Recovering from Operating System Errors

Francis M. David, Roy H. Campbell
Department of Computer Science
University of Illinois at Urbana-Champaign
{fdavid,rhc}@uiuc.edu

Draft of 2006/09/18 10:00

Abstract

User applications and data in volatile memory are usually lost when an operating system crashes because of errors caused by either hardware or software faults. This is because most operating systems are designed to stop operation when some internal errors are detected irrespective of the possibility that user data and applications might still be intact and recoverable. Techniques like exception handling, code reloading, operating system component isolation, micro-rebooting, automatic system service restarts and watchdog timer based recovery can be combined together to attempt recovery of an operating system from a wide variety of errors. Initial experiments show that it is possible to continue running user applications after transparently recovering the operating system. In cases where transparent recovery is not possible, process-level recovery can be attempted as a last resort.

1 Introduction

The reliability of computer systems is an increasingly important issue in the modern world. Complex computer systems govern most of our daily lives. The operating systems that manage critical applications on these computers need to cope with a growing number of software bugs, malicious attacks and hardware faults. Resilience to errors is an important requirement of modern operating systems. The serious nature of this problem has fueled significant amounts of recent research [1, 2, 3].

When most operating systems encounter critical errors in hardware or software, they shut themselves down and have to be rebooted, resulting in a loss of currently running user applications and data. This behavior is unacceptable because the user is only concerned with application data, which might still be intact and recoverable. This paper explores changing this fail-stop behavior to allow for recovery of the system, at least partially, in order to prevent loss of user data.

Our goal is to recover an operating system (OS) transparently to user applications after an internal failure. In this paper, we present a set of error detection and recovery techniques that can be easily implemented to improve the reliability of an OS. Increased developer control over error handling using language supported exception handling, code reloading, OS component isolation, component micro-reboots, automatic system service restarts and watchdog based recovery allow an OS to recover from a wide variety of errors. A process-level recovery procedure which recovers individual user process state can be used as a last resort when all other attempts at transparent recovery fail.

In our experiments, we only consider detectable errors in OS code. For example, null-pointer dereference errors usually cause a processor interrupt. Processor interrupts normally signal errors due to memory alignment, invalid opcodes and virtual memory access control which can be due to hardware or software faults. Some OS errors such as entering infinite loops with interrupts disabled result in the system continuing to run without performing any useful work. Such errors can also be detected using external hardware such as watchdog timers.

Is the state of the system correct after recovering from an OS error? Some types of errors are simple, easily detected and fixed by techniques such as code reloading. The system is restored to a correct state in these cases. Because of the nature of some complex errors, and unknown extents of propagation, it is not possible to guarantee correctness of the system after recovering from them. But this should not be a deterrent to attempting recovery. For example, in the Linux kernel a kernel error condition known as an "Oops" is usually resolved by terminating the process that encountered the error and only halts the system if an interrupt is being serviced. The Linux kernel thus assumes process-level fault containment which is not enforced and may not be valid. Our recovery techniques make similar reasonable assumptions about errors and only recover correctly for the fault models they are designed to handle. Nevertheless, after recovering from an error, we advocate notification to the user that the system might be unstable and should be restarted after saving work.

Our research is currently targeted at the reliability of mobile devices and our ideas have been implemented on

prototype cellphone hardware based on the ARM architecture; however, our recovery solutions are not architecture specific and are generic and widely applicable. We have implemented and evaluated our recovery techniques in the Choices objected-oriented research OS [4]. We have also implemented watchdog based recovery and process-level recovery in Linux.

The remainder of this paper is organized as follows. Section 2 presents techniques for detection and recovery. We evaluate some of these techniques in section 3. We briefly discuss related work in section 4 and conclude in section 5.

