Extending the Native File System Functionality for Data Compression Using Vnode Stacking

Mandar Karyakarte*, Suhas Patil**, R K Prasad***

Abstract

File-handling services are the most user-visible part of the operating system and so new enhancements are often proposed to the existing file-handling services. But very few enhancements become widely available as developing file systems from scratch are difficult and error prone, source access is required, and the work involves deep understanding of that operating system's internals. Once the work is accomplished on one platform, a port to another is almost as difficult as the initial port. A Layered approach based on vnode stacking is easy and feasible technique to extend the file system functionality. The native file system can be enhanced to provide service like fan-out, anti-virus, access control, encryption, compression etc. The paper discusses the Compression File System (CFS)) based on vnode stacking which will reside below the virtual file system (VFS) and above the native file system (Ext2, Ext3 etc.). The data from user is saved to the disk after compressing it and is retrieved by the user after decompressing it. The compression ratio achieved varies in the range of 10 – 70 % based on the type of data compressed. The file system is efficient as it executes in the kernel and easy to use as it can be mounted on top of native file system.

Index Terms: Virtual file system, vnode interface, FIST, mounting.

* Department of Information Technology, Vishwakarma Institute of Information Technology, Pune karyakarte.ms@gmail.com,

** Department of Computer and IT, Bharati Vidyapeeth University College of Engineering, Pune.

*** K J Somaiya Institute Of Engineering And Information Technology, Sion, Mumbai

1. Introduction

The basic task of any operating system is to provide data management functionality. Managing data through the file system includes storing data on disk (or over the network) and naming (i.e., translating a user-visible name such as /root/mydata into an on-disk object). File systems are complex, and enhancing them involves understanding the file system code and internals of the kernel. Furthermore, operating system developers and vendors are reluctant to make major changes to a file system, because file system bugs have the potential to corrupt all the data on a machine. Because file system development is so difficult, extending file system functionality in an incremental manner is valuable. Incremental development also makes it possible for a third-party software developer to release file system improvements, without developing a whole file system from scratch.

The file systems initially were tightly coupled with the core of the operating system, system calls directly invoked the file system functions. This arrangement made the addition of new facilities more difficult. Additional facilities could be incorporated only by developing the new file system from scratch. The introduction of a virtual node or vnode provided a layer of abstraction that separates the core of the operating system from file systems [9]. Each file is represented in memory by a vnode. A vnode has an operations vector that defines several operations that the OS can call, thereby allowing the operating system to add and remove types of file systems at runtime. Most current operating systems use something similar to the vnode interface, and the number of file systems supported by the OS has grown accordingly. For example, Linux 2.6 supports over 30 file systems and many more are maintained outside of the official kernel tree.[11]

The vnode stacking produces a file system that can be 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 [2] and all the new features can be included in the stackable layer. All the invoked system calls [3] will pass through the new file system 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 fan-out, anti-virus, access control, encryption, compression and many more. The position of the new stackable file system is exhibited in the figure 1.

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 topology of our file system stack.

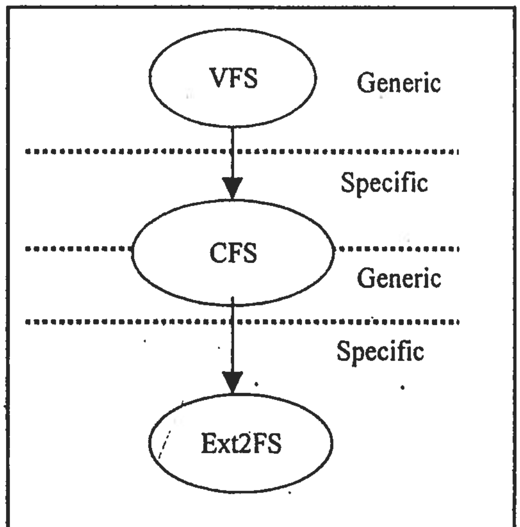

Figure 1. Position of CFS in kernel

The VFS (Virtual File System) [1], [5] 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 [7]. As our file system framework is stackable it only has to implement the vnode [7] 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 super class [2].

The stacked file system handles many of the internal details of operating systems, thus freeing developers from dealing with kernel specifics and provides them with a file system, which has desired facilities enhanced.

2. Terminology

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

2.1 Virtual File System

The Virtual File System is the software layer in the kernel that provides the file system interface to user-space programs [1]. 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 [4].

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 [2], [7].

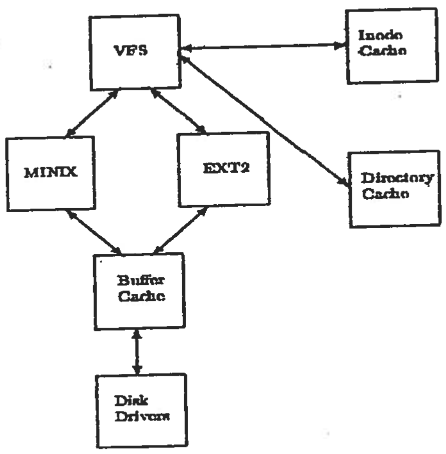

Figure. 2 Vnode Block Diagram

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 [7]. 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.

