Design of a Framework for Compressed-Encrypted-Stackable File Systems

Sapna Bafna IEEE Student Member No. 41292943, sapnabafna@ieee.org
Abhinay Kampasi IEEE Student Member No. 41293277, abhinaykampasi@ieee.org
Aditya Kulkarni IEEE Student Member No. 41293286, adityakulkarni@ieee.org

PICT IEEE Student Branch,
Pune Institute of Computer Technology,
Dhankavdi, Pune-411043, India.

Abstract: *The need for security of information and its efficient storage has gained significance over the years. Existing file systems can be enhanced to do this. In this paper we propose a file system framework, the Compressed-Encrypted-Stackable File System (CES FS). This file system framework aims to provide a secure interface between the user and the device-level file system. The framework supports automatic compression and encryption of data obviating the need for compression and encryption utilities. The file-system transpires to be very efficient and easy to use, coupled with high security and efficient storage management. The CES file-system framework provides developers with an interface that enables them to provide new compression and encryption algorithms and hence gives the developer the flexibility to select from a range of compression and encryption algorithms.*

Index Terms: *Virtual stackable file system, vnode interface, intelligent file system*

1 Introduction

The need for a secure system is paramount today. It is critical for the system to provide efficient and secure data storage. Though the currently available file systems in Unix and NT provide security in terms of restricting access to data on the disc to unauthorized users, they do not change the data format in any way. If the file system encrypts the data before storing it on the disc, then even a direct hit on the disc would not leak out the information.

Another important issue is how efficiently the file system stores data. The NASA Goddard Conference on Mass Storage Systems and Technologies [10] emphasized on current and future practical solutions addressing issues in data management, long-term retention of data and data distribution. One of the key metrics that file systems will be rated by will be the efficient utilization of storage resources. Today there are a host of compression utilities like WinZip [9], which compress data and thus help the user in saving disc space. There are merits and demerits of using a separate utility for compressing data. The key advantage is that the user can decide the file to compress according to his/her requirements. On the other hand, the demerit is that the user has to follow a tedious procedure for compressing the files. Things would be much more efficient if the file system that is operating, performs the job at the kernel level and automatically provides compression of data.

The CES file system framework aims at providing this. In the current scenario the most popular method of adding functionality to file systems it to either build new ones or to add functionality to existing ones by modifying them. For example, it is desirable to extend existing file systems to include new features such as encryption and compression. The CES file system framework will produce an extension to VFS compatible file systems in Linux. It is a file system that is stacked a layer above the physical file system. When a file system is mounted on top of any other file system, the stackable file system adds a performance overhead of only 1–2 % for accessing the other file system [1] and all the new features can be included in the stackable layer.

All the invoked system calls [8] will pass through the CES layer before passing through the underlying file system layers. This concept is exciting because we can leverage existing file systems and add functionality such as encryption and compression.

The currently available file systems provide additional features. Each file system is unique and serves a specific purpose when used independently. However we propose that instead of writing all these different file systems, you could write a file system, which sits on top of the already existing file system and helps to overcome its limitations. Figure1 shows the proposed topology of our file system stack.

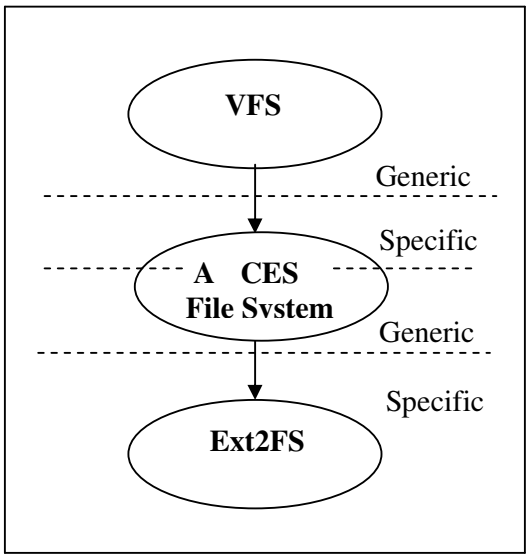

Fig. 1. File system stacking, where CES FS fits inside the kernel.