2 OS Error Management Techniques

2.1 Exception handling

Choices has support for mapping processor exceptions to C++ language exceptions [5]. This allows system developers to write code to handle errors like null pointer dereferences and illegal opcodes in the OS using the C++ "catch" construct. Converting system errors into language exceptions and allowing them to be handled by system code provides developers a flexible and powerful technique to manage errors. Instead of providing generic handlers that just print out an error message (kernel panic, blue screen) and halt the system, local exception handlers can provide a more appropriate response and attempt to recover the system.

Researchers have also worked on adding exception handling support to the Linux kernel [6]. Our previous research has shown that if the compiler implements exception handling using modern table-based techniques, there is no noticeable impact on performance.

2.2 Code reloading

Transient memory faults (bit-flips) or memory corruption because of faulty code can cause errors such as invalid instructions in system code. While ECC memory can help detect and fix some transient bit errors, it cannot detect memory corruption errors caused by program execution. Code reloading is a simple and effective technique that can be used to fix such errors in OS code. The recovery strategy involves reloading the erroneous memory word from stable storage such as disk or other non-volatile memory such as flash. If the error is permanent (this can be discovered by testing), it might still be possible to recover by remapping the affected hardware page using virtual memory support.

In Choices, if an undefined instruction is encountered, the exception handler reloads the instruction from memory-mapped flash and the newly loaded instruction is executed. This recovery strategy is simple to implement; but, it cannot detect memory corruption that results in an opcode changing to another valid opcode.

Periodic code checking can be used to improve detection of memory faults. Hashing and checksums can easily be used to verify signatures of running code and trigger a reload if a fault is detected. This is a preemptive approach and can detect faults before they cause errors. This approach can also detect memory faults that cause an opcode to change to another valid opcode. Choices computes periodic CRC-32 checksums of critical kernel code and ensures that instruction memory has not been corrupted. If a checksum fails, the corresponding block is reloaded from flash. The instruction cache is then flushed to ensure that any cached corrupted instructions are eliminated. A code checksum may also be performed immediately after an OS error to ensure that the system code and recovery code is intact.

2.3 Component isolation

Component isolation helps contain the propagation of a fault. If the fault is contained within a component, recovery from any manifested errors can be targeted toward the affected component. This technique has been investigated for monolithic operating systems by the Nooks project [1, 2] using virtual memory based isolation. This property is inherent for micro-kernel operating systems.

We have implemented support for component isolation using virtual memory protection in Choices. Isolated components are provided with read-write access to defined memory regions which include a stack, a private heap and designated memory mapped hardware. The rest of the kernel is marked read-only and is therefore secure from corruption caused by errors in the component. Unlike the Nooks approach, we execute untrusted components with user privileges for increased security.

Components are encapsulated by classes in Choices and our implementation uses wrapper objects to manage switching in and out of isolation mode. This minimizes the code that needs to be changed in the components. Wrapper objects use multiple inheritance; they inherit generic isolation code from a wrapper base class for code reuse and they also inherit from the wrapped class in order to impersonate it to the rest of the OS.

We have implemented component isolation for several drivers in Choices. The console driver in Choices has been replaced with a wrapper that provides isolation and delegates work to the real console driver. This required no changes to the original driver. The watchdog timer driver has also been implemented as an isolated component. The isolation properties have been verified using various common hand-written programming errors. Component isolation only provides fault-containment. Any error encountered while executing the isolated component causes an exception to be raised. Recovery would require that the exception is appropriately handled.

2.4 Component micro-rebooting

Micro-reboot has been shown to be an effective recovery technique for application programs [7]. Applying this technique to operating systems is also feasible and can help recover from transient hardware faults and some software bugs. In the Nooks project, micro-reboots in the form of extension restarts were originally used to recover the Linux kernel. In Choices, a micro-reboot involves reinitializing the affected component or destroying and re-creating it and then retrying the request to the component. Micro-rebooting in Choices is supported by the exception handling framework. While code reloading only fixes errors in processor instructions, a micro-reboot fixes errors in kernel data structures.

The fault model for this technique is component level fault containment which can be partially enforced by component isolation.

