© 2014 IJIRT | Volume 1 Issue 5 | ISSN : 2349-6002

Operating System sustain for Restartable File System

Priyanka Sahni ,Nonika Sharma
Department of Information Technology, Dronacharya College of Engineering

***Abstract-* We introduce Membrane, a set of changes to the operating system to support restartable file systems. Membrane allows an operating system to tolerate a broad class of file system failures and does so while remaining transparent to running applications; upon failure, the file system restarts, its state is restored, and pending application requests are serviced as if no failure had occurred. Membrane provides transparent recovery through a lightweight logging and checkpoint infrastructure, and includes novel techniques to improve performance and correctness of its fault-anticipation and recovery machinery. We tested Membrane with ext2, ext3, and VFAT. Through experimentation, we show that Membrane induces little performance overhead and can tolerate a wide range of file system crashes. More critically, Membrane does so with little or no change to existing file systems thus improving robustness to crashes without mandate intrusive changes to existing file-system code.**

I. INTRODUCTION

Operating systems crash. Whether due to software bugs[8] or hardware bit-flips [22], the reality is clear: large code bases are brittle and the smallest problem in software implementation or hardware environment can lead the entire monolithic operating system to fail. Recent research has made great headway in operatingsystem crash tolerance, particularly in surviving device driver failures [9, 10, 13, 14, 20, 31, 32, 37, 40]. Many of these approaches achieve some level of fault tolerance by building a hard wall around OS subsystems using address-space based isolation and microrebooting [2, 3] said drivers upon fault detection. For example, Nooks (and follow-on work with Shadow Drivers) encapsulate device drivers in their own protection domain, thus making it challenging for errant driver code to overwrite data in other parts of the kernel [31, 32]. Other approaches are similar, using variants of microkernel-based architectures [7, 13, 37] or virtual machines [10, 20] to isolate drivers from the kernel. Device drivers are not the only OS subsystem, nor are they necessarily where the most important bugs reside. Many recent studies have shown that file systems contain a large number of bugs [5, 8, 11, 25, 38, 39]. Perhaps this is not surprising, as file systems are one of the largest and most complex code bases in the kernel. Further, file systems are still under active development, and new ones are introduced quite frequently. For example, Linux has many established file systems, including ext2 [34], ext3 [35], reiserfs [27], and still there is great interest in next-generation file systems such as Linux ext4 and btrfs. Thus, file systems are large, complex, and under development, the perfect storm for numerous bugs to arise. Because of the likely presence of flaws in their implementation, it is critical to consider how to recover from file system crashes as well. Unfortunately, we cannot directly apply previous work from the device-driver literature to improving file-system fault recovery. File systems, unlike device drivers, are extremely stateful, as they manage vast amounts of both in-memory and persistent data; making matters worse is the fact that file systems spread such state across many parts of the kernel including the page cache, dynamically-allocated memory, and so forth. On-disk state of the file system also needs to be consistent upon restart to avoid any damage to the stored data. Thus, when a file system crashes, a great deal more care is required to recover while keeping the rest of the OS intact. In this paper, we introduce Membrane, an operating system framework to support lightweight, stateful recovery from file system crashes. During normal operation, Membrane logs file system operations, tracks file system objects, and periodically performs lightweight checkpoints of file system state. If a file system crash occurs, Membrane parks pending requests, cleans up existing state, restarts the file system from the most recent checkpoint, and replays the in-memory operation log to restore the state of the file system. Once finished with recovery, Membrane begins to service application requests again; applications are unaware of the crash and restart except for a small performance blip during recovery.

II. BACKGROUND

Before presenting Membrane, we first discuss previous systems that have a similar goal of increasing operating system fault resilience. We classify previous approaches along two axes: overhead and statefulness. We classify fault isolation techniques that incur little overhead as lightweight, while more costly mechanisms are classified as heavyweight. Heavyweight mechanisms are not likely to be adopted by file systems, which have been tuned for high performance and scalability [15, 30, 1], especially when used in server environments. We also classify techniques based on how much system state they are designed to recover after failure. Techniques that assume the failed component has little in-memory state is referred to as stateless, which is the case with most device driver recovery techniques. Techniques that can handle components with in-memory and even persistent storage are stateful; when recovering from file-system failure, stateful techniques are required. We now examine three particular systems as they are exemplars of three previously explored points in the design space. Membrane, described in greater detail in subsequent sections, represents an exploration into the fourth point in this space, and hence its contribution.