The VFS (Virtual File System) [3], [4] is the software layer in the kernel that provides the file system interface to user-space programs. It also provides an abstraction within the kernel, which allows different file system implementations to coexist [2]. As our file system framework is stackable it only has to implement the vnode [2] operations that it wishes to change. Other operations are automatically passed through between stacked layers. This option is similar to that of object-oriented programming models, where a subclass can use the methods of the superclass [1].

CES FS framework handles many of the internal details of operating systems, thus freeing developers from dealing with kernel specifics and provides them with a framework to build a file system, which has facilities for compression and encryption. This framework enables the user to select from a variety of compression and encryption algorithms. Moreover the user has an alternative of having different combinations of algorithms from amongst those provided by the file system framework. As multiple algorithms can be used, the user has the flexibility of selecting an encryption algorithm and one-or-more compression algorithms of his choice, depending on the importance of the file.

Thus the proposed file system framework can successfully accomplish the need for security and storage management. This framework can be extended to become an intelligent file system that understands patterns

and accordingly decides the level of compression to be applied to it. Such an implementation could then be considered to serve the purpose of an ideal file system.

2 Design Goals

In this section we take a glance at the development and architecture of the CES file system framework.

2.1 Flexibility

Traditional file systems generally incorporate a single compression or encryption algorithm. The file system framework proposed by us takes a step ahead by providing the user a wide range of algorithms. The user has the choice of selecting any one, or a combination of these algorithms. There is nothing like a standard algorithm that is used for all system calls (like read/write). The file system framework provides the user the liberty to choose the algorithms that he deems fit. The user also has the freedom to skip compressing or encrypting his files.

2.2 Programming Model

In the file system framework an interface has been provided by which the user can select any algorithm from the available list. If the developer desires to use an algorithm apart from this list then he/she can do so by making an appropriate entry in the existing list. Along with this entry, he/she has to include the corresponding functions with predefined parameters, i.e. the functions for encoding and decoding included by him/her should accept some parameters and return some values in a standard format prescribed by the CES file system framework.

2.3 Performance

User-level file systems are the slowest because each exchange of information between the kernel and the file-system server causes a context switch [6]. File systems are fastest when they run in the kernel and interact directly with device drivers. Stackable file systems are generally fast because they run in the kernel and are much faster than their user-level counterparts.

We propose to blend the advantages of both these techniques. Here the code is written at the user level and implemented at the kernel level thereby resulting in easier coding and faster execution. This implicitly means that the compression and encryption will be done within the kernel.

2.4 Portability

This factor defines the ease with which code written for one file system can be ported to another.

User-level code is the easiest to port, because it is written just like any other user-level C program. For example, the Amd automounter has been easily ported to dozens of different Unix platforms and operating-system versions [1]. Low-level in-kernel file systems are the hardest to port because they depend on many kernel specifics and details of device-driver implementations [2]. Rarely do these file systems get ported to other platforms; often the effort is large enough that a complete rewrite is easier than porting.

Stackable file systems run as part of the kernel and as such also depend on kernel internals. The file system generated by our framework will be portable across different Unix systems. As it uses a stackable vnode interface, porting it to other platforms requires those platforms to support the same stacking API.

2.5 Error Checking

The user is given the flexibility to choose algorithms for compression and encryption. Thus it thus becomes mandatory to verify the entries provided by the user and check if any errors are encountered. In that case, the CES file system framework must handle the errors.

The primary error that can be detected by our error detection mechanism is the improper selection of an algorithm. While mentioning the algorithms to be used, if the user does not enter the correct algorithm or does not enter any algorithm at all, then the framework selects its default algorithm, Data Encryption Standard (D.E.S.) for encryption and Huffman algorithm for compression.

3 Design

This section will elaborate on the design goals mentioned briefly in the previous section. It describes the architecture and functionality of the CES file system framework.

3.1 Terminology

The design of the CES framework incorporates a thorough study and understanding of concepts like Virtual File System Interface, Stackable templates and other related terms.

3.1.1 Virtual File System

The Virtual File System is the software layer in the kernel that provides the file system interface to user-space programs [3]. It also provides an abstraction within the kernel, which allows different file system implementations to coexist. When you wish to mount a block device onto a directory in your file space, the VFS will call the appropriate method for the appropriate file

system. The dentry for the mount point will then be updated to point to the root inode for the new file system [6].

