Open Source Code Doesn't Always Help: Case of File System Development

Wasim Ahmad Bhat [*]
S.M.K. Quadri [**]

Abstract

Purpose: *One of the most significant and attractive features of Open Source Software (OSS), other than its cost, is its open source code. It is available in both flavours; system and application. It can be customized and ported as per the requirements of the end user. As most of the system software run in the kernel mode of operating system and system programmers constitute a small chunk of the programmers, the code customization of Open Source System Software is less realized practically. In this paper, the authors present file system development as a case of Kernel Mode System Software development and argue that customization of Open Source Code available for file systems is not preferred. To support the argument, the authors discuss various challenges that a developer faces in this process. Furthermore, the authors look into the user mode file system development for possible solution and discuss the architecture, advantages and limitations of most popular and widely used framework called File system in User-Space (**FUSE)**. Finally, the authors conclude that the user mode alternative for file system development and/or extension supersedes kernel mode development.*

Design/Methodology/Approach: *The broad domain, complexity, irregularity and limitations of kernel development environment are made as a base to put forth our argument. Moreover, the existence of rich and capable user-mode file system development frameworks are used to supplement the argument.*

Findings: *The research highlights the fact that kernel mode file system development is difficult, bug prone, time consuming, exhaustive and so on, even with source code at disposal. Furthermore, it highlights the existence of user mode alternative which is easy, reliable, portable, etc.*

Research Implications: *The research considers file system development as a case of kernel mode development. Fortunately, in this case, the authors have choice of user mode alternatives. However, author argument cannot be generalised for those kernel modules wherein there is no user mode alternative. Furthermore, the authors did not take into consideration the benefits of extending file systems in kernel mode.*

Originality/Value: *The research stresses that having open source code is not enough to make a choice when we cannot use it in a reliable and productive manner.*

Keywords*: Open Source Software); Open Source System Software, Source Code, File System, Kernel Mode, User Mode, File system in User-Space (FUSE)*

Paper Type*: Argumentative*

[*] Ph.D. Scholar. P. G. Department of Computer Sciences, University of Kashmir, Jammu and Kashmir. 190 006. India. email: wasim.ahmed.bhat@gmail.com

[**] Head. P. G. Department of Computer Sciences, University of Kashmir, Jammu and Kashmir. 190 006. India. email: quadrismk@hotmail.com

Introduction

Open Source Software is consistently gaining on its software market share because of its two most notable strengths which include low cost and availability of source code. Although, for some low cost is enough to make a choice while for others availability of source code is mandatory. Having source code at our disposal, the product can be customized or optimized as per the requirements or can be used to fix unanticipated bugs. Open source ideology is logically simple; one creates OSS project, uploads the project along with source code and license to download, customize, distribute, compile and use it. There are many portals that host OSS projects; http://sourceforge.net being the most popular one. OSS paradigm started unknowingly early in 1960's when RFCs for network protocols were created by ARPANET, followed by a big boost by Linus Torvalds' Linux OS. The paradigm has spread geographically because of Internet and has penetrated into every aspect of software development, be it an application software or system software. This penetration is largely because of Linux Operating System which is the one of the most prominent example of OSS and provides an excellent platform to develop such software. OSS has attracted Computer Science researchers all over the globe because of the availability of source code that too just couple of clicks away. Specifically, researchers working on the system side aspects of Computer Science have been using Linux OS to implement and test their ideas and innovations by customizing the source code and recompiling it.

One of the most notable system side research areas that specifically depend upon the availability of source code is File System Development (FSD). FSD includes designing and developing a new file system from scratch and/or extending the existing ones in order to accommodate and cope up with change in hardware technology and user requirements. Designing and developing a file system from the scratch is practically rarely practiced. There are many reasons for this; a significant innovation is required in new design, a number of good designs are already available and implemented a lot of knowledge about operating systems internals and experience with system programming is required for development. But, because the hardware technology is both getting advanced and affordable, the digital data proliferation rate is very high. This has created voluminous amount of digital data which needs to be managed efficiently, reliably and securely. This change in hardware technology and user requirements asks for optimization, refinement and fine tuning of existing file systems.

