"Otherworld" - giving applications a chance to survive OS kernel crashes

Alex Depoutovitch Michael Stumm
Department of Computer Science, Department of Electrical and Computer Engineering
University of Toronto, Toronto, Canada
{*depout,stumm*}*@eecg.toronto.edu*

Abstract

We propose a mechanism that allows applications to survive operating system kernel crashes and continue functioning with no application data loss after a system reboot. This mechanism introduces no run-time overhead and can be implemented in a commodity operating system, such as Linux. We demonstrate the feasibility of our mechanism on two example applications: JOE text editor and MySQL database server.

1 Introduction

An unexpected error within an operating system (OS) kernel, also referred to as a kernel panic, typically leads to a system reboot, which destroys all applications without giving them an opportunity to take remedial actions, such as saving data in memory to disk. In this paper, we describe a mechanism that, with a few changes to the applications, allows them to survive system reboots. Specifically, the mechanism preserves application data across kernel reboots and gives the applications the opportunity to save their state or, in many cases, to continue execution under the control of the rebooted kernel.

Many techniques have been developed over the last few decades that allow applications to handle critical application-level errors. Examples include registration of handler routines (e.g. SIGSEGV signal in UNIX) that are called by the OS when the application encounters a critical error, watchdog processes that monitor the health of the application, and dividing an application into a set of restartable components [5]. However, in all of the above approaches, the kernel itself remains a weak point in that whenever it experiences an unexpected error, there is no other software component that can reliably deal with the error. As a result, when an OS, such as Linux or Windows, encounters a critical error, it simply reboots the system. A side effect of this is immediate termination of all running applications and loss of all unsaved data.

A number of techniques exist that minimize the consequences of a kernel crash (i.e. an error in the kernel that leads to reboot), including periodic saving of application state to persistent storage, propagating application state to another system, or performing redundant calculations. These techniques may minimize or prevent data loss, but they also introduce significant overhead in terms of runtime performance and/or system cost. We are unaware of any existing technique that allows an application to survive a critical error in the OS and continue execution, even with limited functionality, without this type of overhead.

The key idea behind our work is that an OS kernel is simply a component of a larger software system, which is logically well isolated from other components, such as applications, and, therefore, it should be possible to reboot the kernel without terminating everything else running on the same system. Of course, rebooting a component as important as the kernel will be difficult without support from the applications, but we argue in this paper that this is possible with minimal and straightforward changes to application code.

Two properties of OS kernels complicate the process of rebooting the kernel without affecting running applications. The first is that the kernel resides in a privileged layer underneath all applications, so there is no other software component that can manage kernel reinitialization without destroying all applications running on top of this kernel. The second property is that the kernel itself contains data critical for running applications, such as a physical memory page maps, location of data paged to disk, opened files, and network connections, which are lost during kernel reinitialization.

To address these issues, we propose having two OS kernels resident in physical memory. The first (*main*) kernel performs all activities an OS is responsible for. The second (*crash*) kernel is passive and activated only when the main kernel experiences a critical error. When the main kernel crashes (i.e. "panics"), instead of re-

booting, it passes control to the crash kernel, which is not affected by the error because it has been passive and may be protected by memory hardware. After passing control to the crash kernel, all information on running applications in the main kernel as well as the application data still exists in memory and is accessible by the crash kernel. This allows the crash kernel to reconstruct application state and pass control to the application without loosing data. We refer to this reconstruction process as a *resurrection* of an application.

Being able to reboot a kernel and not destroy all running applications would allow us to achieve a higher level of fault tolerance and increased mean time to system failure without having to resort to expensive methods like checkpointing or systems redundancy. In addition, as we will show, our methodology opens new possibilities for performance improvements of those applications that traditionally have had to sacrifice performance for reliability, such as databases. Finally, our method supports dynamic updates of OS kernel code without requiring a system reboot, extending a concept of software rejuvenation to OS kernels.

We present the details of our approach and describe its technical aspects in the next section. In section 3, we discuss changes required to applications in order to support kernel reboot. Section 4 discusses the reliability of our technique. We present related work in Section 5 and close with concluding remarks.

