Fault Isolation and Quick Recovery in Isolation File Systems

Lanyue Lu, Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau
Computer Sciences Department, University of Wisconsin, Madison

Abstract

File systems do not properly isolate faults that occur within them. As a result, a single fault may affect multiple clients adversely, making the entire file system unavailable. We introduce a new file system abstraction, called *file pod*, to allow applications to manage failure and recovery polices explicitly for a group of files. Based on this abstraction, we propose the isolation file system, which provides fine-grained fault isolation and quick recovery.

1 Introduction

High availability is critical for file systems. For desktops and laptops, local file systems directly affect data access for the user; for mobile devices [2, 9], user data is also stored in a local file system; for file and storage servers, a shared cluster file system may be used to store virtual machine disks from multiple clients [13, 3].

File systems must handle a wide range of faults [15], including resource allocation failures, metadata corruption, failed I/O operations, and incorrect system state. These faults are caused by both hardware defects [4] and software bugs [8].

Unfortunately, the effect of a single fault can have a large-scale impact on the operation of the entire file system. Such *global* failures are prevalent in file systems. For example, when Ext3 detects a corruption in the data block bitmap of a block group, it will re-mount the whole file system as read-only or call `panic()` to crash the operating system. There are also numerous assertions (e.g., `Assert`, `BUG_ON`) in file system code, which will crash the file system when only a small piece of system state is not as expected.

Global failures severely harm the availability of file systems in various scenarios. For example, in server virtualization environments, multiple virtual machines share the same underlying host file system; a fault that arises within a single VM image file may lead to a crash or read-only remount, and thus affect all running VMs. Isolation, a key property of virtualized systems, is not preserved.

To prevent global failures, we propose isolation file systems, which provide fine-grained fault isolation and quick recovery. Isolation file systems have the following major characteristics. First, they are *thoroughly partitioned*: file system resources are broken down into many independent units. Second, they are *independent*: any individual fault within defective units will not affect other healthy units. Finally, they are *resilient*: faulty units can be identified and repaired quickly.

In this paper, we begin by analyzing the failure causes and global failure policies of existing file systems. Motivated by this data, we propose a new file system abstraction known as a *file pod* which allows applications to manage failure policies and recovery polices for their data. We then briefly sketch an implementation of isolation file system based on the existing Ext3 file system.

2 Failure Policies in File Systems

Before describing the isolation approach, we first analyze existing file systems and their reaction to various faults. This section presents our initial results.

2.1 Global Fault-Handling Policies

Global fault-handling policies are used to react to serious errors within a file system. Such serious errors include metadata corruption, I/O failures, and incorrect system states caused by software bugs. We focus on three major file systems in this section: Ext3 (Linux 2.6.32), Ext4 (Linux 2.6.32) and Btrfs (Linux 3.8).

From our analysis of the code, we have found that there are two types of global reactions in modern file systems: *remount read-only* and *crash*. For example, when Ext3 detects a block bitmap is corrupted, it may remount the whole file system as read-only to prevent further corruption. In contrast, the Ext3 journaling layer, JBD, may trigger a `BUG_ON` statement to crash when it finds the journal in an unexpected state.

We analyze the file system code to identify the error handling functions that cause these reactions; for example, Btrfs calls `btrfs_handle_error()` to force the file system read-only. Then, we count how many times these basic error functions are called in different places. Note that we also count wrapper functions which directly call these basic error handling functions.

Figure 1 shows the total number and breakdown of global failure types in Ext3, Ext4, and Btrfs. We find that global failure reactions are common in both young and mature file systems. Over two thirds of these global behaviors will directly crash the whole file system, greatly reducing availability.

2.2 Global Failure Causes

To understand if these global failures are warranted, we identify the root cause of each global failure statement in

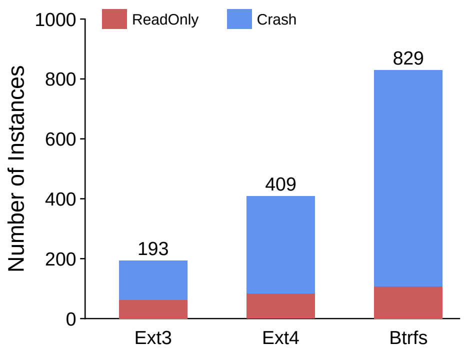

Figure 1: **Failure Types.** This figure shows the failure types for each file system. The total number of global failure instances is on top of each bar.