2.5 Automatic service restarts

When a critical OS service such as the paging daemon fails, it grinds the system to a halt. If the failure of such an important process is detected, a simple restart may ensure the continued operation of the OS. The fault model assumed by this technique is single process failure with no external state corruption.

In micro-kernel operating systems, this essentially involves detecting and restarting failed system services which are run as user processes. For example, in MINIX3 [3], this job is performed by the reincarnation server. In Choices, a system process can be created so that it is automatically restarted if it encounters an unhandled exception. The process dispatcher is a special system process that loops continuously waiting for a ready process and yields to the new process. If the process dispatcher crashes, the system is rendered unusable. Therefore, in Choices the process dispatcher is implemented as a restartable process that is always recovered if it crashes.

Restarting a system process that uses locks to access shared data structures will not be successful if the process dies holding locks. Assuming that the shared data structures are not corrupted or that they can be checked for correctness and fixed [8], system recovery is only possible if all held locks are released. For this reason, Choices tracks all locks held by a process and forcibly releases any held locks when a process is terminated. We have also implemented lock tracking and forced unlocking for some types of Linux locks.

2.6 Watchdog-based recovery

External watchdog timers are used to detect errors where the OS is not performing any useful work and is in an infinite loop. A watchdog timer has to be periodically reset (kicked) by the OS and will raise an interrupt to the processor (bite) if the timer expires. Watchdog timers are normally wired to the reset pin on the processor and cause a full reboot of the system for recovery. Unfortunately, a reboot of the system results in a loss of user data and applications currently in memory.

By taking advantage of the fact that volatile memory is still preserved after a processor reset, we can reconstruct both OS and user state and continue to execute even after the reset. This novel approach avoids complete loss of user data and results in increased reliability.

We have implemented watchdog-based recovery in both Linux and Choices. The Linux implementation was part of a course project and was based on the latest version of the kernel (2.6.16.7) at the start of the project. When the watchdog bites, the processor, the memory management unit (MMU) and interrupt subsystem are reset and all registers are lost. Our modified reset handler skips the normal boot sequence if the reset is initiated by a watchdog timer. The handler turns the MMU back on, deactivates the process that was running when the reset was issued, reinitializes the interrupts and jumps to the operating system's process dispatch loop, which picks up the next ready process and runs it. The only information that is lost is the state of the process (in registers) that was running when the processor was reset. This process cannot be scheduled again and is removed from the process queue. The fault model assumed by this technique is also single process failure with no external state corruption. Watchdog recovery makes use of the lock tracking code described in the previous section to release shared resources held by the process that is deactivated.

An external watchdog interrupt can be translated to a SystemProcessNotResponding exception and dispatched to a software handler if the interrupt is wired to a NMI (Non-Maskable Interrupt) processor pin. Our current implementation cannot use exceptions for watchdog interrupts because the watchdog interrupt on our ARM hardware causes a processor reset and loss of the stack pointer register which is required for exception propagation. A processor reset also results in the loss of the data cache. Thus, if the watchdog is wired to the reset pin, enabling recovery has a serious impact on performance because normal execution cannot use the data cache.

2.7 Process-level recovery

If transparent recovery is not possible, or if the recovery process itself encounters errors, individual process state can be saved to stable storage as a last resort. After user processes are saved, a normal full reboot may be attempted and the state of the processes can be restored on the computer. This ensures that all user data is not lost when the error only affects a few applications or irrelevant OS state.

This only requires minimal support from the OS - a functioning non-volatile storage driver and user process state management code. These can be reloaded from stable storage if their integrity is in doubt. This can be easily implemented in Linux with process state checkpointing software [9, 10]. We have included support from the CRAK project [11] for checkpointing all user processes in the Linux kernel. The processes can be selectively restored after a reboot. Currently, the code requires the user to issue an explicit process save request. Ideally, this should be automatically done after attempts at transparent recovery have failed. Choices does not yet include support for process-level recovery.

