6.033 Spring 2000

Lab 3: A Simple File System

Due date of Stage 1 (preliminary design): Friday, April 7
Due date of Stage 2 (implementation): Friday, April 28
Due date of Stage 3 (design paper): Thursday, May 4

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 4, not April 27 as in 6.033. Please submit the paper to your recitation instructor or your lab TA by 5pm. **There will be NO extensions** since it is extended a week already.

Unlike the previous projects, you are to execute this project in teams of three. **Find someone else to work with and send mail to your lab TA. 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 7).

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 7. This is a very important recitation because (1) you need to turn it in, (2) this is our last recitation. After this recitation, you can still contact the TAs by making appointments.

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/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.

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).

5 What to hand in?

The due date of this part of the project is April 28. Place a tar ball that contains the source code (including Makefile) and documentation in the turnin directory of a member of your group, then send an email to your lab TA. 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? We may also crash your file system and check whether it recovers correctly. The file system is a 60 point project and is half of your 6.033 lab grade. For each day late, you lose 10 percent.

6 Final paper

The due date for the final design paper is May 4 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.

7 Recommended Readings

What do real file systems look like? Below are several real-world examples of file systems from research and industry. The UNIX file system design is the oldest and simplest. It dates back to the 1970s in origin. The Berkeley Fast File System is a derivative of the original UNIX file system with incremental improvements for performance reasons. The Network File System (NFS) provides transparent, remote access to files.

In addition to describing interesting file systems, the papers are examples of well-written papers for the field. For example, most have clear motivation sections explaining what issues influenced the on-disk layout. You should emulate their style for your final write-up. We attached a fundamental paper to this handout. The other references are completely optional, but may be useful to you depending on what kind of file system you design.

- The UNIX Time-Sharing System (reading #9)

  Dennis M. Ritchie and Ken L. Thompson. Bell System Technical Journal 57, 6, part 2 (1978) pages 1905-1930.

- Berkeley Fast File System (attached)

  McKusick et al. "A Fast File System for UNIX," ACM Transactions on Computer Systems, Vol. 2, No. 3, August 1984, pp. 181-197.

- Log Structured File System (reading #26)

  Rosenblum and Ousterhout. "The Design and Implementation of a Log-Structured File System," Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles, Pacific Grove, California (October 13-16, 1991) pages 1–15. Published as Operating Systems Review 25, 5 (1991). Also appeared in ACM Transactions on Computer Systems 10, 1 (February, 1992) pages 26–52.

- UNIX File System Case Study

  Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, 1992, pp. 307–310.

- DOS (FAT) File System Case Study

  Tanenbaum, pp. 340–342.

- Silicon Graphics XFS

  Sweeny et al. "Scalability in the XFS File System," Proceedings of the USENIX 1996 Annual Technical Conference.

- Co-Locating Fast File System

  Ganger and Kaashoek. "Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files," Proceedings of the USENIX 1997 Annual Technical Conference, January 6–10, 1997, Anaheim, California, pages 1–17
  (from http://www.pdos.lcs.mit.edu/pubs.html#Storage+Management).

8 Introduction to the loopback server

The goal of the third lab is to build a real file system. Unfortunately, real file systems often live in the kernel, which is hard to modify and debug. However, for many OSes, there are hacks that allow a file system to be implemented as an application outside the kernel. The NFS loopback server is an especially portable example of such a hack.

8.1 How it works

The idea behind an NFS loopback server is simple. In the usual NFS scenario, a machine `A` will mount a network volume exported by an NFS server running on machine `B`. In the loopback scenario, the machine `A` mounts a volume from an NFS server running on machine `A` (i.e. itself). Your job is to implement file system operations in the NFS server.

9 Using and understanding the NFS loopback server

9.1 How to compile the NFS loopback server

Look in `/mit/6.033/lab/three` and decompress the archive file into a directory of your choice. In the `sfs/upt` directory, type `./configure` and then `gmake`. Then, in the `sfs` directory, type `./configure` and then `gmake`. On Solaris machines, you may need to add the `gnu` locker to run `gmake`.

