DESIGN AND IMPLEMENTATION OF SMAFS

Dhirendra Singh Kholia

Department of Computer Science, UBC

1. Abstract

smaFS is a generic file system overlay, capable of providing file-level snapshots, transparent versioning and recovery to any-point-in-time on top of any existing file system. smaFS aims to be non-invasive and extensible in nature, possibly at the cost of performance and efficiency.

This report describes the design and implementation details of smaFS. It also addresses topics commonly associated with versioning file systems like efficient storage problem, long term retention policies, transparent compression, data encryption and data integrity checks.

2. Motivation

smaFS is inspired by the lack of an existing production grade file system capable of providing a combination of file-level snapshots, transparent versioning and portability.

smaFS tries to guarantee that the user wouldn't lose any modifications made to a file during its lifetime. smaFS also aims to provide guaranteed data recovery from 0-day malware attacks.

Some existing systems and their drawbacks are listed below:

ZFS: ZFS does not do transparent versioning and it requires a dedicated block device. Also it cannot be natively ported to run on Linux due to licensing problems.

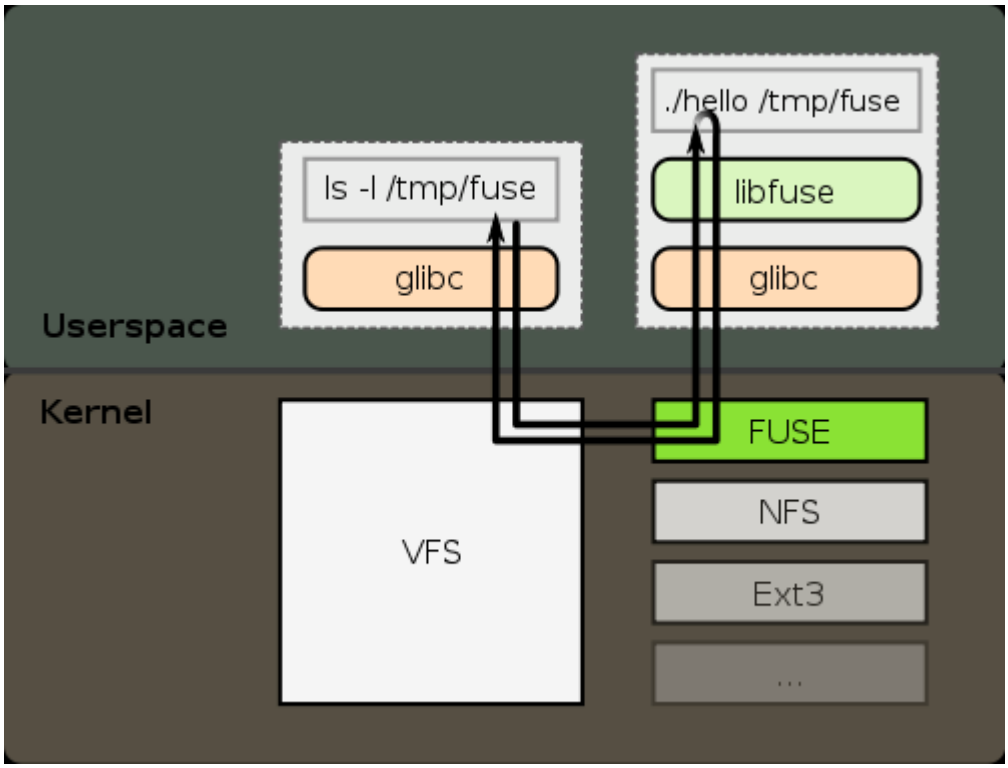
Dropbox: It supports transparent versioning and centralized backups, works on top of any existing file system and is portable. However it's a proprietary solution and requires internet connectivity to work. The low frequency of snapshot process can lose rapidly changing data.

HAMMER: It supports full-history retention transparently and data integrity checks. It however it too requires a dedicated block device to work. In general, the design doesn't seem to emphasize extensibility.

Elephant File System: It is a research file system and as such not widely used. Porting will be required in order making it work on modern operating systems. It too requires a dedicated file system on a block device to work.

3. Design and implementation of smaFS

smaFS is a userspace filesystem built using FUSE (Filesystem in Userspace) technology. FUSE consists of two main parts: The FUSE kernel module and the FUSE library (libfuse). Essentially FUSE allows a filesystem to be built and operated in user-space.