Basically, VFS is a generic section of file-system code in the (Unix) kernel, often called the upper-level file-system code because it is a layer of abstraction above the file-system specific code. In particular, when system calls begin executing in the kernel's context, the kernel then executes VFS code for those system calls. The VFS then decides which file system to pass the operation onto. The VFS is generic in that it does not contain code specific to any one file system; instead, it calls the predefined file-system functions that were given to it by specific (lower level) file systems [1], [2].

Vnodes are the primary objects manipulated by the VFS. The VFS creates and destroys vnodes. It is a handle to a file maintained by a running kernel. This handle is a data structure that contains useful information associated with the file object, such as the file's owner, size, last modification date, etc [2]. The Vnode object also contains a list of functions that can be applied to the file object itself. These functions form a vector of operations that are defined by the file system to which the file belongs. It fills them with pertinent information, some of which is gathered from specific file systems by handing the vnode object to a lower level file system. The VFS treats vnodes generically without knowing exactly, which file system they belong to.

The Vnode Interface is an API that defines all of the possible operations that a file system implements. This interface is often internal to the kernel, and resides in between the VFS and lower-level file systems. Since the VFS implements generic functionality, it does not know of the specifics of any one file system. Therefore, new file systems must adhere to the conventions set by the VFS; these conventions specify the names, prototypes, return values, and expected behavior from all functions that a file system can implement.

3.1.2 Stackable Templates

Stackable templates provide basic stacking functionality without changing other file systems or the kernel. This functionality is useful because it improves portability of the system. Such a template handles many of the internal details of operating systems, thus freeing developers from dealing with kernel specifics [7]. It provides a stacking layer that is independent from the layers above and below it .

In the generation of file system code, we have used Basefs [1] as a stackable template. Basefs appears to the upper VFS, as a lower level file system and to file systems below it as a VFS. Initially, Basefs simply calls the same vnode operation on the lower level file system. Basefs performs data reading and writing on whole pages. This simplifies mixing regular reads and writes with memory-mapped operations, and gives developers a single paged-based interface to work with.

To improve performance, Basefs copies and caches data pages in its layer. It will also cache pages of the layers below it, in case the lower-level file system does not do so directly (on some operating systems, the VFS is responsible for inserting pages into the cache, not the actual file system). Basefs saves memory by caching at the lower layer only if file data is manipulated and fan-in was used; these are the usual conditions that require caching at each layer. Basefs adds support for fan-out file systems natively. This code is also included conditionally, because it is more complex than single-stack file systems, adds more performance overhead, and consumes more memory. Basefs includes (conditionally compiled) support for many other features. This added support can be thought of as a library of common functions: opening, reading or writing, and then closing arbitrary files; storing extended attributes persistently; user-level utilities to mount and unmount file systems, inspecting and modifying file attributes, and more.

3.2 Code Generation

The C.E.S file system generates a lot of code using the File System Translator i.e FiST [1]. The FiST language is the first of the three main components of the FiST system. Figure 2 shows the Operational diagram of FiST.

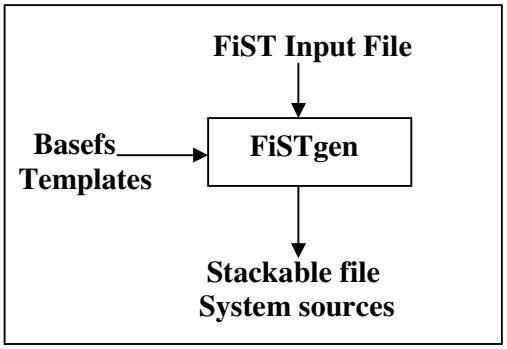

Fig. 2. FiST Operational Diagram

The FiST system is composed of three parts: the language specification, the code generator, and stackable file system templates. The overall operation is shown in figure 2. The figure illustrates how the three parts of FiST work together.

3.3 Implementation

The CES file system framework has a very user friendly API. The user has the option of selecting a name space on which he wants to mount the file system. He has a range of algorithms to select from and default ones are used if non are provided. Depending upon the algorithms selected by the user, the ones barring them are commented in the file, which will later be referred to by the file system to call appropriate functions upon invocation of system calls.

The format of the command entered by the developer is as follows:

[name space] [compression algorithm no. 1, 2,…]
[encryption algorithm no. a , b,…]

List of compression algorithms:
1. Huffman
2. BMP
3. RLE

List of encryption algorithms:
a. D.E.S.
b. Blowfish
c. Rot2

Figure 3 explains the directory structure before mounting while Figure 4 shows the same after mounting.