2.2 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 [6]. 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 [2] 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. Design and implementation

To implement vnode stacking and thus enhance the native file system, a compression file is developed. The compression file system (CFS) generates a lot of code using the File System Translator i.e FiST [2]. The FiST language is the first of the three main components of the FiST system. The FiST system is composed of three parts:

1. The language specification,

2. The code generator, and

3. Stackable file system templates.

The system is implemented using following brief steps

1. The compression functionality to be added requires selection of algorithm that will work irrespective of nature of the data and the file type.
2. After selecting and implementing the compression algorithm it must be tested independently to understand the compression ratio.
3. Write the code in fist language in the file name as cfs.fist. the code is simple which calls the entry point of the compression algorithm.
4. Compile the cfs.fist using the FIST compiler that will compile the all the files of compression algorithm to *.ko.
5. Run insmod command to insert the kernel-output files and also define the mount point which is addressed below
6. The fist compiler ensures of registering the file system in the /etc/filesystems.

The selection of the compression algorithm was a tedious job as after deploying the developed file system it should not put much overhead on the existing read/ write calls. Secondly, as file system is a generic system-level application it needs to work with all kind of the data and can't be data specific.

The developed file system can be easily mounted on any name space or path of the native file system that the user wants. The user prior to mounting the filesystem on top of the native file system should create the CFS_folder. After successful mounting [3], the folder will have type of file system as CFS. All file system operations performed under the CFS_folder will henceforth follow the specifications of the CFS file system framework.

When you unmount this file system, the CFS will no longer serve the CFS_folder. Hence any access to files under the CFS_folder will result in the user reading compressed data.

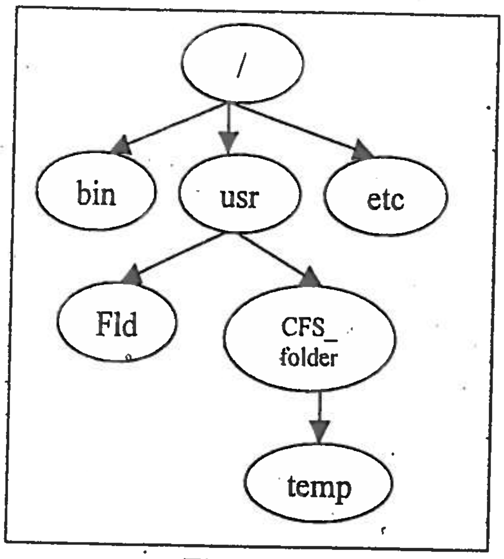

Figure. 3
Directory Structure before mounting

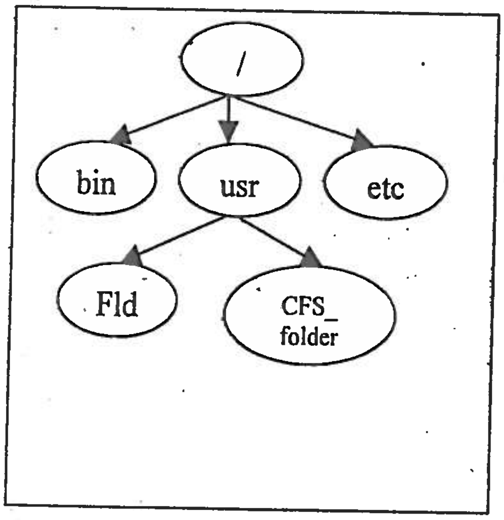

Figure.4
Directory Structure after mounting

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 CFS, as they would normally have. As the CFS is in the kernel, the overhead incurred is just 1-2% [2].

Figure 5 shows how the CFS will process a read system call. As shown in the figure and with respect to the command given above, when a system call, which processes a request to read data on the disc, then it, passes through the CFS 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. Firstly LZ77/ RLE decompression gets called and finally Huffman decompression is called [10].

Similarly when a write system is encountered then the reverse process occurs. In this case, our layer calls the functions of Huffman Compression, LZ77/ RLE compression in order.

As we can see from the figure above, the interface we 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 CFS 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. Efficient storage management is achieved, improving system performance besides this developer can provide new algorithms for compression with minimum effort thus building new file systems.

Shrinking the size of demands for efficient mapping between the original file and new compressed file so as to have lossless data compression. The strategy used is based on