The fault model addressed by this recovery technique is arbitrary OS corruption not affecting user process state and process recovery code.

3 Evaluation

All of the proposed recovery extensions have been implemented on the Texas Instruments OMAP1610 H2 prototype cellphone hardware and also on a virtual hardware platform based on the ARM Integrator board emulated by the QEMU [12] software. In order to perform some fault injection studies, we built a fault injector based on QEMU capable of injecting faults into memory, hardware registers and raising processor exceptions. In this section, we describe our initial experiences with code-reloading, automatic restarts and watchdog based recovery. The effectiveness of the other techniques has been explored previously [1, 2, 7] and is not discussed here because of space constraints.

To test the effectiveness of code-reloading, we injected 100 random memory corruption faults into CRC monitored regions holding Choices interrupt vectors and handling code. This simulates errors due to transient memory bit-flips or software bugs in drivers. 85 faults were corrected by the periodic CRC checking support, avoiding a possible future failure. 4 of the faults caused an undefined instruction interrupt and were automatically corrected. Only 11 faults caused a kernel crash. These crashes occured because errors were encountered before the periodic (5 seconds) CRC check could fix the faults. This shows that code-reloading is an extremely effective technique in reducing the number of faults that can be attributed to corrupted OS code.

Automatic process restarts, especially when applied to critical kernel processes also provide significant improvements in reliability. In a fault injection experiment that was performed 1000 times, random processor exceptions were introduced while the process dispatcher was running. We found that automatically restarting the dispatcher resulted in recovery 78.9% of the time. The failures are due to exceptions being raised during updates to critical data structures, thus causing corruption.

Our Linux watchdog recovery implementation was tested by manually writing a device driver that spawns a buggy kernel thread. The introduced bug causes the thread to eventually enter into a state in which it enters an infinite loop with interrupts turned off. Without an external watchdog, this causes the kernel to lock up and hangs the system. One of our tests consists of a script that runs the bzip2 decompression algorithm on a compressed file as a user process and instructs the device driver to spawn the buggy kernel thread. The decompression is interrupted by the buggy thread which crashes the kernel. With watchdog based recovery support turned on, the kernel recovers as soon as the processor is reset and the decompression runs to completion. In all of our experiments, the decompression was verified to be correct. In another test, we cause the kernel to crash after allowing a user to open a text editor and start to type text into it. With watchdog based recovery, the kernel is able to recover after the processor reset and the user is able to continue editing the text and is eventually able to save the file to stable storage. It should be noted that the recovery is perfect in these cases because the bug does not corrupt external kernel state. Similar experiments in Choices also result in complete recovery after a kernel hang. These experiments demonstrate that it is possible to recover an OS that has hung because of a kernel level infinite loop and continue to run user processes.

4 Related Work

There is a plethora of work in hardware and software fault-tolerance that we are unable to discuss in this short paper. We instead highlight some directly related work in application recovery after OS crashes. The recovery box approach [13] uses non-volatile memory to store application state that is restored when the system is restarted after a crash. Researchers at Rutgers have used remote-DMA in order to access the memory of a crashed system and recover application state [14].

Checkpointing can be used to recover from crashed systems running in virtual machines. VMWare and Xen [15] provide mechanisms to checkpoint running operating systems and restore them. When the OS crashes, the checkpoint can be restored, thus providing limited recovery. Compared to this approach which loses all information after the checkpoint, our recovery techniques attempt to recover currently running processes.

Self-checking code has been used to detect changes to running user applications [16]. There is also some work in detecting infinite loop errors in OS code using experimental processor extensions [17]; however, recovery is not addressed.

5 Conclusions and Future Work

Our experiments demonstrate that it is possible to increase the reliability of operating systems through simple and effective techniques such as code reloading, component isolation and automatic restarts. With the addition of external watchdog hardware support, it is also possible to detect and attempt recovery from system hangs that would otherwise remained undetected.