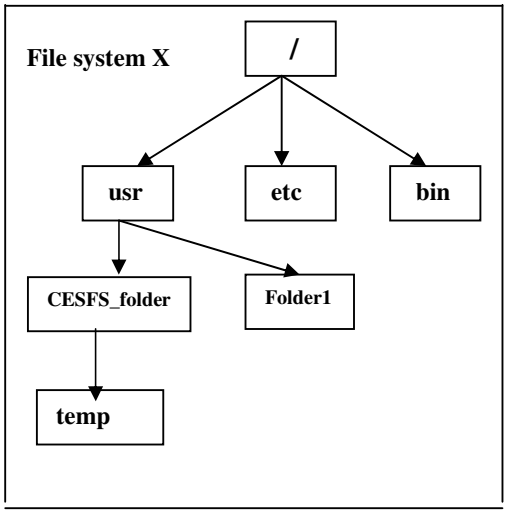

Fig. 3. Directory structure before mounting

Consider that the developer gives a command like:

/usr/CESFS_folder 1 3 a

Interpretation of the Command:
The developer prior to giving the command should create the CESFS_folder. After successful mounting [2], the folder will have type of file system as CES FS, with the mentioned algorithms being used for compression and encryption as parameters.

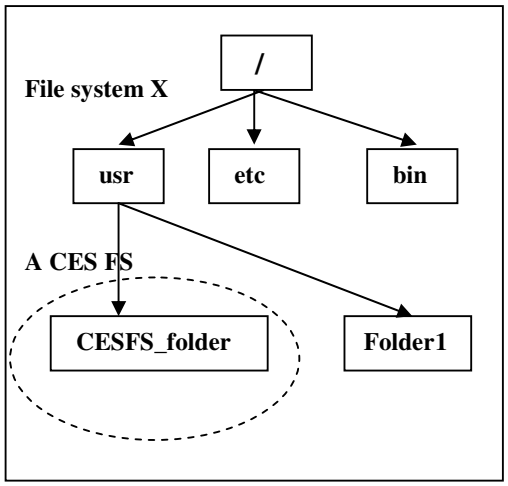

Fig. 4. Directory structure after mounting

All file system operations performed under the CESFS_folder will henceforth follow the specifications of the CES file system framework. When you unmount this file system, the CES FS will no longer serve the CESFS_folder. Hence any access to files under the CES_FS folder will result in the user reading encrypted data., if encryption was used when it was mounted.

After the file system has been mounted it is stacked on top of the lower level file systems. As we use a stackable interface, the system calls, which are executed for doing any operation, pass through our file system layer. It is at this time that we can call the required functions as per the users specifications and the output is given to the lower level file systems. The lower level file system is unaware of the source from where they are getting the input. Thus they just process the output of the file system generated by our framework, as they would normally have. As the CES file system framework is in the kernel, the overhead incurred is just 1-2% [1]. Figure 5 shows how the C.E.S framework will process a write system call.

As shown in the figure and with respect to the command given above, when a system call, which processes a request to save data on the disc, then it, passes through the CES layer. And the appropriate compression and encryption functions get called as per those specified by the user. It is important to note that the user has to specify this order of algorithms only at the time of mounting. In this case, our layer calls the functions of Huffman Compression, RLE compression and

DES encryption in order. Similarly when a read system is encountered then the reverse process occurs as shown in Figure 6.
Firstly DES decryption gets called, followed by RLE decompression and finally Huffman decompression is called.