Data Structure	MC	IOF	SB	Shared
b-bitmap	2	2		Yes
i-bitmap	1	1		Yes
inode	1	2	2	Yes
super	1			Yes
dir-entry	4	4	3	Yes
gdt	3		2	Yes
indir-blk	1	1		**No**
xattr	5	2	1	**No**
block			5	Yes/No
journal		3	27	Yes
journal_head			31	Yes
buf_head			16	Yes
handle		22	9	Yes
transaction			28	Yes
revoke			2	Yes
other	1		11	Yes/No
Total	19	37	137	= 193

Table 1: **Global Failure Causes of Ext3.** This table shows the failure causes for Ext3, in terms of data structures, failure causes and their related numbers. **MC**: Metadata Corruption; **IOF**: I/O Failures; **SB**: Software Bugs; **Share**: whether this structure is shared by multiple files or directories.

each file system. We have found that there are three major root causes for each failure case: metadata corruption, I/O failure, and software bugs.

Table 1 shows our analysis for the Ext3 file system. Specifically, the table shows the interplay between each major data structure of the file system (e.g., bitmaps, inodes, superblock, directory entries) and the root cause of a global failure involving that data structure.

Ext3 explicitly validates the integrity of metadata in many places, especially at the I/O boundary when reading from disk. For example, Ext3 validates a directory entry before traversing that directory and Ext3 checks that the inode bitmap is in a correct state before allocating a new inode. Unfortunately, as indicated by the Metadata Corruption column, if Ext3 detects a corruption in any of these structures, it causes a global failure. The I/O Failure column similarly shows that Ext3 causes global failures when an individual I/O request fails. Finally, the Software Bugs column shows that there are a significant number of internal assertions (such as `BUG_ON`), which are utilized to validate file system state at runtime, and these also cause a global failure when invoked. We observe that all of the global failures in Ext3 are due to problems with metadata and other file system internal state, and not user data.

For each data structure, we also check whether it is shared across different files. As shown in the last column of Table 1, most metadata structures are organized in a shared manner and thus can cause global failures. However, even for local structures, such as indirect blocks, a global failure can still occur.

2.3 Discussion

A file is the basic file system abstraction used to store the user's data in a logically isolated unit. Users can read from and write to a file. Another basic abstraction is a directory, which maps a file name to the file itself. Files and directories are usually organized as a directory tree.

A namespace holds a logical group of files or directories. To protect files in a shared environment, different applications are isolated within separated namespaces. Typical examples include `chroot`, BSD `jail`, Solaris Zones, and virtual machines.

However, these abstractions do not provide any fault isolation within a file system. Files and directories only represent and isolate data logically for applications. Within a file system, different files and directories share metadata and system state; when faults are related to these shared metadata, global failure policies are triggered.

One might think using multiple physical partitions to separate file systems would provide equivalent fault tolerance and protection to a file pod. For example, corrupted data are isolated to a single partition. However, a single `panic()` on one file system may crash the whole operating system, affecting all partitions. Furthermore, static physical partitions are not elastic; thus the storage space is not efficiently utilized.

Therefore, file system abstractions lack a fine-grained fault isolation mechanism. Current file systems implicitly use a single fault domain; a fault in one file may cause a global reaction, thus affecting all clients of the file system.

3 New Abstraction: File Pod

To address the problem of inadequate fault isolation in file systems, we propose a new abstraction, called a *file pod*, for fine-grained fault isolation in file systems.

A file pod is an abstract file system partition that contains a group of files and their related metadata. Each file pod is isolated as an independent fault domain within the file system, with its own failure policy. Any failure related to a file pod only affects itself, not the whole file system. For example, if metadata corruption is detected within a file pod and the failure policy is to remount as read-only, then an isolation file system marks only that pod as read-only, without affecting other consistent file pods.

File pods allow applications to control the failure policy of their own files and related metadata, instead of letting the file system manage the failures globally. Furthermore, this new abstraction supports flexible bindings between namespaces and fault domains; thus it can be used in a wide range of environments, such as server virtualization (a primary target of ours), security isolation, and personal computer scenarios.