2 Architecture

Our proposed mechanism of rebooting a kernel while continuing application process execution is shown in Fig. 1. Initially a computer system boots normally by loading and initializing the (*main*) OS kernel. A special region of kernel memory is reserved (e.g., 64 MB) for a (*crash*) kernel. The crash kernel image is loaded in this region and is left there untouched and uninitialized (Fig. 1a). In our implementation under Linux, we use the existing KDump mechanism to load the crash kernel into memory [10].

Any user application that wishes to be resurrected must register with the kernel a special *crash procedure*, which is located in the application address space. The address of this procedure is stored in the process descriptor of the main kernel and serves as an entry point to be called if and when the crash kernel gains control.

Whenever the main kernel experiences a critical error, it normally prints an error message and reboots the system. In our case, instead of rebooting, the main kernel passes control to the initialization point of the crash kernel. The crash kernel initializes itself normally with the only difference being that it only uses the memory region reserved for it (Fig. 1b). In order not to corrupt any pages

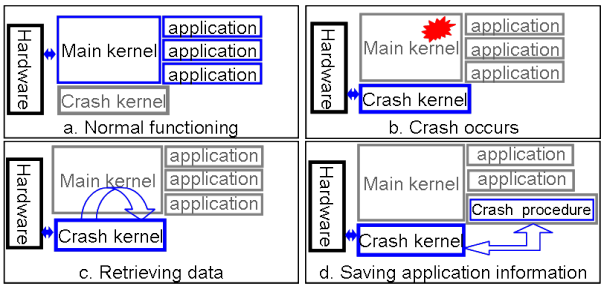

Figure 1: Surviving kernel panics
a) The system functions normally. **b)** A critical error occurs and control is passed to the *crash* kernel **c)** The *crash* kernel starts up and retrieves process data from the *main* kernel **d)** The *crash* kernel starts execution of processes that were running when the *main* kernel crashed.

that were swapped out by the main kernel, we use two swap partitions in our system: one is used by the main kernel and the other by the crash kernel. The crash kernel loads the same device drivers as the main kernel and mounts the same file systems at the same mount points. Our goal is to make the application environment in the crash kernel as similar as possible to that of the main kernel so that the same resources are available to the applications after resurrection.

After the crash kernel completes initialization, it starts a recovery phase, in which it accesses the kernel structures of the main kernel. It reads the list of processes and selects those that have registered a crash procedure. For each of these processes, we call a modified *fork* that takes the id of the process being resurrected and retrieves its page tables from the main kernel memory. For each physical memory page of the process, a new page is allocated in the crash kernel and filled with the content copied from the corresponding page of the main kernel. Hence, pages that were resident in memory at the time of the crash are copied to the memory space of the newly created process. Further, the pages that were swapped out to disk or were backed by a file (e.g. shared library) are reread from the main kernel swap partition or the corresponding backing file. This fully restores the user-mode memory space of each target process. Finally, the modified *fork* call reopens all files that were open for the application when the crash occurred and restores their file pointers (offset of the next file operation) (Fig. 1c).

The crash kernel then allocates a new stack and passes control directly to the crash procedure that was registered by the process in the main kernel. From this point on, the resurrected process continues executing with all of its global data available as if no crash had occurred (Fig. 1d). From the resurrected process's point of view,

the crash procedure is similar to a signal handler: normal execution is interrupted and control is passed to the crash procedure. Depending on the application architecture, complexity, and robustness of the crash procedure algorithm, the crash procedure can provide several levels of recovery. In the best case, the crash procedure can restore all kernel state that was lost during reboot (such as locks, threads, etc.) based on information available in the application space and then continue executing under control of the new kernel. If that is not possible, the crash procedure can simply save important process data (e.g., client session state, unsaved user document data, etc.) and restart the application.

There are several problems the crash procedure has to cope with. First, our current implementation of the crash kernel does not restore the point at which the application flow was interrupted by the error in the main kernel. Second, all of the kernel objects belonging to the process, such as network sockets, threads, etc. no longer exist. Whether they should be resurrected by the crash kernel or the applications crash procedure depends on the target object type.

The current implementation of the crash kernel reopens files that were open under the main kernel, as well as memory mapped files. In order to simplify the file reopening process, the main kernel stores the full file name and opening flags in the structure that describes the open file. The crash kernel assigns the same file descriptors to reopened files as they had in the main kernel and restores file pointers from the main kernel. Thus, the fact that files have been reopened is transparent to the resurrected application.