(Source: http://en.wikipedia.org/wiki/Filesystem_in_Userspace)

Being able to run a file system in userspace allows rapid prototyping and debugging using standard tools like gdb and Valgrind. Both these tools were used extensively to test smaFS.

smaFS is an overlay file system in a sense that it operates on top of existing filesystem. FUSE allows a developer to hook filesystem related system calls and redirect them to a userspace functions. This mapping in maintained in struct fuse_operations which looks like:

```
smaFS_oper.init    = smaFS_init;
smaFS_oper.getattr = smaFS_getattr;
smaFS_oper.access  = smaFS_access;
smaFS_oper.readdir = smaFS_readdir;
smaFS_oper.mknod   = smaFS_mknod;
smaFS_oper.mkdir   = smaFS_mkdir;
smaFS_oper.symlink = smaFS_symlink;
smaFS_oper.unlink  = smaFS_unlink;
smaFS_oper.rmdir   = smaFS_rmdir;
smaFS_oper.rename  = smaFS_rename;
```

smaFS is a transparent versioning system and it maintains versions (history) and associated information (metadata) in “store” which itself is stored in “.store” directory. This store is hidden from user view by trapping and denying access to it in smaFS_readdir() module.

```
if(!strcmp(de->d_name, ".store"))
    continue;
```

Users can access previous versions of a file by using a special filename format “filename#<revision number”. This “transparent” access of history is made possible by a translation mechanism implemented in smaFS in get_translated_path function of translate.c.

The history and the metadata are protected from direct write access by user and unauthorized users.

```
if(strstr(path, ".store") && fuse_get_context()->uid !=0)
    return -ENOENT;
```

smaFS utilizes a “cache” which stores a history of operations done on a filename. This cache makes versioning efficient by skipping creation of unnecessary copies. The core idea is that if truncate () is followed by open() in write mode then we can skip creating a version. Similarly smaFS detects such combinations of operations on files and skip creating versions wherever possible. The implementation of “cache” took the longest time to design, implement and test.

The “cache” also allows the implementation of “lazy metadata commit” idea in future. Currently the cache data structure is not thread-safe!

smaFS guarantees data integrity. It can detect “bit rot” and unauthorized tampering of underlying filesystem. This is made possible y using checksums which are maintained for every version in the file’s history. smaFS uses a flexible “hashlib” library which allows the checksum/hashing function to be as strong as SHA 3 proposal Blake-64 or as fast as Fletcher-32. Both Blake-64 and Fletcher-32 have been implemented in smaFS.

User can see all the versions of a file that exist by running the “versions” utility program. Another interesting utility program is “reaper” which is capable of compressing file’s versions (history) and thus reducing storage requirements. Currently reaper use “lzo” compression scheme which allows almost real-time compression performance. The “reaper” also does “thinning” operation (not implemented currently) which essentially removes versions which are no longer required by the user.

smaFS is extensible by nature and such utility programs can be implemented rapidly. Both “versions” and “reaper” utility programs were designed and implemented in under five hours.

4. Current Limitations and Future Work

a. The current version of smaFS does extra reads for implementing compression and hashing. Instead, hashing and compression should be performed within make_copy() function loop in a block wise fashion. The implementation of this idea should be straightforward and save on multiple duplicate file read operations. Care in implementation must be taken to allow for different optimal block sizes for different operations.

b. Lazy metadata commit: Metadata can be cached in memory and flushed to disk later. This should boost the performance on update intensive workload.

c. The store directory where metadata and version history is stored is weakly protected by setuid utilities. A better security model should be researched.

d. reaper utility currently requires complete re-write of metadata file. An alternate strategy for maintaining metadata should be investigated.

e. smaFS runs on 32-bit as well as 64-bit systems fine. However smaFS's metadata handling is not 64-bit clean. It should be straightforward to fix this though.

f. Add "thinning" operation support to reaper. In addition it is also possible to maintain only the diffs/deltas instead of complete files.

g. Currently the "cache" data structure is not thread-safe. The future versions of smaFS will implement a thread-safe "cache". The resulting locking and scaling issues are likely to be interesting ☺

h. Add support for directory versioning. This idea hasn't been thought about much although it seems straightforward.

5. Conclusions

smaFS has been a "fun" project in every phase. I learned some new tools, discovered lot of ideas-which-failed and some ideas-which-worked. I still strongly believe that smaFS idea has real utility in daily computing. In future, I would love to work on interesting file systems.

6. References

1. FUSE wiki: http://sourceforge.net/apps/mediawiki/fuse/index.php?title=Main_Page
2. CopyFS: http://n0x.org/copyfs/
3. SGI STL Thread-Safety: http://www.sgi.com/tech/stl/thread_safety.html
4. LZO real-time data compression library: http://www.oberhumer.com/opensource/lzo/
5. SHA-3 proposal BLAKE: http://www.131002.net/blake/
6. The ChaCha family of stream ciphers: http://cr.yp.to/chacha.html
7. Fletcher's checksum: http://en.wikipedia.org/wiki/Fletcher's_checksum