Once you start adding your own source files, you will probably need `automake` and `autoconf`. You will have to add the `outland` locker for `automake`. If you are running Linux, you will need to make sure to have those utilities installed. You can get the latest `RPM` archive of the `automake` and `autoconf` from the your favorite RPM distributor. To add `.c` files for the `sfsfs` binary, edit `sfs/sfsfs/Makefile.am`. Add new `.c` files to sfsfs_SOURCES and new `.h` files to `noinst_HEADERS`. Then run `gmake`. For more information, type `C-h i` in `emacs` and look for `automake`.

To summarize, the following instructions will build the package on either the x386 or Solaris platforms after you decompress the archive:

1. `add -f gnu outland`
2. `cd upt; ./configure; gmake`
3. `cd ..; ./configure; gmake`

The `-f` option will add the lockers `gnu` and `outland` to the beginning of your PATH.

As you build this system, you will probably notice a few warnings. Currently, the NFS loopback server has been tested under Linux 2.2.14 (RedHat 6.1) and Solaris 5.6.

9.2 How to run the NFS loopback server

A few things to do before you actually run the file system:

1. Make sure that the RPC portmapper (`portmap` under Linux, `rpcbind` under Solaris) is up. You can verify that the RPC portmapper is up by running the `rpcinfo` command. Portmapper is often a lead-in for security problems. We recommend you disable `portmap` when you are not using it.

2. Create a directory to be used as the mount point for your file system. For example, you can create a directory `/tmp/sfs` or `/sfs`.

3. Create a file to be used as the disk for the file system. You can use the `mkfs` command located in the `sfs/sfsfs` directory. For example, the following command creates a 1 megabyte file system in `/tmp/disk`:

```
sfsfs/mkfs 2048 /tmp/disk
```

Once you've created a mount point and a disk for the file system, you're ready to start the NFS server and mount the file system. The `sfslaunch` program will do both operations for you. You need only specify the proper parameters to `sfslaunch`. A sample invocation (assuming that you are in the `sfs/sfslaunch` directory) is

```
./sfslaunch ../sfsfs/sfsfs /tmp/disk /tmp/sfs
```

Running `sfslaunch` will require super-user privileges, which are available on Athena by using the `su` command. You can find out the password by typing `tellme root` on Athena. Do not login as root, instead `su` to root. Please do not modify the configuration of the workstation as root.

You can now browse the `/tmp/sfs` directory. For more information about how to run the system, see the README and USING files included in the distribution.

9.3 Understanding the NFS loopback server

There is a README file in the sfs directory the tar file created. You might find it useful.

`sfs/sfsfs/nfsserv.c` contains the NFS server stubs to sfsfs, the server you will run as the NFS loopback server. `sfs/sfsfs/nfsserv.c` calls functions in `sfs/sfsfs/sfsimpl.c`. You should need to only modify functions in `sfs/sfsfs/sfsimpl.c` that are called by `sfs/sfsfs/nfsserv.c` (i.e., those specified in Section 3.1) and add your own `.c` files to implement your in-memory and on-disk structures. However, as always, you are free to modify anything.

In its current state, this distribution implements a very rudimentary file system. The file system is capable of storing only 1 regular file. For example, in the `sfs_create` function in `sfs/sfsfs/sfsimpl.c`, when an user tries to create a second file, an error NFSERR_NOSPC (no space on device) is returned. The basic read/write I/O operations have been implemented. You should read and understand these operations and model your read/write operations after it. The end of this handout shows some sample output describing how you can use the file system in this rudimentary state.