3.1 Operations on File Pods

The file pod abstraction supports following operations.

Create a file pod: An application can create a file pod when needed. A file pod has a unique ID and attributes. A default global file pod is assigned when creating a new file system using `mkfs`.

File pod's attributes: Each file pod has attributes in addition to its ID. An application can get and set a file pod's attributes. Attributes can include: failure policies (e.g., read-only, pure crash, on-going accesses are allowed but new accesses are rejected), file characteristics hints (e.g., large virtual disk files, small configuration files), and recovery policies (e.g., online `fsck`, offline `fsck`).

Set a file's pod: An application can assign a file pod for a file or a directory. If the file or directory has a file pod previously, then its file pod will be changed to this new file pod. For a directory, the file pod is inherited by default for all files and directories created later under this directory.

Remove a file's pod: An application can remove a file pod for a file or a directory. If the file or directory only has one file pod previously, then its file pod will be changed to the global file pod.

Share a file between pods: An application can share a file or a directory between several different file pods. A special API is provided for applications to add a file to other file pods in addition to the file's own file pod. If faults are related to a shared file or directory, then different failure policies will be triggered for different file pods containing the faulty file or directory. Thus, applications should be aware of all pods for their files and corresponding failure policies.

3.2 Typical Usage Cases

We envision a number of typical usage scenarios for pods.

Server virtualization environment. Each virtual machine has its own file pod with its virtual disk files and configuration files. The failure policy can be set specifically per pod, and thus true isolation is enforced across VM domains. Once failures happen, `fsck` can be run immediately to recovery corrupted files.

Running untrusted applications. Each untrusted application runs within a separate file pod with its files. The failure policy of this file pod can be set as killing all the related processes and removing the file pod namespace.

Intermediate data. Big data applications may generate a large amount of intermediate data. A useful failure policy is marking the file pod as erroneous to prevent new processes from accessing it, but to allow running processes to finish. After that, we can run `fsck` or applications can check their data integrity directly.

4 Fault Isolation

This section describes how an isolation file system could provide fine-grained fault isolation for file systems. The goal of fault isolation is to allow each file pod to handle its own failures. Our solution consists of two components: metadata isolation and local failure polices.

4.1 Metadata Isolation

Modern file systems manage metadata in a shared manner. For example, an inode block may contain multiple inodes. A single failure that occurs with an inode block may impact multiple files. We argue that shared metadata organization is harmful for fine-grained fault isolation.

Our idea is to isolate metadata for each file pod. As we described before, a file pod contains all its files and related metadata. Because we organize each file pod's metadata independently without any sharing, then any metadata related failure can be narrowed down to a specific file pod.

4.2 Localize Failures

As we showed earlier, current file systems handle serious failures by remounting as read-only or crashing the whole system. These global actions need to be changed to localize failures within erroneous file pods. Our goal is not to change the failure polices in file systems, but adapt them to local file pods.

How can we handle a read-only remount locally? If a file pod's failure policy is remounting as read-only, then we simply mark the file pod as read-only instead of marking the whole file system as read-only. We need to prevent further updates for both file system structures and the on-disk journal for a faulty pod. In this manner, only files within this file pod will be affected, while other consistent files are still available for normal accesses.

How can we handle a pure crash locally? If a file pod's failure policy is to crash, then we need to provide the same states and behaviors for a file pod as for whole file system restarts. When such a pure crash is triggered, an isolation file system immediately stops any file access for this file pod. It may also need to return error codes to all the processes which have opened files of this file pod. Furthermore, an isolation file system needs to clean the file pod's related system states and free resources (such as buffers in the page cache and metadata in the journal). Note that an isolation file system does not preserve or unwind file system states for transparent recovery, but instead provides the same semantics of system crashes for a file pod locally. To warrant a continuous execution while a pure crash is triggered, we need to isolate or even re-design a file pod's in-memory and on-disk states very carefully, preventing incorrect system states to propagate to other pods. We are in progress of solving this problem.

An isolation file system may also support other useful failure policies. For example, we may allow processes with opened files of a target file pod to finish their data accesses even when failures are detected. Until then, we mark the file pod as read-only or pure crash the file pod. Otherwise, immediately remounting a faulty file pod as read-only or crashing it may cause data loss for applications.

