SIFS

Phase 1 Due: October 14, 2007 at midnight
Phase 2 Due: December 5, 2007 at midnight

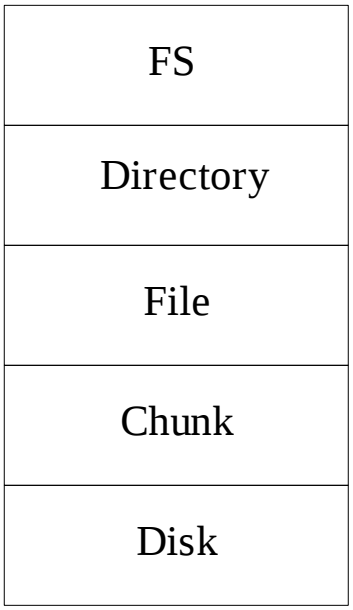
1. Overview

This semester you will implement a *single-instance file system* (SIFS) that stores only one copy of data, even if the data appear in multiple files. This is a hot topic in the data storage industry, as storing only a single copy of data reduces the amount of storage media required, which in turn reduces power, space, and most importantly, IT costs. IBM refers to this problem as "data de-duplication". You will implement SIFS using FUSE (Linux's user-level filesystem functionality) so that unmodified applications can make use of SIFS.

This project will be done in two phases and in groups of up to three.

2. Architecture

You will implement your SIFS as a process that reads and writes the disk in response to requests made by FUSE, which in turn are caused by file accesses by application programs. I will provide you with a Disk module that reads and writes a virtual disk stored in a regular Unix file. You must use FUSE and you must use Disk, what you do between these two is up to you, although you must justify the design in your design document. I would suggest the following hierarchical architecture but you are welcome to develop your own. I've sketched out the basic APIs for the layers but not provided all the details.



2.1 Disk

You will be provided with the Disk layer; do not make any changes. The Disk API is defined in disk.h, which is also provided and you should not change. The basic API is:

Disk_Read(disk, offset, length, buffer)

Disk_Write(disk, offset, length, buffer)

2.2 Chunk

The Chunk layer is responsible for storing variable-size chunks of data identified by a unique 160-bit chunk ID (CID). The CID for a chunk is the SHA-1 hash of the chunk's contents – you may assume that collisions do not happen, allowing each chunk to be uniquely identified by its CID. The one exception is CID '0', whose contents need not have a SHA-1 of '0'. This well-known CID gives the File layer a starting point for accessing chunks.

The Chunk layer allows the higher-levels to read and write chunks. To do this it must manage the disk space, keeping track of which space is free and allocating space for new chunks, as well as keeping track of where on disk chunks are stored. One of the challenges is designing a data structure for mapping CIDs to disk addresses that is compact and doesn't require too many disk access. The Chunk layer must also maintain a reference count on individual chunks. When a chunk is written that already exists its reference count is simply incremented. Deleting a chunk decrements its reference count; when the reference count reaches zero the chunk is deleted from disk. The Chunk layer also contains a cache of recently-used chunks to reduce disk accesses. The basic Chunk API is:

Chunk_Read(CID, length, buffer)

Chunk_Write(CID, length, buffer)

Chunk_Delete(CID)

2.3 File

The File layer implements the file abstraction. The File layer stores everything in chunks and does not access the Disk layer directly. Conceptually, a file consists of a sequence of chunks, the boundaries of which are determined using Rabin fingerprinting. This sequence is stored in an inode; again, one of the challenges is storing the sequence efficiently considering it changes when the file is modified. The inode also contains other metadata about the file such as the number of links to it, its size, etc. Files are uniquely identified by a file ID (FID). The basic File API is:

File_Read(FID, offset, length, buffer)

File_Write(FID, offset, length, buffer)

FID = File_Create()

File_Delete(FID)

2.4 Directory

A directory is a special type of file that maps names to FIDs. It therefore accesses only the File layer and not the Chunk nor Disk layers. The basic API is:

Dir_Create(FID, name, target_FID)

FID = Dir_Lookup(FID, name)

2.5 FS

The FS ("filesystem") layer implements the rest of the functionality that spans directories and files. For example, renaming a file involves the file and two directories (the old parent and the new parent). Also, the FUSE API passes in pathnames for files and directories, so the FS layer should probably have a library routine that repeatedly calls Dir_Lookup to resolve the name into the proper FID (creating it if necessary). The FS layer consists of FUSE stubs plus library routines and doesn't have a proper API as such. The FS layer accesses the Directory and File layers, but not anything below them.