Linux, the Open Source System Software, provides a good platform for testing and implementing such refinements. Linux is a pioneer and prominent software in OSS community. The Linux kernel comes with

more than two dozen file systems along with source code. These in-kernel file systems are difficult to develop and debug. In this paper, authors argue that code customisation of in-kernel file systems, to extend their capabilities, is not preferred even with open source code. To support the argument authors discuss various challenges that are faced by a developer in this process. Furthermore, authors navigate to file system development in user space to look for possible solution. The authors present an overview of various user space frameworks and discuss the architecture, advantages and disadvantages of most popular and widely used framework called ***FUSE***. This existence of rich and capable user space framework supplement author argument.

Why in-kernel code customization of File System is not preferred?

File systems represent one of the most important aspects of operating-system services. Traditionally, file systems are integrated with the operating system kernel. Earlier, file system *syscals* directly invoked file system methods. This architecture made it difficult to add multiple file systems to an OS. In 1986, to address this problem, **Kleiman (1986)** introduced virtual node or *vnode* which provides a layer of abstraction that separates core OS from file systems. This architecture finally matured into VFS in UNIX like and UNIX based OSes. **Rosenthal (1992)** proposed layering to extend capabilities of file systems and modified VFS of SunOS to support it. All these demarcations and modifications remained within the boundary and domain of the kernel.

As mentioned earlier, file systems need to evolve. Customizing in-kernel file systems is a challenging task because of variety of reasons. First, this approach requires the programmer to understand and deal with complicated kernel code and data structures. Thus, a deep understanding of operating system (kernel) internals is required even to make a small change in existing code or to add some new code. The situation is worse than it seems as the operating systems vary in their kernel architectures, same architectures vary in major aspects for different flavours, same flavours vary in crucial implementations for different versions and same versions vary in degree of cohesion for different underlying hardware. All these factors finally lead to a time consuming and exhaustive effort of a programmer to understand internals of a specific kernel release. Furthermore, such programmers constitute a small chunk of programmers.

Second, even if this is all what is required and is successfully done, the code customization can induce more bugs than expected. The kernel development environment lacks facilities that are available to application programmers. For instance, the kernel code lacks memory protection as it runs in supervisor mode of operating system and as such a single wild

pointer can bring down the system which otherwise could have only terminated the application. Also, it requires careful use of synchronization primitives, can only be written in C and that too without being linked against the standard C library and so on. All these factors lead to a higher probability of not only a simple bug induction but a bug that is capable enough to bring down the system and hence affects the reliability of the operating system.

Third, if the customization is inevitable then debugging not only is obvious but tedious. Debugging kernel code is much difficult than debugging user space code as kernel code development lacks facilities found in IDE for most programming languages for user mode development. For instance, the famous "Blue Screen of Death" on Windows platform is still there since the inception of Windows.

Fourth, even a fully functional in-kernel file system still has several disadvantages. Porting a file system written for a particular kernel to a different one can require significant changes in the design and implementation, though the use of similar file system interfaces (such as the VFS layer) on several Unix-like systems makes the task somewhat easier.

Finally, an in-kernel file system can be mounted only with *super user* privileges. This can be a hindrance for file system development and usage on centrally administered machines, such as those in universities and corporations.

How File Systems can be extended in User Space?

In contrast to kernel development, programming in user space minimizes or completely eliminates several of the aforementioned issues. By developing and/or extending file systems in user space, the programmer need not to worry about the intricacies and challenges of kernel-level programming and has access to a wide range of familiar programming languages, third-party tools and libraries. Further, a highly dangerous bug can at most terminate the application and hence can never break the reliability of kernel. Moreover, debugging is comparatively much easier. Of course, user space file systems may still require some effort to be ported to different operating systems. However, this depends on the extent to which a file system's implementation is coupled with a particular operating system's internals.