5 Quick Recovery

Recovering a whole file system is time consuming, especially running `fsck`. With the increased capacity of disks, users' file systems also easily scale to multiple TBs. Even when `fsck` can run at peak disk speed, it still takes a long time to finish checking. For example, it takes nearly seven hours to read a 2 TB disk sequentially.

Since we isolate faults for each file pod, this provides great opportunities to recover corrupted file pods efficiently. Instead of checking the whole disk, we can narrow down our target to certain file pods which triggered their failure policies.

When should we run online recovery? We utilize a file system's own internal detection mechanism to identify various failures. For example, when an isolation file system finds a file pod's block bitmap is corrupted, it will first trigger the file pod's failure policy, such as remounting as read-only. Then, custom recovery policies are executed for this file pod. An isolation file system can run checking immediately after such failures are detected. It may also run checking when the file pod is idle without any failure detected. Or it can periodically run checking for some important file pods, such as file pods storing system configuration files.

How can we improve checking efficiency? Since an isolation file system can only check a small part of the whole file system, it can provide quick checking both online and offline. Metadata of isolation file systems is isolated in such a manner that checking can be done independently for each file pod, avoiding expensive global cross checking. Furthermore, an isolation file system utilizes the file system's fault detection mechanism to provide hints for integrity checking, such as a corrupted block bitmap. This can even narrow down the checking to certain data structures. For online checking, when failures happen, the metadata of the target file pod may be already in memory, thus avoiding slow disk reads.

6 Implementation

We sketch out our initial ideas for a standard journaling file system, Ext3. Major changes are described in the following categories.

File system layout. Each block group only belongs to a single file pod, while a file pod may have multiple block groups. With this new layout, any metadata corruption can be narrowed down to a single file pod. Block groups in Ext style file systems provide a good model for data locality. Our file pod is built on top of block groups to maintain the performance benefits and provide extra fault isolation. For other file systems, such as log-structured file system, we will need a new way to map a file pod to underlying disk structures.

Data structures. The file pod structure for a block group is stored in the group descriptor of that block group in Ext3. We do not maintain extra mapping structures for file pods. When mounting an Ext3 file system, all group descriptors will be loaded into memory by default. Since all group descriptors are replicated in multiple locations, we can retrieve other replicas if needed. To get a file pod information for a file, we can easily map the file's inode to a block group, and then retrieve its file pod information from the corresponding group descriptor.

Algorithms. Data, inode, and directory block allocation / de-allocation algorithms need to be changed to be file pod based. Isolation file systems still preserve the locality property of default allocation algorithms of Ext3. But when the allocation moves across block group boundaries, Isolation file systems make sure that the target block group belongs to the same pod or it is an empty block group. Readers may be concerned about the internal fragmentation within a file pod. A possible solution is to provide a de-fragmentation tool for pods. Similar solution exists in Ext4 (online de-fragmentation).

Journaling. Ext3 consolidates multiple atomic updates from different files into a single transaction. To isolate updates from different file pods, we change the journaling mechanisms for both better reliability and performance. To provide reliability isolation, each transaction contains updates only from a single file pod. When an isolation file system updates its metadata, it will pass the file pod information to its journaling layer. The journaling layer maintains a separate transaction for each active file pod. Thus, once any failure happens in the journaling layer, we can relate the failure to a single transaction of a specific file pod. During the commit phase, three ordering points are enforced for data blocks, metadata blocks and the commit block respectively. To improve journaling performance, an isolation file system commits multiple transactions from different file pods in parallel by using multiple committing threads. In this manner, we hope to get better I/O scheduling for submitted blocks and overlap waiting time from different transactions.

Failure Policy Support. For read-only, we can mark all the block groups in the file pod as read-only and stop journaling updates for the file pod. For pure crash, we need to clean the file pod's system states, by doing a lightweight restart. This includes returning an error code to processes which are opening files in this file pod, freeing all cached memory objects, and marking errors on disk for later recovery.

Recovery Policy Support. We can utilize existing `fsck` code to conduct checking for a file pod. We need to instrument all the global failure polices in Ext3 to use our failure and recovery framework.