In order to protect the system from critical errors in the crash kernel, we plan in the future to allow the crash kernel to load another crash kernel, in which case the crash kernel starts playing the role of main kernel and the newly loaded kernel becomes the crash kernel.

3 Evaluation

To test our mechanism, we used Linux kernel version 2.6.18 for both the main and crash kernels. We added code to the crash kernel that retrieves information on processes that were running at the time the main kernel crashed and modified the *fork* system call so that it can clone processes from the main kernel to the crash kernel. These changes to the stock kernel required fewer than 1,500 lines of code. Experimentally, we were able to successfully load the crash kernel into memory and pass control to it when the main kernel experienced a critical error. After the crash kernel initialization completes, we are able to list all applications that were running on the system at the time of the crash and obtain a complete memory dump of each application.

We evaluated our mechanism using two applications: JOE text editor, an interactive application, and MySQL, a database server application.

3.1 JOE editor

JOE is shipped with the Linux distribution and is an open-source, terminal based editor capable of editing multiple documents at the same time. It is powerful editor that supports macros and syntax highlighting. The code base of JOE is around 30,000 lines of C code. We found that we did not need to know the details of JOE's internal design or data layout in order to be able to create a crash procedure that can restore all opened documents after a kernel crash. Each open document is described by a structure maintained in a linked list of open documents with a global variable pointing to the head of the list. In addition, JOE code contains a *save* function that saves a target document to a file. Our crash procedure walks through the list of all open files and calls the save function for each. Since application memory layout and all opened files are preserved during the transition of control from the main kernel to the crash kernel, all of JOE's functions continue to work unmodified, as if no kernel crash had occurred. Because we can use unmodified JOE's functions, our crash procedure required only 25 lines of code. With the exception of making a system call to register the crash procedure, adding the crash procedure did not require any additional modifications to the existing text editor code.

The JOE crash procedure we implemented is able to operate in two modes. In the first mode, the crash procedure saves to disk the contents of each file that the user had been editing at the time of the crash so that they can be opened later with no data loss due to the main kernel crash. In the second mode, the crash procedure restores the editor with exactly the same documents that were open at the time of the crash and continues running. No changes to the documents are lost, and the interactive user of the system sees the same screen he had before the crash.

3.2 MySQL database server

To evaluate our approach with a server-type application, we experimented with MySQL, the popular open source database. MySQL supports multiple pluggable storage engines (SE), which are responsible for the low-level functions that store and retrieve data. One of these, called MEMORY SE, implements memory-resident tables. All tables allocated by MEMORY SE are organized internally in a linked list, which is pointed to by a global variable, and MySQL has defined functions that scan the table and return its rows in some internal format.

Storing data in RAM instead of disk can significantly improve database server performance. For example, Oracle found performance to improve by a factor of 3 for sequential scans and by a factor of 140 for 4-way joins when all data is resident in memory [14], and Ng found memory-resident databases to perform up to 5 times faster than disk-resident databases [16]. However, a key reason why in-memory database design is problematic is that critical data is lost when the OS crashes. Our mechanism addresses this problem by preserving the critical data across kernel crashes.

The crash procedure that we created for the MySQL server iterates through the list of all allocated tables, calls the appropriate functions to retrieve data rows from these tables, and saves them to disk. Since the row format is not relevant for our purposes, we interpret the row contents as an array of bytes. After the crash procedure has saved all data, it simply restarts MySQL. Furthermore, we modified MySQL to *(i)* read during startup the content of all MEMORY SE tables from disk that were saved by the crash procedure, and *(ii)* initialize the in-memory tables with this content. Overall, MySQL has about 700,000 lines of code, and our modifications consisted of 70 new and 5 modified lines of code.

Since in the current implementation we do not restore network connections, a main kernel crash is not completely transparent to MySQL clients: they must reestablish their connections and reissue the last database request, but the contents of the in-memory tables is preserved across kernel crashes.