While micro-kernels are well suited for fault-tolerant operation because of their architecture with inherent isolation, it is also possible to reap similar benefits in a monolithic kernel through careful design.

The generic recovery techniques described in this paper can be improved by incorporating support for a framework that allows the use of developer specified policies that govern recovery actions on a case-by-case basis. We have experimented with some rudimentary support for specifying simple policies like retry counts for micro-reboots and automatically restartable processes. But there is a need for the ability to specify more complex recovery actions that can take into account dependencies and OS state checking routines. This design could potentially increase the chances of a successful recovery.

More details, reports and code related to our implementations of the recovery techniques described in this paper and the code for the QEMU based fault injector are available online at `http://choices.cs.uiuc.edu/`.

6 Acknowledgments

Part of this research was made possible by a grant from DoCoMo Labs USA and generous support from Texas Instruments. We would also like to thank Professor Ravishankar K. Iyer for helpful discussions during his class on fault-tolerant systems. Daniel Chen helped implement part of the Linux recovery code. Ganesh Bikshandi, Jia Guo and Justin Trobec helped implement parts of the component isolation code in Choices. Ramkumar Vadali and Shankar Kalyanaraman helped integrate code from CRAK for checkpointing user processes in the Linux kernel.

References

[1] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, pages 207–222, New York, NY, USA, 2003. ACM Press.

[2] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering Device Drivers. In *Symposium on Operating Systems Design and Implementation*, pages 1–16, 2004.

[3] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. Can we make operating systems reliable and secure? *Computer*, 39(5):44–51, 2006.

[4] R. H. Campbell, G. M. Johnston, and V. Russo. "Choices (Class Hierarchical Open Interface for Custom Embedded Systems)". *ACM Operating Systems Review*, 21(3):9–17, July 1987.

[5] Francis M. David, Jeffrey C. Carlyle, Ellick M. Chan, David K. Raila, and Roy H. Campbell. *Exception Handling in the Choices Operating System*. Lecture Notes in Computer Science. Springer-Verlag Inc., New York, NY, USA, 2006.

[6] Halldor Isak Glyfason and Gisli Hjalmtysson. Exceptional Kernel: Using C++ Exceptions in the Linux Kernel. October 2004.

[7] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – A Technique for Cheap Recovery. In *Symposium on Operating Systems Design and Implementation*, San Francisco, CA, December 2004.

[8] B. Demsky and M. Rinard. Automatic Data Structure Repair for Self-Healing Systems. In *Proceedings of the First Workshop on Algorithms and Architectures for Self-Managed Systems*, San Diego, California, June 2003.

[9] Eduardo Pinheiro. Truly-Transparent Checkpointing of Parallel Applications. 1998.

[10] J. Duell, P. Hargrove, and E. Roman. The design and implementation of berkeley lab's linux checkpoint/restart. Technical Report LBNL-54941, Lawrence Berkeley National Laboratory, 2003.

[11] Hua Zhong and Jason Nieh. CRAK: Linux Checkpoint/Restart as a Kernel Module. Technical Report CUCS-014-01, Columbia University, November 2002.

[12] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, 2005.

[13] Mary Baker and Mark Sullivan. The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment. In *USENIX*, pages 31–44, Summer 1992.

[14] Florin Sultan, Aniruddha Bohra, Stephen Smaldone, Yufei Pan, Pascal Gallard, Iulian Neamtiu, and Liviu Iftode. Recovering Internet Service Sessions from Operating System Failures. *IEEE Internet Computing*, 9(2):17–27, 2005.

[15] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.

[16] Bill Horne, Lesley R. Matheson, Casey Sheehan, and Robert Endre Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *Digital Rights Management Workshop*, pages 141–159, 2001.

[17] Nithin Nakka, Zbigniew Kalbarczyk, Ravishankar K. Iyer, and Jun Xu. An Architectural Framework for Providing Reliability and Security Support. In *DSN*, pages 585–594. IEEE Computer Society, 2004.