III. GOALS

We believe there are five major goals for a system that supports restartable file systems.

Fault Tolerant: A large range of faults can occur in file systems. Failures can be caused by faulty hardware and buggy software, can be permanent or transient, and can corrupt data arbitrarily or be fail-stop. The ideal restartable file system recovers from all possible faults.

Lightweight: Performance is important to most users and most file systems have had their performance tuned over many years. Thus, adding significant overhead is not a viable alternative: a restartable file system will only be used if it has comparable performance to existing file systems.

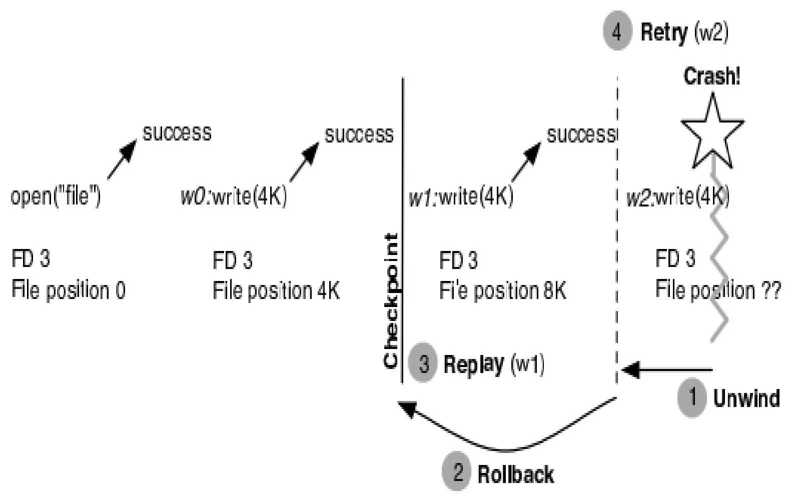
Transparent: We do not expect application developers to be willing to rewrite or recompile applications for this environment. We assume that it is difficult for most applications to handle unexpected failures in the file system. Therefore, the restartable environment should be completely transparent to applications; applications should not be able to discern that a file-system has crashed.

Generic: A large number of commodity file systems exist and each has its own strengths and weaknesses. Ideally, the infrastructure should enable any file system to be made restartable with little or no changes.

Maintain File-System Consistency: File systems provide different crash consistency guarantees and users typically choose their file system depending on their requirements. Therefore, the restartable environment should not change the existing crash consistency guarantees. Many of these goals are at odds with one another. For example, higher levels of fault resilience can be achieved with heavier-weight fault-detection mechanisms. Thus in designing Membrane, we explicitly make the choice to favor performance, transparency, and generality over the ability to handle a wider range of faults. We believe that heavyweight machinery to detect and recover from relatively-rare faults is not acceptable. Finally, although Membrane should be as generic a framework as possible, a few file system modifications can be tolerated.

IV. Overview

The main design challenge for Membrane is to recover file-system state in a lightweight, transparent fashion. At a high level, Membrane achieves this goal as follows. Once a fault has been detected in the file system, Membrane rolls back the state of the file system to a point in the past that it trusts: this trusted point is a consistent filesystem image that was checkpointed to disk. This checkpoint serves to divide file-system operations into distinct epochs; no file-system operation spans multiple epochs. To bring the file system up to date, Membrane replays the file-system operations that occurred after the checkpoint. In order to correctly interpret some operaotions



Membrane must also remember small amounts of application-visible state from before the checkpoint, such as file descriptors. Since the purpose of this replay is only to update file-system state, non-updating operations such as reads do not need to be replayed. Finally, to clean up the parts of the kernel that the buggy file system interacted with in the past, Membrane releases the kernel locks and frees memory the file system allocated. All of these steps are transparent to applications and require no changes to file-system code. Applications and the rest of the OS are unaffected by the fault. Figure 1 gives an example of how Membrane works during normal file-system operation and upon a file system crash. Thus, there are three major pieces in the Membrane design. First, fault detection machinery enables

Membrane to detect faults quickly. Second, fault anticipation mechanisms record information about current file-system operations and partition operations into distinct epochs. Finally, the fault recovery subsystem executes the recovery .