3.3 Testing results

The Linux kernel has over 700 run-time consistency checks, and the failure of any of them results in an OS panic. For testing, we randomly failed some of these checks. We tested both of the above examples in dozens of kernel crashes. We found that our approach allowed applications to preserve their data in every case. With the exception of few details, like broken network connections and service delays caused by the time required to initialize the crash kernel, the kernel crashes did not affect the end users of the applications. Since no extra code is executed unless a crash occurs, there is no run-time overhead.

4 Probability of successful resurrection

The practicality of our approach depends to a large extent on the probability of the bug that caused the kernel crash to have, directly or indirectly, corrupted kernel structures that are needed for recovery or a having corrupted application memory. Many errors in the OS kernel conform to the fail-stop model and cause an immediate crash [1, 11, 15, 18], leaving application data intact. However, there are some errors that do not result in an immediate OS crash, thus leading to potential data corruption.

Previous research using both artificially created and real bugs in MVS, Linux and FreeBSD showed that fewer than 5-12% of all bugs corrupt data structures manipulated by components of the OS other than the one containing the bug [1, 11, 15, 18]. Considering that most kernel crashes are caused by third-party modules [8, 9] and that we only use a relatively small subset of memory and process management related structures, we expect the probability of the structures important for process resurrection becoming corrupted to be low.

Several simple, but effective, techniques can be used to detect such corruption, should it occur. First, much data in the kernel is already duplicated in order to speed-up operations. By carefully analyzing data integrity, the crash kernel can often detect corruption. Second, one could add checksums or data duplication to the most important data structures, like process descriptors and memory maps. This would introduce some run-time overhead but will guarantee that corruption will not go undetected.

A second concern is that the kernel bug may have corrupted application-level data before crashing the OS . As was shown by Chandra and Chen, the probability of application data corruption in this case is less than 4% [6]. It might be possible to protect application space using memory management hardware [7]. Alternatively, the application can add checksums or data duplication for analyzing application data integrity. However, the performance implications of such measures would need to be carefully evaluated, but the user should be able to choose an adequate trade-off between additional system reliability and performance.

Another concern is that the crash may occur in the middle of important application or OS data structures being modified so that they are in an inconsistent state. In kernel and multi-threaded applications such data structures are usually protected by a lock. This lock is typically held by the thread modifying the data during the period the data is inconsistent and the period is typically kept short by design so as to minimize contention and thus improve performance. By examining the lock state, the crash procedure can determine if the data is potentially inconsistent and proceed accordingly.

Our mechanism cannot guarantee protection from memory corruption errors, but it should be noted that alternative techniques also cannot not provide this guarantee. For example, an error in the kernel may corrupt application data before a checkpoint is taken, corrupting the checkpoint as well [15]. Verifying data consistency at every checkpointing event usually results in high run-time overhead. In contrast, the advantage of our scheme

is that the application is fully aware at which point it is being resurrected and can therefore localize the effort required to check for integrity to only that point in time. Furthermore, even if application data is not corrupted, using checkpoints still does not guarantee successful recovery: the file system could be damaged by the kernel error, causing the checkpoint file to be corrupted. Our approach has the advantage that we already know the crash occurred immediately previously and can take appropriate precautions. For example, after the crash kernel boots, it immediately checks the file system integrity using the standard *fsck* utility *before* proceeding to application resurrection.

5 Related Work

Other research groups have investigated ways of preserving application memory state across OS crashes and subsequent reboots. Baker and Sullivan introduced the notion of a fixed sized, pinned region of memory called a Recovery Box that is accessed through a simple API and is not destroyed during a system reboot [2]. Applications need to be modified to periodically save critical application state to the Recovery Box. On startup, applications recognize that a previous instance of the application was terminated unexpectedly (possibly due to an OS crash) and recover the critical state from the Recovery Box.

Chen et al. proposed Rio – a reliable file write-back cache, whose contents is preserved during a reboot and is saved to disk after system reinitialization [7]. The authors showed that it is possible to protect contents of this cache from being corrupted by errors in the kernel by using memory protection hardware. In subsequent work, Chen et al. suggested using similar techniques to implement transactional memory and in-memory checkpoints. Our approach is more generic and can be applied to any program or kernel module rather then to certain specialized applications, such as disk caches. Bohra et al. suggest using network cards that support remote DMA (RDMA) to implement a mechanism similar to Recovery Box [4]. After an OS crash, another machine extracts the contents of the memory region with application state through an RDMA enabled network card.

