundant state to efficiently recover).

4.2 What to hand in?

The due date of this part of the project is April 29. 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.

5 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. Many problems with making a file system robust stem from the issue of when state is moved from the non-persistent realm to the persistent realm. The state is moved in a series of discrete steps, and a failure can occur after any step; you must ensure that any possible failure leaves the system in a usable state.

For instance, your SFS will commonly write data to a file. This will probably involve a number of steps; for instance: 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 very important.

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 library call returns the byte has been written? Or do I have to perform `sf_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 changing 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.

Some failures to guard against:

- Use of allocated disk blocks: This can happen if blocks are allocated to a particular file and the changes to the relevant inode were stored on disk, but the change to the freelist (marking the block as allocated) didn't make it to disk.
- Loss of disk blocks: This is the flip side of the above problem. Here, a block was taken off the freelist, and the modification to the freelist made it to disk, but the inode that was to refer to the block wasn't saved. Here, the freelist thinks that a free block is actually allocated.
- Loss of data: Data that has been written to disk should stay there.

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).

5.1 What to hand in?

The due date of this part of the project is May 10. Send us a pathname to a directory in your Athena locker that contains the source code (including Makefile) and documentation.

The due date is set on May 10—note that your design paper and your software are not due on the same date. However, plan your time carefully in this period because in this week you will also have to take the third 6.033 quiz. The Chair of the MIT faculty has assured us that this due date is in accordance with the end-of-term rules.