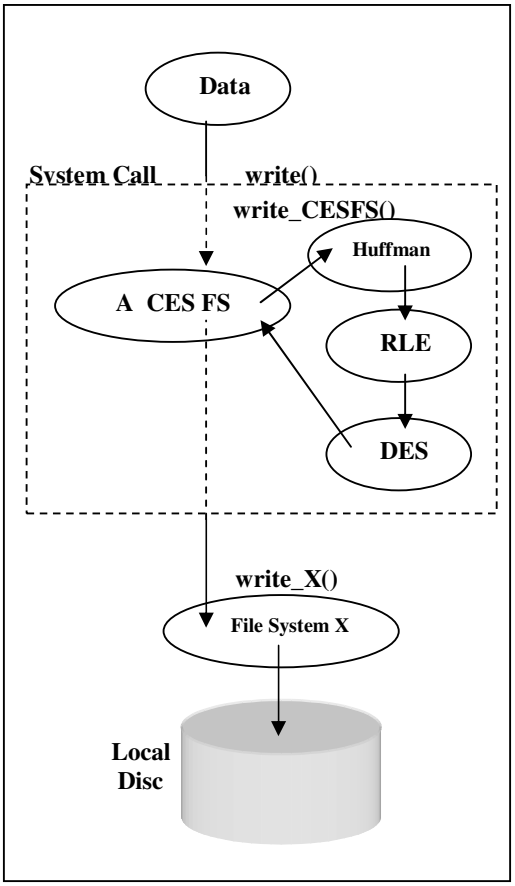

Fig. 5. Illustration of write system call

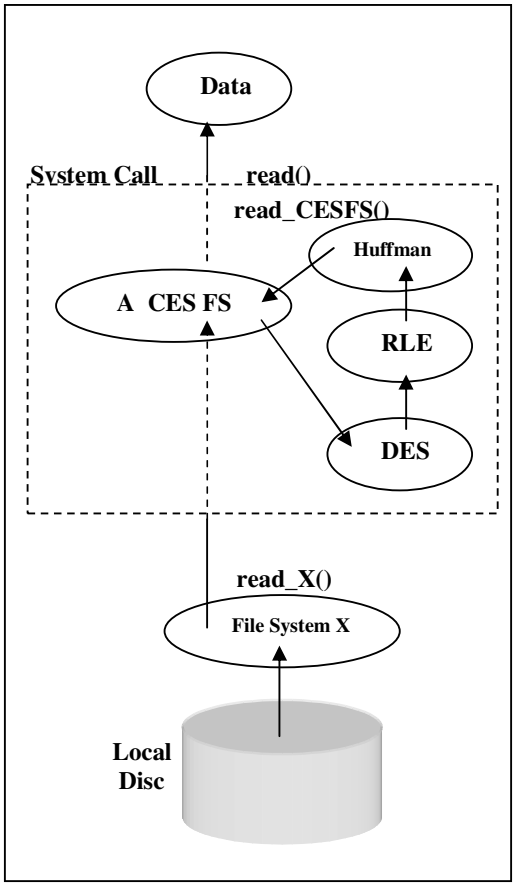

Fig. 6. Illustration of read system call

In case, the user enters a command of the format:

/usr/CES_folder 1 NULL

With this specification all the operations performed in the CES_folder undergo only Huffman compression and no encryption is performed.

As we can see from the figure above, the interface we propose will act as a perfect stack. It simply places itself between the user and the kernel, without either having knowledge about its existence. The primary advantage of the CES file system framework is that the existing file systems need not be changed at all. Moreover as the stacking takes place in the kernel, there is hardly any compromise on the speed. Security of the system is enhanced greatly and efficient storage management is achieved, improving system performance Besides this developers can provide new algorithms for compression and encryption with minimum effort thus building new file systems.

4 Expected Results

A detailed study of the file system framework proposed by us shows that it will improve the performance of the system to a great extent. By performance, we mean the security of the system coupled with efficient storage management. The initial results have been encouraging. Figure 7 shows the system time comparison between various CES file systems [1].

We hope to achieve improvement in system security as well as storage management. This improvement when compared to the Ext2 [5] file system also results in performance overhead. As all system calls have to pass through the CES layer, a small amount of additional time is required to perform the desired operations. However the advantages far outweigh these limitations of the file system.

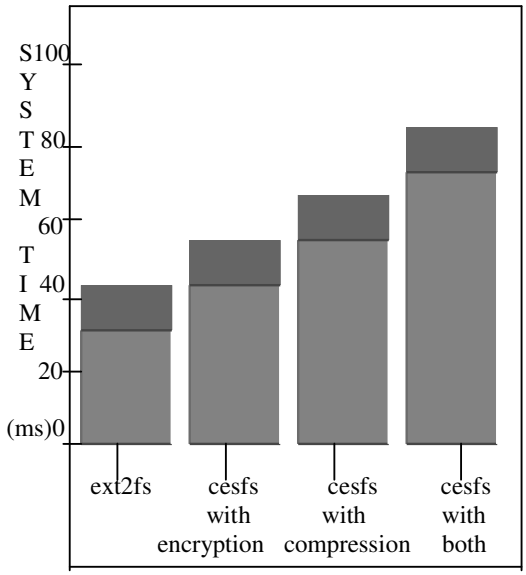

Fig. 7. System time for retrieving file attributes using lstat system call

We anticipate that the additional time taken for encryption will be slightly less than that required for compression. The framework promises to be very successful because it is an enhancement and not a replacement to existing file systems, and as the code resides in the kernel, the overhead incurred is low.