All of the above techniques have the disadvantage of saving only the specially designated region of physical memory. Although they validate the concept that memory contents can survive OS crashes, they limit this memory to a specific region. This region of memory cannot be used for any purpose other than saving application state, so the choice of size is a trade-off between how much physical memory is reserved for this purpose and how much data can be saved in it by applications. Since the entire memory space of the application cannot be saved, the application has to regularly update the protected region of memory with the latest, most critical data. This introduces a constant overhead estimated by Baker et al. to be around 5% [2].

Biederman describes *KExec*, a solution for fast OS reboots, bypassing firmware and BIOS initialization [3]. He suggests loading a second kernel image into memory while the system is running. When the system administrator wishes to reboot the system, control is passed to the second kernel's initialization routine. Goyal et al. created KDump - a KExec based crash dumping mechanism that saves the full contents of physical memory to a file after a Linux system experiences a critical kernel error [10]. The second kernel's behavior is modified so that it restricts itself to a small region of memory not used by the first kernel. As a result, after the second kernel is initialized, the memory contents of the kernel that experienced the critical error is untouched and can be easily and reliably be saved to disk by the second kernel for further investigation. In this work, the second kernel is only used to create a physical memory dump for further investigation - there is no attempt to recover applications. However, the authors demonstrated that it is possible to pass control from one OS kernel to another without reinitializing system memory and running firmware and BIOS initialization procedures. They also demonstrated that it is possible to access the data of the crashed kernel. Our work takes this idea a step further: it restores full application memory state and continues to run the application.

One of the most popular techniques that allow application data to survive OS kernel crashes is checkpointing. Full application state is saved regularly on disk in order to survive an OS crash. One of the drawbacks of checkpointing is its overhead, especially for applications that use large amount of frequently changed data. Laadan and Nieh show that checkpointing a MySql server running one session takes more than a second even on a system with Fibre Channel hard drives [13]. Checkpointing in-memory databases would turn them more into a regular disk-based databases with a big impact on performance. King et al. have suggested making checkpoints of an entire virtual machine [12]. They measured the time overhead of the checkpoints taken every 10 seconds to be 15-27%. In order to reduce overhead, Srinivasan et al. suggest doing in-memory checkpoints [17]. This reduces time overhead to several percentage points even when checkpoints are taken several times a second. But in-memory checkpoints have the same physical memory overhead and restrictions as the Recovery Box approach. In contrast, our mechanism does not produce any runtime overhead. Moreover, in-memory checkpointing solutions can make use of our mechanism to withstand OS crashes by simply implementing a crash procedure.

6 Concluding Remarks

In this paper, we presented a novel approach that allows applications to survive OS kernel crashes with only minor changes to the kernel and applications. This represents an improvement over the current state of affairs where kernel crashes result in a system reboot with the loss of all volatile application state. We have implemented our approach using Linux, and tested it using two applications: JOE, an interactive text editor, and MySQL that represents a server-type application. We showed that the required changes to the applications were minimal and straightforward in both cases.

Our current implementation is just a first step in the direction of achieving our objectives and as such has quite a number of limitations. While our implementation is capable of restoring application address spaces as well as open files (including the file pointers) and memory-mapped files, it currently does not restore: sockets, pipes, threads, terminals, or the process' registers, including program counter.

In future work, we intend to add the capabilities to restore all of the above. We expect some of these to be straightforward. For example, because kernel crashes occur only when the processor is executing kernel code, user-mode thread registers will have been saved in kernel memory and hence can be restored by the crash kernel. This should allow us to resurrect not only each process address space, but all its threads as well to allow the thread to continue executing from where it was executing when the kernel crashed. On the other hand, we expect resurrecting sockets to be significantly more challenging with its multiple layers and more complex context.

Our current implementation does not introduce any runtime overhead and only requires a small and fixed amount of memory. But it does not currently check for consistency of kernel data. We intend to add consistency checking and, where necessary, add redundant data or checksums to enable such checking. This will add overhead, and we will have to evaluate its impact.