In order to develop and/or extend file systems in user space, a framework is required which traps the file system calls in kernel and passes them to user space to be processed. The framework should also provide a simple and powerful set of APIs in user space that are common amongst most operating systems. Various projects have aimed to support development of user space file systems while exporting an API similar to that of the VFS

layer. A brief introduction of such popular frameworks is as follows.
UserFS consists of a kernel module that registers a UserFS file system type with the VFS **(Fitzhardinge, n.d)**. All requests to this file system are then communicated to a user space library through a file descriptor. The Coda distributed file system contains a Coda kernel module which communicates with user space cache manager, *Venus,* through a character device /dev/cfs0 **(Satyanarayanan, Kistler, Kumar, Okasaki, Siegel & Steere, 1990)**. UserVFS, which was developed as a replacement for UserFS, uses this Coda character device for communication between the kernel module and the user space library **(Machek, n.d)**. Similarly, *Arla* is an AFS client that consists of a kernel module, *xfs*, which communicates with the *arlad* user space daemon to serve file system requests **(Westerlund & Danielsson, 1998)**.
The *ptrace ()* system call can also be used to build an infrastructure for developing file systems in user space **(Spillane, Wright, Sivathanu & Zadok, 2007)**. An advantage of this technique is that all OS entry points, rather than just file system operations, can be intercepted. The downside is that the overhead of using *ptrace()* is significant, which makes this approach unsuitable for production level file systems.
The number of production-quality systems that provide a standardized API for developers to design a unique file system in user space is still small, but there is one commonly used and well deployed system called FUSE; part of the Linux kernel since version 2.6.14.

FUSE: A Widely Used Framework for File Systems in User Space
The fundamental design consideration of microkernel implementations such as Mach and the MIT exo-kernel is to reduce the complexity of the kernel. Both approaches remove all but the most basic operating system services from the kernel, moving it to programs residing in userspace. FUSE (File system in User-Space) is a recent example of this general trend in operating system design **(Szeredi, n.d)**. FUSE is the most well-known example of a user space file system framework. FUSE design provides a thin layer in kernel which traps and forwards file system calls meant for mounted FUSE file system to user space. In user space, FUSE provides a library interface to implement the corresponding file system call functionality.

Architecture of FUSE
FUSE is a three-part system (shown as shaded blocks in **Fig. 1**).The first of those parts is a kernel module, *FUSE*, which hooks into the VFS code and looks like a file system module. It registers *fusefs* file system type with VFS and also implements a special-purpose device */dev/fuse*. In user space, FUSE implements a library, *libfuse*, which manages

communications with the kernel module. It accepts file system requests from the FUSE device and translates them into a set of function calls which look similar (but not identical) to the kernel's VFS interface. Finally, there is a user-supplied component (*userfs* in our example in **Fig. 1**) which actually implements the file system of interest. It fills a structure with pointers to its functions which implement the required operations in whatever way makes sense.