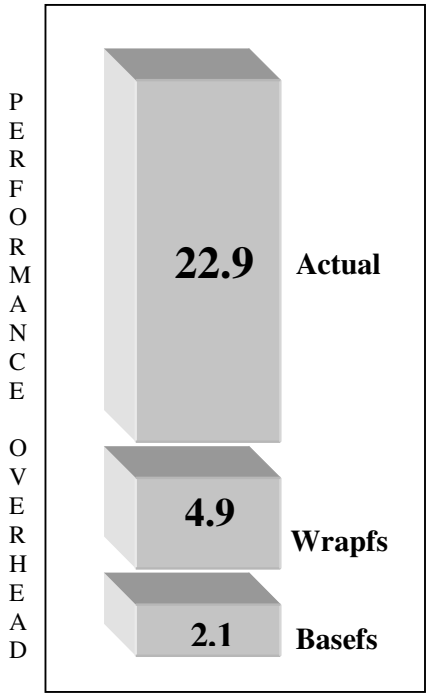

Fig. 8. Performance overhead of a CES FS across three operating systems

The overhead will be different across different Unix platforms [2], [7]. Figure 8 shows the proposed behavior of our file system across the basefs, wrapfs and actual operating systems [1]. We have based our assumption on performance figures of file systems produced by FiST.

5 Future Scope

The proposed framework will generate file systems, which could be called as *complete file systems.* We are currently working on enhancing the concept of CES FS by making it an intelligent file system.

5.1 Intelligent CES FS

We have seen that CES FS applies the encryption and compression algorithms to every file within the name-space, irrespective of its access frequency. In broad terms, it will compress every file to the same extent whether it is accessed once a year or once a day.

We are working on heuristics that will enable us to evolve CES FS into an intelligent file system. By 'intelligence' we mean that the file system keeps a track of how frequently a file is used and accordingly decides the level of compression to be applied to it. For instance, a particular which is accessed once a month would be reduced by 90% of it's original size while a file which is used once a week would be shrunk by 10% of its size. This is because compression requires considerable CPU utilization. Additionally, some files, which are used frequently, would be stored in some sort of a cache, with minimal compression so that they can be accessed with minimal delay.

6 Conclusion

The basic aim of working on the CES FS was to design a file system framework, which would serve the purpose of filling the loopholes of existing file systems enable developers to leverage existing stable file systems by providing them with a means to incorporate both security through encryption and efficient data storage through compression. The developer no longer will require intricate knowledge of file system internals to provide this functionality. The entire process of writing a file system with the desired compression or encryption algorithm will require writing 2 simple functions which will provide the ability to encrypt and decrypt and the other which will provide the compression and decompression facility.

We have been successful in doing so by designing a file system framework that builds itself on existing file systems providing additional benefits. The CES file system automatically

encrypts and compresses the files stored in its name-space. Thereby providing optimum security and efficient use of storage resources.

With more work on this in the future, we aim to build an intelligent file system that will selectively compress files after studying usage patterns.

Appendix

The CES file system framework makes use of the concept of virtual file system. A clear understanding of the VFS interface [1], [2], [4] is essential for getting acquainted with the functionality of our file system framework.