3. FUSE

Your SIFS consists of a user-level process that makes use of FUSE to service filesystem operations made by unmodified application programs. Your SIFS process need not be multi-threaded. You must implement at least the following FUSE routines:

- getattr
- readlink
- mkdir
- unlink
- rmdir
- open
- read
- write
- statfs
- release
- opendir
- readdir
- releasedir
- init
- destroy
- create

Your SIFS process has the following command-line syntax:

sifs [options] *file mountpoint*

Where the options consist of:

-c *num*, --cache=*num*

*S*ize of the cache in the Chunk layer, in chunks.

The *file* argument specifies the name of the virtual disk file, and *mountpoint* specifies where the sifs filesystem should be mounted. Sifs must pass the '-f' argument to fuse_main so that the process runs in the foreground instead of detaching and running in the background.

4. mksifs

You must write a command to create and format a disk for SIFS. It should zero-out the disk, then initialize all your on-disk data structures so that that your SIFS process can access it properly. The initial filesystem should be empty, consisting of only the root directory and the '.' and '..' entries. The mksifs command has the following syntax:

mksifs [options] *file*

Where options consist of:

-i *num*, --inodes=*num*

Maximum number of inodes in the filesystem. The default is 1000.

-m *size*, --min-chunk=*size*

Minimum chunk size, in bytes. The default is 4KB.

-M *size*, --max-chunk=*size*

Maximum chunk size, in bytes. The default is 16KB.

-s *sectors*, --sectors=*sectors*

Size of the disk, in sectors. The default is 1000.

The mksifs command should exit with a return status of '0' if there are no errors, '1' otherwise.

5. sifsck

You must write a command that verifies the integrity of a SIFS disk and prints out any errors it finds. The things it should check include verifying the chunk database (e.g. All chunks on disk have the proper CID, all chunks have the proper reference count, and all free space on the disk is accounted for properly), verifying files(e.g. All chunks of all files exist on disk and have the proper reference count, file sizes are accurate), and verifying directories (e.g. All files have at least one link, link counts are accurate). The sifsck program need not repair the disk, but should be pedantic about checking the integrity of the disk. Keep in mind that your SIFS process is likely to die in mysterious ways if the disk is corrupted – it is much better to put the effort into writing sifsck than debugging SIFS. The syntax is:

sifsck *file*

The sifsck command should exit with a return status of '0' if there are no errors, '1' otherwise.

6. Testing

I will provide test cases, but any program should be able to access your filesystem.

7. Phase 1

For this phase you should have the following implemented for each of the layers:

Chunk

Able to read and write chunks, perhaps using an inefficient data structure such as scanning the disk. Chunk reference counts and deletion not implemented.

File

Able to read and write files. Inode data structure perhaps inefficient. Overwrite and truncation not implemented. Multiple links not necessary.

Directory

Support a flat namespace in which files are named "/123" where "123" is the inode number.

FS

Whatever is necessary to implement the above functionality.

8. Extra Credit

With my permission you may implement the following for extra credit. In general you should complete the rest of the assignment before doing any extra credit.

- **Crash Recovery.** Recovery from a crash during SIFS operation. This means checking and fixing the on-disk data structures before resuming.
- **Multi-threading**. Multi-thread the SIFS process so that it can concurrently handle requests.
- **Chunk Logging.** Treat the disk as a log into which chunks are stored. Similar to the log-structured filesystem except this is a log-structured chunk store.
- **Additional FUSE routines.** Implement more of the FUSE routines such as symbolic links, file locks, mmap, etc.

9. Logistics

This project will be done in teams of no more than three. Since working in groups means that there is a danger of one person not carrying his or her load, I'm likely to quiz each group member orally on any part of the system during the final demos. Each group member must be familiar with the overall design and structure of their group's project.

Turn in your phases using the names *552sifs_1* and *552sifs_2*. You must include a design document (PDF preferred) for both phases. Be sure to include what does and doesn't

work, as well as any cool features you implemented. You will demo the project to me at the end of the semester. Your assignment will be graded on lectura, so verify that it works there before you turn it in (try this out early -- there are always last-minute glitches).