V. FAULT DETECTION

The main aim of fault detection within Membrane is to be lightweight while catching as many faults as possible. Membrane uses both hardware and software techniques to catch faults. The hardware support is simple: null pointers, divide-by-zero, and many other exceptions are caught by the hardware and routed to the Membrane recovery subsystem. More expensive hardware machinery, such as 4address-space-based isolation, is not used. The software techniques leverage the many checks that already exist in file system code. For example, file systems contain assertions as well as calls to panic() and similar functions. We take advantage of such internal integrity checking and transform calls that would crash the system into calls into our recovery engine. An approach such as that developed by SafeDrive [40] could be used to automatically place out-of-bounds pointer and other checks in the file system code. Membrane provides further software-based protection by adding extensive parameter checking on any call from the file system into the kernel proper. These lightweight boundary wrappers protect the calls between the file system and the kernel and help ensure such routines are called with proper arguments, thus preventing file system from corrupting kernel objects through bad arguments. Sophisticated tools (e.g., Ballista[18]) could be used to generate many of these wrappers automatically.

VI. FAULT ANTICIPATION

As with any system that improves reliability, there is a performance and space cost to enabling recovery when a fault occurs. We refer to this component as fault anticipation. Anticipation is pure overhead, paid even when the system is behaving well; it should be minimized to the greatest extent possible while retaining the ability to recover. In Membrane, there are two components of fault anticipation. First, the checkpointing subsystem partitions file system operations into different epochs (or transactions) and ensures that the checkpointed image on disk represents a consistent state. Second, updates to data structures and other state are tracked with a set of in-memory logs and parallel stacks. The recovery subsystem (described below) utilizes these pieces in tandem to restart the file system after failure. File system operations use many core kernel services (e.g., locks, memory allocation), are heavily intertwined with major kernel subsystems (e.g., the page cache), and have application-visible state (e.g., file descriptors). Careful state-tracking and checkpointing are thus required to enable clean recovery after a fault or crash.

VII. CHECKPOINTING

Checkpointing is critical because a checkpoint represents a point in time to which Membrane can safely roll back and initiate recovery. We define a checkpoint as a consistent boundary between epochs where no operation spans multiple epochs. By this definition, file-system state at a checkpoint is consistent as no file system operations are in flight. We require such checkpoints for the following reason: file-system state is constantly modified by operations such as writes and deletes and file systems lazily write back the modified state to improve performance. As a result, at any point in time, file system state is comprised of (i) dirty pages (in memory), (ii) in-memory copies of its meta-data objects (that have not been copied to its on-disk pages), and (iii) data on the disk. Thus, the file system is in an inconsistent state until all dirty pages and meta-data objects are quiesced to the disk. For correct operation, one needs to ensure that the file system is in a consistent state at the beginning of the mount process (or the recovery process in the case of Membrane). Modern file systems take a number of different approaches to the consistency management problem: some group updates into transactions (as in journaling file systems [12, 27, 30, 35]); others define clear consistency intervals and create snapshots (as in shadow-paging file systems [1, 15, 28]). All such mechanisms periodically create checkpoints of the file system in anticipation of a power failure or OS crash. Older file systems do not impose any ordering on updates at all (as in Linux ext2 [34] and many simpler file systems). In all cases, Membrane must operate correctly and efficiently. The main challenge with checkpointing is to accomplish it in a lightweight and non-intrusive manner. For modern file systems, Membrane can leverage the in-built journaling (or snapshotting) mechanism to periodically checkpoint file system state; as these mechanisms atomically write back data modified within a checkpoint to the disk. To track file-system level checkpoints, Membrane only requires that these file systems explicitly notify the beginning and end of the file-system transaction (or snapshot) to it so that it can throw away the log records before the checkpoint. Upon a file system crash, Membrane uses the file system's recovery mechanism to go back to the last known checkpoint and initiate the recovery process. Note that the recovery process uses on-disk data and does not depend on the in-memory state of the file system. For file systems that do not support any consistentmanagement scheme (e.g., ext2), Membrane provides a generic checkpointing mechanism at the VFS layer. Membrane's checkpointing mechanism groups several file-system operations into a single transaction and commits it atomically to the disk. A transaction is created by temporarily preventing new operations from entering into the file system for a small duration in which dirty meta-data objects are copied back to their on-disk pages and all dirty pages are marked copy-on-write. Through copy-on-write support for file-system pages, Membrane improves performance by allowing file system operations to run concurrently with the checkpoint of the previous epoch. Membrane associates each page with a checkpoint (or epoch) number to prevent pages dirtied in the current epoch from reaching the disk. It is important to note that the checkpointing mechanism in Membrane is implemented at the VFS layer; as a result, it can be leveraged by all file system with little or no modifications.