Struct VFS

An instance of the `vfs` structure exists in a running kernel for each mounted file system. All of these instances are chained together in a singly-linked list. The head of the list is a global variable called `root_vp`, which contains the vfs for the root device. The field `vfs_next` links one vfs structure to the following one in the list.

```
typedef struct vfs
 {
 struct vfs          *vfs_next;
 struct vfsops       *vfs_op;
 struct vnode        vfs_vnodecovered;
 u_long              vfs_flag;
 u_long              vfs_bsize;
 int                 vfs_fstype;
 fsid_t              vfs_fsid;
 caddr_t             vfs_data;
 dev_t               vfs_dev;
 u_long              vfs_bcount;
 u_short             vfs_nsubmounts;
 struct vfs          *vfs_list;
 struct vfs          *vfs_hash;
 kmutex_t            vfs_reflock;
 } vfs_t;
```

Struct Vfsops

The vfs operations structure (struct vfsops, seen below) is constant for each type of file system. For every instance of a file system, the vfs field vfs_op is set to the pointer of the operations vector of the underlying file system.

```
typedef struct vfsops
 {
 int       (*vfs_mount)();
 int       (*vfs_unmount)();
 int       (*vfs_root)();
 int       (*vfs_statvfs)();
 int       (*vfs_sync)();
 int       (*vfs_vget)();
 int       (*vfs_mountroot)();
 int       (*vfs_swapvp)();
 }vfs_ops_t;
```

Struct Vnode

An instance of struct vnode exists in a running system for every opened (in-use) file, directory, symbolic-link, hard-link, block or character device, a socket, a Unix pipe, etc.

```
typedef struct vnode
 {
 kmutex_t          v_lock;
 u_short           v_flag;
 u_long            v_count;
 struct vfs        *v_vfsmountedhere;
 struct vnodeops   *v_op;
 struct vfs        *v_vfsp;
 struct stdata     *v_stream;
 struct page       *v_pages;
 enum vtype        v_type;
 dev_t             v_rdev;
 caddr_t           v_data;
 struct filock     *v_filocks;
 kcondvar_t        v_cv;
 } vnode_t;
```

Struct Vnodeops

An instance of the vnode operations structure struct vnodeops, listed in exists for each different type of file system. For each vnode, the vnode field v_op is set to the pointer of the operations vector of the underlying file system.

```
typedef struct vnodeops
 {
 int       (*vop_open)();
 int       (*vop_close)();
 int       (*vop_read)();
 int       (*vop_write)();
 int       (*vop_ioctl)();
 int       (*vop_setfl)();
 int       (*vop_getattr)();
 int       (*vop_setattr)();
 int        (*vop_lookup)();
 int       (*vop_link)();
 int       (*vop_rename)();…
 }vnodeops_t;
```

These structures lay the foundation of a stackable file system. The CES FS uses these extensively to form a layer above the underlying file system inside the kernel.

Acknowledgement

We would like to thank Mr. Aditya Kini for his guidance and encouragement throughout the development of this file system framework. We are also grateful to Mr. Erez Zadok for answering our queries relating to FiST. A special thanks to Mr. Gangadhar Hariharan and Mr. Devendra Desai for their help during our work on the CES project.

References

[1] E. Zadok, " FiST: A System for Stackable File-System Code Generation," Thesis Computer Science Department Columbia University , New York, NY10027, May 2001. http://www.cs.columbia.edu/~ezk/research/fist/

[2] U. Vahalia, *UNIX Internals, The New Frontiers,* Prentice-Hall, Upper Saddle River, New Jersey 07458, 1996.

[3] D. S. H. Rosenthal, "Requirements for a 'Stacking' Vnode/VFS Interface," UNIX International, 1992.

[4] R. Gooch, " Overview of The Virtual File System," July 1999. http://www.atnf.csiro.au/~rgooch/Linux/vfs.txt

[5] R. Card Laboratoire MASI-Institut Blaise Pascal, T. Ts'o, Massachussets Institute of Technology, S. Tweedie , University of Edinburgh, " Design and Implementation of Second Extended File system," Proceedings of the First Dutch International Symposium on Linux, ISBN 90-367-0385-9. http://web.mit.edu/tytso/www/Linux/ext2intro.htm

[6] M. J. Bach, *The Design of The UNIX Operating System,* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[7] A. S. Tanenbaum, *Modern Operating Systems,* Prentice-Hall, Englewood Cliffs, NJ, 1992

[8] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner, *Linux Kernel Internals*, Addison Wesley Longman, 1996.

[9] WinZip, The Archive Utility for Windows, version 8.1, 2002. http://www.winzip.com/winzip.htm

[10] Eighteenth IEEE Symposium on Mass Storage Systems, San Diego, California, April 2001. http://esdis-it.gsfc.nasa.gov/MSST/