10 Sample Output

Here's some sample output from the rudimentary file system that's currently implemented in the NFS loopback server distribution. The file system was started in one shell and some simple UNIX file operations were performed in another shell. The text from the shell where the file system was started is preceded by a > character on each line.

```
> [root@hu-home sfslaunch]# ./sfslaunch ../sfsfs/sfsfs /tmp/disk /tmp/sfs
> Using /tmp/sfs as mount point for file system.
> sfslaunch: running ../sfsfs/sfsfs (pid 4592, port 2491)
> Using /tmp/disk as disk.
> sfsfs: Reading from disk
> Hostname: (sfs)
> NFS request 1  (uid 0, gid 0)
> sfsfs: Reading from disk

[jhu@hu-home /]$ cd /tmp/sfs
[jhu@hu-home sfs]$ ls -l
total 0
[jhu@hu-home sfs]$

> Hostname: (sfs)
> NFS request 1  (uid 0, gid 0)
> sfsfs: Reading from disk
> NFS request 4  (uid 11058, gid 100)
> sfsfs: Found one in buffer cache
> NFS request 1  (uid 11058, gid 100)
> NFS request 1  (uid 11058, gid 100)
> NFS request 16  (uid 11058, gid 100)
> NFS request 1  (uid 11058, gid 100)

[jhu@hu-home sfs]$ cat > file
The quick brown fox jumped over the lazy 6.033 TAs.
^D
[jhu@hu-home sfs]$

> NFS request 4  (uid 11058, gid 100)
> NFS request 9  (uid 11058, gid 100)
> sfsfs: Found one in buffer cache
> NFS request 4  (uid 11058, gid 100)
> NFS request 8  (uid 11058, gid 100)
> sfsfs: write: 0, 52
> sfsfs: Reading from disk
> sfsfs: Found one in buffer cache

[jhu@hu-home sfs]$ ls -l
total 1
-rw-------   1 jhu      users          52 Apr  7 00:06 file
```

```
[jhu@hu-home sfs]$

> NFS request 4  (uid 11058, gid 100)
> NFS request 1  (uid 11058, gid 100)
> NFS request 1  (uid 11058, gid 100)
> NFS request 16  (uid 11058, gid 100)
> NFS request 4  (uid 11058, gid 100)
> NFS request 1  (uid 11058, gid 100)

[jhu@hu-home sfs]$ cat file
The quick brown fox jumped over the lazy 6.033 TAs.
[jhu@hu-home sfs]$

> NFS request 4  (uid 11058, gid 100)
> NFS request 4  (uid 11058, gid 100)
> NFS request 1  (uid 11058, gid 100)
> NFS request 6  (uid 11058, gid 100)
> sfsfs: read: 0, 1024
> sfsfs: Found one in buffer cache

[jhu@hu-home sfs]$ cat > file2
file2: No space left on device.
[jhu@hu-home sfs]$

> NFS request 4  (uid 11058, gid 100)
> NFS request 9  (uid 11058, gid 100)
> sfsfs: File exists

[jhu@hu-home sfs]$ rm file

> NFS request 4  (uid 11058, gid 100)
> NFS request 4  (uid 11058, gid 100)
> NFS request 4  (uid 11058, gid 100)
> NFS request 4  (uid 11058, gid 100)
> NFS request 10  (uid 11058, gid 100)
> sfsfs: Found one in buffer cache

[jhu@hu-home sfs]$ ls -l
total 0

> sfsfs: Found one in buffer cache
> NFS request 4  (uid 11058, gid 100)
> NFS request 1  (uid 11058, gid 100)
> NFS request 1  (uid 11058, gid 100)
> NFS request 16  (uid 11058, gid 100)
> NFS request 1  (uid 11058, gid 100)
```

11 Random bugs about the loopback server

If you use RH 4.2, you might need to use our `rpcgen` that is included in `sfs/rpcgen`. If `gmake` complains about "int i" after the expression "(void)buf", you will need to run `cd rpcgen; gmake`. Then replace `/usr/bin/rpcgen` with your new binary.