We intent to extend the capabilities of the crash kernel to (*i*) reclaim all of physical memory after having resurrected all targeted applications, (*ii*) morph itself to take over the role of the main kernel, and then (*iii*) install a new crash kernel. We also intend to allow privileged users to trigger kernel reboots. Since booting the crash kernel takes less than a minute, this feature can be used for fast system rejuvenation or hot-updating of the operating system kernel.

Finally, while the results of our testing with the two applications are encouraging, a lot more work is required. We need to experiment with many more applications and kernel crash scenarios before a final verdict is possible.

References

[1] BAKER, M., ASAMI, S., DEPRIT, E., OUSETERHOUT, J., AND SELTZER, M. Non-volatile memory for fast, reliable file systems. *Proc. of the 5th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (1992), 10–22.

[2] BAKER, M., AND SULLIVAN, M. The Recovery Box: Using fast recovery to provide high availability in the UNIX environment. *Proc. of the 1992 USENIX Summer Conf.* (1992), 31–43.

[3] BIEDERMAN, E. Kexec. http://lwn.net/Articles/15468/.

[4] BOHRA, A., NEAMTIU, I., GALLARD, P., SULTAN, F., AND IFTODE, L. Remote repair of operating system state using Backdoors. *Proc. of the Intl. Conf. on Autonomic Computing* (2004), 256–263.

[5] CANDEA, G., AND FOX, A. Recursive restartability: Turning the reboot sledgehammer into a scalpel. *Proc. of the 8th Workshop on Hot Topics in Operating Systems* (2001), 125–130.

[6] CHANDRA, S., AND CHEN, P. M. The impact of recovery mechanisms on the likelihood of saving corrupted state. *Proc. of the 13th Intl. Symposium on Software Reliability Engineering* (2002), 91–101.

[7] CHEN, P. M., NG, W., CHANDRA, S., AYCOCK, C., RAJAMANI, G., AND LOWELL, D. The RIO file cache: surviving operating system crashes. *Proc. of the 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems* (1996), 74–83.

[8] CHOU, A., YANG, J., B., C., HALLEM, S., AND ENGLER, D. An empirical study of operating system errors. *Proc. of the 18th Symposium on Operating Systems Principles* (2001), 73–88.

[9] GANAPATHI, A., GANAPATHI, V., AND PATTERSON, D. Windows XP kernel crash analysis. *Proc. of the Large Installation System Administration Conf.* (2006), 149–159.

[10] GOYAL, V., BIEDERMAN, E., AND NELLITHEERTHA, H. Kdump, A Kexec-based Kernel Crash Dumping Mechanism. *Proc. of the Linux Symposium* (2005), 169–181.

[11] GU, W., KALBARCZYK, Z., IYER, R., AND YANG, Z. Characterization of Linux kernel behavior under errors. *Proc. of the Intl. Conf. on Dependable Systems and Networks* (1993), 459–468.

[12] KING, S., DUNLAP, G., AND CHEN, P. Debugging operating systems with time-traveling virtual machines. *Proc. of the USENIX 2005 Technical Conf.* (2005), 1–15.

[13] LAADAN, O., AND NIEH, J. Transparent checkpoint-restart of multiple processes on commodity operating systems. *Proc. of the 2007 USENIX Technical Conf.* (2007), 323–336.

[14] LEHMAN, T., SHEKITA, E., AND CABRERA, L. An evaluation of the Starburst memory-resident storage component. *IEEE Trans. on Knowledge and Data Engineering* (1992), 555–566.

[15] LOWELL, D. E., CHANDRA, S., AND CHEN, P. M. Exploring failure transparency and the limits of generic recovery. *Proc. of the 4th Symposium on Operating System Design and Implementation* (2000), 289–304.

[16] NG, W. Design and implementation of reliable main memory. *Ph.D. thesis* (1999).

[17] SRINIVASAN, S., ANDREWS, C., KANDULA, S., AND Y., Z. Flashback: A light-weight extension for rollback and deterministic replay for software debugging. *Proc. of the USENIX 2004 Annual Technical Conf.* (2004).

[18] SULLIVAN, M., AND CHILLAREGE, R. Software defects and their impact on system availability: A study of field failures in operating systems. *Proc. of the 21st Intl. Symposium on Fault-Tolerant Computing* (1991), 2–9.