Fig. 1: Path of a read () call for a file residing in FUSE file system

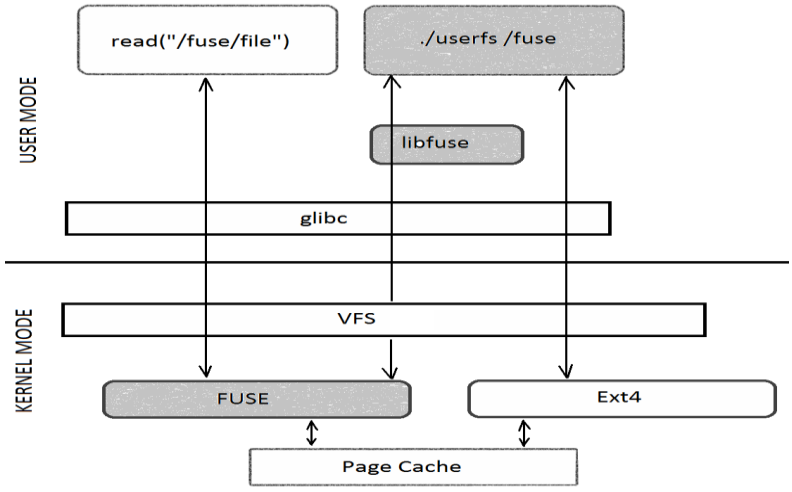

Fig. 1 shows the path of a *read()*call for a file residing in FUSE file system. In this example, the user space file system functionality is implemented as a set of *callback* functions in the *userfs* program, which is passed the mount point, */fuse*. Once *userfs* FUSE file system is mounted, all file system calls targeting the mount point, */fuse*, are forwarded to the FUSE kernel module. When an application issues a *read()* system call for the file */fuse/file*, the VFS invokes the appropriate handler in *fusefs*. If the requested data is found in the page cache, it is returned immediately. Otherwise, the system call is forwarded over a character device, */dev/fuse*, to the *libfuse* library, which in turn invokes the *callback* defined in *userfs* for the *read()* operation. The *callback* may take any action, and return the desired data in the supplied buffer. For instance, it may do some pre-processing; request the data from the underlying file system (such as Ext4 in our example) and then post-process the read data **(Mathur, Cao, Bhattacharya, Dilger, Tomas & Vivier, 2007)**. Finally, the result is propagated back by *libfuse*, through the kernel to the application that issued the *read()* system call.

Advantages of using FUSE framework

FUSE's relatively loose policy of implementing file system APIs allows developers to run file systems with only a few functions implemented. Also, FUSE presents an application with a well-known, standardized and native file system that accepts regular system calls. This means that applications can use interesting and cutting edge file systems on FUSE without changing any code inside the application. This easy prototyping and application friendliness of FUSE's design clearly encourages not only file system developers but also other people who are not familiar with kernel programming to challenge themselves by implementing their own file systems.

Developers implementing a file system in user space no longer have to recompile the kernel or worry about crashing the operating system during development. FUSE moves a step further by allowing unprivileged users to safely mount their own file systems, even ones they make themselves, as long as the system administrator loads the FUSE kernel module.

More than twenty different language bindings are available for FUSE, allowing file systems to be written in languages other than C. This means that programmers can use languages that are based on different programming paradigms, offer different levels of type safety and type checking, and are generally intended for different usage scenarios.

The main advantage of FUSE over other similar projects is its large and active user community. The FUSE user community has developed several dozen file systems to date, several of which provide significant functionality to the platforms supported by FUSE. Among the more interesting FUSE file systems are Wayback **(Cornell, Dinda & Bustamante, 2004)**, NTFS-3g **(NTFS-3g, n.d)** and SSHFS **(Szeredi, n.d)**. These provide a versioning file system, safe read and write support for NTFS volumes, and a file system based on secure communications over SFTP, respectively.

Furthermore, FUSE framework has been ported to almost all platforms including Windows **(Driscoll, Beavers & Tokuda, n.d)**. Thus, FUSE file systems not only are reliable and easy to develop and debug, but are also highly portable.

Performance Issues in FUSE File Systems

There are certain performance issues related to FUSE framework's architecture **(Rajgarhia & Gehani, 2010)**. First, when only anin-kernel file system (such as Ext4 alone) is used, there are two user-kernel mode switches per file system operation (i.e. to and from the kernel), and no process context switches. User-kernel mode switches are inexpensive and involve only switching the processor from unprivileged user mode to

privileged kernel mode, or vice versa. However, FUSE introduces two process context switches for each file system call. There is a process context switch from the user application that issued the system call to the FUSE user space library, and another one in the opposite direction. A context switch can have a significant cost, although the cost may depend vastly on a variety of factors such as the processor type, workload, and memory access patterns of the applications between which the context switch is performed.

Second, while using an in-kernel file system alone, data need to be copied in memory only once, either from the kernel's page cache to the application issuing the system call, or vice versa. FUSE also introduces two additional memory copies. While writing data to a FUSE file system, the data is first copied from the application to the page cache, then from the page cache to *libfuse* via /dev/fuse, and finally from *libfuse* to the page cache when the system call is made to the in-kernel file system. For *read()*, the copying is similarly performed in opposite direction. If the FUSE file system is mounted with the DIRECT_IO option, then the FUSE kernel module bypasses the page cache and forwards the application-supplied buffer directly to the user space daemon. In this case, only one additional memory copy is performed. The advantage of using DIRECT_IO is that writes are significantly faster due to the reduced memory copy. The downside is that each *read()* request has to be forwarded to the user space file system, as data is not present in the page cache, thus affecting read performance severely.