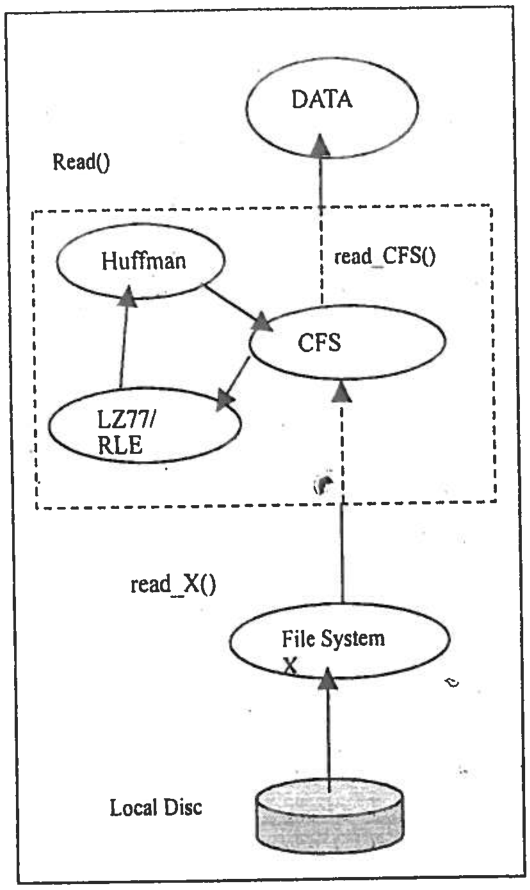

Figure 5.
Read system call illustration

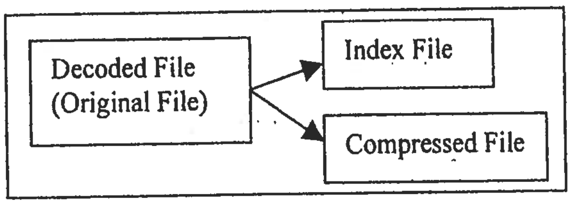

Figure 6 Original File and Compressed File with generated index file

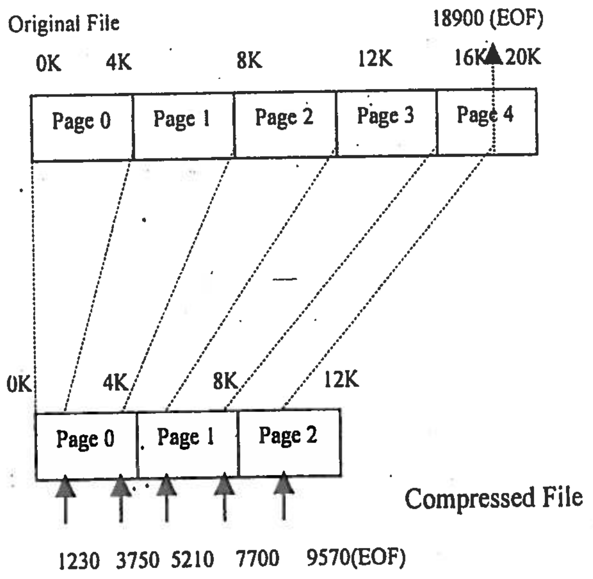

Figure 7 Illustration of original file and compressed file

index file. The index file is a separate file containing Meta data that serves as a fast index into an encoded file the index file stores meta-data information identifying offsets in an associated encoded file. For efficiency reasons, we can read the generally smaller index file quickly, and hence find out the exact offsets in the larger data file to read from. The alternative to our design would have been to include the index data into the main encoded file, but this would have

1. Hurt performance as we would have had to perform many seeks in the file to locate the data needed.
2. Complicated the rest of the design and implementation considerably.

The figure 6 shows how original decoded file is mapped into an encoded file of smaller size. The contents of the index file are tabulated below

Sr. No.	Word (32 bit)	Representing	IDX File
1	0-31	Flags and # of pages	4
2	32-63	Original File size	18900 bytes
3	64-95	Page 00	1230
4		Page 01	3750
5		Page 02	5210
6		Page 03	7700
7		Page 04	9570

Table 1 shows the contents of the index file

little or no modification (e.g., text editors). In contrast, many applications now must implement their own security and data compression features.

Moreover as code runs at kernel level harder to crack where user-level applications are susceptible to attacks. New user applications also gain from such system security, rather than have it become a design afterthought.

6. Conclusion

The basic aim of working on the CFS was to design a file system, 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 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. 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 requires writing 2 simple functions which will provide the ability for compression and decompression facility. Improvement when compared to the Ext2 file system also results in performance overhead. As all system calls have to pass through the CFS 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.

The CFS automatically compresses the files stored in its name-space. 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.

The vnode stacking can be used to extend and enhance the file system functionality by introducing a layer(s) between the disk file system and virtual file system. As the mounted file system is below VFS it remains to be efficient as it executes in the kernel space.

References

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

[2] E. Zadok, " FiST: A System for Stackable File-System Code Generation," Thesis Computer Science Department Columbia University, New York, NY10027, May 2001.

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

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

[5] R. Gooch, " Overview of The Virtual File System," July 1999.

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

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

[8] WinZip, The Archive Utility for Windows, version 8.1, 2002.

[9] . R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the Summer USENIX Technical Conference,* pages 238-247, Atlanta, GA, June 1986. USENIX Association.

[10] RFC 1951 DEFLATE Compressed Data Format Specification ver 1_3

[11] Erez Zadok, Rakesh Iyer, Nikolai Joukov, Gopalan Sivathanu, And Charles P. Wright. "On incremental file system development" appears in the may 2006 issue of the acm transactions on storage (tos)