VIII. TRACKING STATE WITH LOGS AND STACKS

Membrane must track changes to various aspects of file system state that transpired after the last checkpoint. This is accomplished with five different types of logs or stacks handling: file system operations, application-visible sessions, mallocs, locks, and execution state. First, an in-memory operation log (op-log) records all state-modifying file system operations (such as open) that have taken place during the epoch or are currently in progress. The op-log records enough information about requests to enable full recovery from a given checkpoint. Membrane also requires a small session log (s-log). The s-log tracks which files are open at the beginning of an epoch and the current position of the file pointer. The op-log is not sufficient for this task, as a file may have been opened in a previous epoch; thus, by reading the oplog alone, one can only observe reads and writes to various file descriptors without the knowledge of which files such operations refer to. Third, an in-memory malloc table (m-table) tracks heap-allocated memory. Upon failure, the m-table can be consulted to determine which blocks should be freed. If failure is infrequent, an implementation could ignore memory left allocated by a failed file system; although memory would be leaked, it may leak slowly enough not to impact overall system reliability.

IX. IMPLEMENTATION

We now present the implementation of Membrane. We first describe the operating system (Linux) environment, and then present each of the main components of Mem- 6brane. Much of the functionality of Membrane is encapsulated within two components: the checkpoint manager (CPM) and the recovery manager (RM). Each of these subsystems is implemented as a background thread and is needed during anticipation (CPM) and recovery (RM). Beyond these threads, Membrane also makes heavy use of interposition to track the state of various in-memory objects and to provide the rest of its functionality. We ran Membrane with ext2, VFAT, and ext3 file systems. In implementing the functionality described above, Membrane employs three key techniques to reduce overheads and make lightweight restart of a stateful file systems feasible. The techniques are (i) page stealing: for low-cost operation logging, (ii) COW-based checkpointing: for fast in-memory partitioning of pages across epochs using copy-on-write techniques for file systems that do not support transactions, and (iii) control-flow capture and skip/trust unwind protocol: to halt in-flight threads and properly unwind in-flight execution.

X. CONCLUSIONS

File systems fail. With Membrane, failure is transformed from a show-stopping event into a small performance issue. The benefits are many: Membrane enables filesystem developers to ship file systems sooner, as small bugs will not cause massive user headaches. Membrane similarly enables customers to install new file systems, knowing that it won't bring down their entire operation. Membrane further encourages developers to harden their code and catch bugs as soon as possible. This fringe benefit will likely lead to more bugs being triggered in the field (and handled by Membrane, hopefully); if so, diagnostic information could be captured and shipped back to the developer, further improving file system robustness. We live in an age of imperfection, and software imperfection seems a fact of life rather than a temporary state of affairs. With Membrane, we can learn to embrace that imperfection, instead of fearing it. Bugs will still arise, but those that are rare and hard to reproduce will remain where they belong, automatically "fixed" by a system that can tolerate them.

REFERENCES

[1] Jeff Bonwick and Bill Moore. ZFS: The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs last.pdf, 2007.
[2] George Candea and Armando Fox. Crash-Only Software. In The Ninth Workshop on Hot Topics in Operating Systems (HotOS IX), Lihue, Hawaii, May 2003. 13
[3] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – A Technique for Cheap Recovery. In Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04), pages 31–44, San Francisco, California, December 2004.

[4] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95), Copper Mountain Resort, Colorado, December 1995.

[5] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating System Errors. In Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01), pages 73–88, Banff, Canada, October 2001.