Finally, when using the in-kernel file system alone, all data that is read from or written to the disk is cached in the kernel's page cache. With FUSE, *fusefs* also caches the data in the page cache, resulting in two copies of the same data being cached. Although the use of the page cache by FUSE is very beneficial for read operations since it avoids unnecessary context switches and memory copying, the fact that the same data is cached twice reduces the efficiency of the page cache. In Linux, one can open files on the native file system using the O_DIRECT flag and thereby eliminate caching by the native file system. However, this is generally not a feasible solution since O_DIRECT imposes alignment restrictions on the length and address of the *write()* buffers and on the file offsets.

Conclusion

In this paper the authors argued that the in-kernel code customization of open source system software like file system, is not practically feasible as it requires deep understanding of operating system internals, experience with kernel level programming, is time consuming and exhaustive, and so on. Furthermore, the process is highly prone to simple bugs which can

crash operating system, and demand studious and exhaustive debugging. All these factors lead to slow progress in file system development with higher probability of low operating system reliability and file system productivity; all this with even source code at the disposal.

The research also highlighted the concept of file system development (extension) in user space and explained the basic architecture of most popular user space file system development framework called *FUSE*. Although, there are certain performance issues related to FUSE but the gains outcast the issues. It can be safely argued that a file system extended using FUSE framework is very easy to develop and debug, in addition to being highly reliable and portable as compared to the one which is extended by customising and recompiling the source. This is enough to validate the argument that almost all user-space file system frameworks are from open source community; surfaced to overcome the code customisation problem in one of their pioneer and flagship product, Linux OS.

References

Cornell, B., Dinda, P., and Bustamante, F. (2004). *Wayback: A User-level Versioning File System for Linux*. *In Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '04),* Article 27.

Driscoll, E., Beavers, J., & Tokuda, H. (n.d). *FUSE-NT: Userspace File Systems for Windows NT*. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.131.3896

Fitzhardinge, J. (n.d). *UserFS.* Retrieved from http://www.goop.org/~jeremy/userfs

Kleiman, S. R. (1986). Vnodes: An architecture for multiple File system types in Sun UNIX. *In Proceedings of the Summer USENIX Technical Conference,* pp. 238-247.

Machek, P. (nd). *UserVFS.* Retrieved from http://sourceforge.net/projects/uservfs

Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A., & Vivier, L. (2007). The new ext4 filesystem: Current status and future plans. *In Proceedings of the Ottowa Linux Symposium*.

NTFS-3g. (n.d). Retrieved from http://www.tuxera.com/community/ntfs-3g-manual/

Rajgarhia, A., & Gehani, A. (2010). Performance and extension of user space file systems. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC '10)*, pp. 206-213.

Rosenthal, D. S. H. (1992). *Requirements for a "Stacking" Vnode/VFS interface.* Tech. Rep. SD-01-02-N014, UNIX International.

Satyanarayanan, M., Kistler, J. J., Kumar, P., Okasaki,M. E., Siegel, E. H., & Steere, D. C. (1990). Coda: A highly available file system for a distributed workstation environment. *IEEE Transaction on Computers*, 39(4), 447-459.

Spillane, R. P., Wright, C. P., Sivathanu, G., and Zadok, E. Rapid file system development using ptrace. *In Proceedings of the 2007 workshop on Experimental computer science (ExpCS '07)*, ACM, Article 22.

Szeredi, M. (n.d). *File system in user space.* Retrieved from http://fuse.sourceforge.net

Szeredi, M. (n.d). SSH filesystem. Retrieved from http://fuse.sourceforge.net/sshfs.html

Westerlund, A., & Danielsson, J. (1998). Arla- a free AFS client. *In Proceedings of the annual conference on USENIX Annual Technical Conference(ATEC '98)*, Article 32.