7 Related Work

Security / Namespace Isolation. Previous file system isolation mechanisms focus purely on the namespace. `chroot` [1] confines a process to a portion of a file system. The namespace is limited to a single directory subtree. BSD `jail` [7] is based on `chroot` mechanism. Each jail includes processes, a file system directory and network resources. Solaris zones [11] are based on `jail`. Each zone is confined to a disjoint portion of a file system. Hypervisors not only isolate each virtual machine's namespace, they also provide resource and performance isolation. Their fault isolation mechanisms still focus on process and memory faults. Both Solitude [6] and Denali [14] target on security isolation for untrusted applications. For all these solutions, they only provide namespace isolation for file systems. The underlying file-system failures are still shared across namespaces, jails, zones, and virtual machines.

File System Checkers. Windows ReFS [12] can detect and recover data corruption at runtime. Its recovery mechanism depends on metadata checksum and replicas on multiple disks. Specifically, it can only auto-recover corrupted files if the file system is run on mirrored storage devices. Furthermore, it cannot handle memory corruption and software bugs. Wafliron [10] is an online file system checker for WAFL file system. It allows data accesses from volumes not being checked. However, no details about how it conducts online checking are available. Chunkfs [5] partitions a file system into fixed size, independent chunks, and hopefully can check each chunk independently. However, cross-chunk references still exist; for example, a large file may span on multiple chunks. It is also hard to know when to trigger online checking. Furthermore, its design is only based on Ext2, without modern journaling features.

8 Conclusion

Global failures are prevalent in modern file systems, which severely harm the availability of file systems in various scenarios. We analyzed global failure policies and failure causes of existing file systems. A new file system abstraction was proposed to allow applications to manage failure policies and recovery polices explicitly. Finally, we briefly discussed the design and implementation of an isolation file system. What we present here is just a first step towards a resilient file system.

Acknowledgments

We thank Ed Nightingale (our shepherd) and the anonymous reviewers for their excellent feedback and suggestions. This material is based upon work supported by the National Science Foundation under the following grants: CNS-1218405, CCF-0937959, CSR-1017518, CCF-1016924, as well as generous support from EMC, Facebook, Fusion-io, Google, Huawei, Microsoft, NetApp, Samsung and VMware. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or other institutions.

References

[1] Change Root Directory. http://linux.die.net/man/2/chroot/.

[2] First Galaxy Nexus Rom Available, Features Ext4 Support. http://androidspin.com/2011/12/06/first-galaxy-nexus-rom-available-features-ext4-support/.

[3] Oracle Cluster File System (OCFS2). https://oss.oracle.com/projects/ocfs2, 2013.

[4] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of the 2007 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, San Diego, California, June 2007.

[5] Val Henson, Arjan van de Ven, Amit Gud, and Zach Brown. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *IEEE 2nd Workshop on Hot Topics in System Dependability (HotDep '06)*, Seattle, Washington, November 2006.

[6] Shvetank Jain, Fareha Shafique, Vladan Djeric, and Ashvin Goel. Application-Level Isolation and Recovery with Solitude. In *Proceedings of the EuroSys Conference (EuroSys '08)*, Glasgow, Scotland UK, March 2008.

[7] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *Second International System Administration and Networking Conference (SANE '00)*, May 2000.

[8] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Shan Lu. A Study of Linux File System Evolution. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST '13)*, San Jose, California, February 2013.

[9] Sean Morrissey. *iOS Forensic Analysis: for iPhone, iPad, and iPod Touch*. Apress, 2010.

[10] NetApp. Overview of WAFL_check. http://uadmin.nl/init/?p=900, Sep. 2011.

[11] Oracle Inc. Consolidating Applications with Oracle Solaris Containers. www.oracle.com/technetwork/server-storage/solaris/documentation/consolidating-apps-163572.pdf, Jul 2011.

[12] Steven Sinofsky. Building the Next Generation File System for Windows: ReFS. http://blogs.msdn.com/b/b8/archive/2012/01/16/building-the-next-generation-file-system-for-windows-refs.aspx, Jan. 2012.

[13] Satyam B. Vaghani. Virtual Machine File System. *ACM SIGOPS Operating Systems Review*, 44(4):57–70, Dec 2010.

[14] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[15] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, San Francisco, California, December 2004.