Transparent Recovery from Operating System Errors

Francis M. David
Department of Computer Science
University of Illinois at Urbana-Champaign
201 N Goodwin Ave, Urbana, IL 61801
fdavid@uiuc.edu

Abstract

Errors that occur in operating systems usually impact all user applications and may render a computer unusable. This is unfortunately the case even when the error only affects an operating system component that is not crucial to the functioning of most applications. CuriOS is a new operating system that uses lightweight distribution, isolation and persistence of state to transparently recover from errors and continue running existing applications. Errors are detected through techniques such as virtual memory protection, watchdog timers and checksums. Errors are fixed by restarting OS components without any loss of state.

1 Introduction

Users of computer systems normally interact with applications; they do not directly interact with the underlying operating systems. Unfortunately, existing operating systems usually crash when they encounter errors caused by either hardware or software faults, resulting in the failure of all running applications. This behavior may be unacceptable; especially when the error is not related to currently executing applications. For example, a fault that occurs in the networking stack should not impact a user editing a local file. The fundamental issue impeding error correction and transparent recovery of an operating system (OS) is the fact that error diagnosis is extremely difficult. Fail-stop behavior is therefore very common; for example, Windows blue screen failures and UNIX panic failures.

In the absence of accurate error diagnosis, error confinement can be used to limit damage. In monolithic operating systems, there is nothing preventing errors that occur in an OS subsystem from propagating to other parts of the OS or to applications. For example, buggy code can potentially corrupt arbitrary memory. Some microkernel operating systems improve the isolation between OS components and applications by using separate virtual memory protection domains for OS components similar to those used for user applications. An error that causes a failure of an OS component can be potentially fixed by restarting that component in a manner similar to micro-rebooting [3]. This is the approach taken by the Minix3 [8] microkernel OS, where the *Reincarnation Server* monitors and restarts failed OS components (also called servers).

Checkpointing and restoring individual OS component state is yet another possible technique that can be used to fix an error. However, ensuring a consistent system-wide checkpoint incurs overheads in terms of memory and performance. It is also not easy to assert the correctness of a checkpoint. On the other hand, microkernel server restarts have been shown to be a simple and effective technique to eliminate errors. However, this does not immediately provide transparent recovery of the OS.

Applications are usually tightly coupled with the OS services they use. OS services maintaining state information related to applications cannot be easily restarted to fix errors because they would lose the state. Also, although microkernel designs reduce inter-component error propagation, intra-component error propagation remains an issue. Errors in OS components that maintain state information related to multiple applications can potentially affect all of those applications. For example, in Minix3, an error in the filesystem service can impact all applications accessing the filesystem.

My thesis introduces CuriOS, a new operating system that restructures OS state with the goals of minimizing intra-component error propagation and reducing coupling [7]. This is accomplished by lightweight distribution, isolation and persistence of application-specific state information used by OS services. Application-specific state is stored in application-associated, but application-inaccessible memory called Server State Regions (SSRs) and servers are only granted access to this information when servicing a request. This prevents errors in servers from affecting state related to all applications. Because SSRs are not associated with the server, the coupling between applications and servers

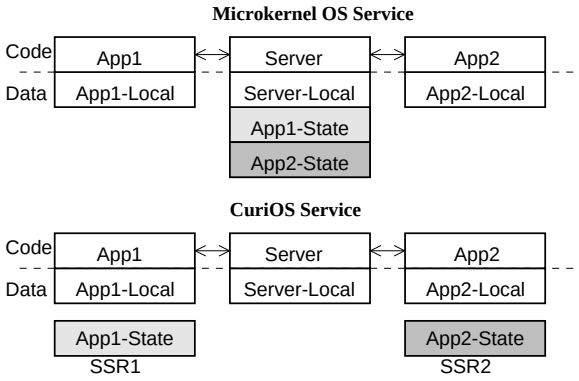

Figure 1. State Distribution

is reduced and servers can be transparently restarted. This distribution of state is illustrated in figure 1. In a sense, this creates stateless servers reminiscent of the implementation of NFS [13]. An important difference is that CuriOS does not suffer from the credential spoofing security problems in NFS [10] because SSRs cannot be modified by applications.

CuriOS provides a state management framework that can be used by OS services to allocate and manage SSRs. SSR access control is lightweight and is implemented using virtual memory maps (page tables). Many traditional OS services can be written to use SSRs for storing application-related state. The design of the service should take into account the fact that SSRs are only mapped into its address space when processing a request. If the service requires a global view of all application-related state in order to process requests, it can maintain a local redundant cached copy of the distributed SSRs. This would be necessary for services like a print spooler which needs to maintain a globally prioritized list of print jobs.

CuriOS currently supports restartable timer managers, schedulers, filesystem objects and several device drivers. Some of these components were written to use SSRs; some device drivers, like the serial port driver, are completely stateless and do not require any special support.

2 Error Detection and Recovery

This section describes the various error detection mechanisms currently supported by CuriOS and the techniques used for transparent recovery.

CuriOS is written in C++ and is based on the Choices object-oriented OS [2]. OS components are implemented as C++ classes. Isolation between OS components is implemented using virtual memory mechanisms. When errors are detected, C++ exceptions are used to signal the error to recovery routines [6].

Virtual Memory Errors: The virtual memory based isolation enforced on OS components helps detect errors that manifest as invalid memory accesses. CuriOS attempts to recover from such errors by restarting the OS component and retrying the request to the component. If several attempts at retrying the request fail, the exception is dispatched to the previous component in the call chain which is in turn restarted.

The structure of CuriOS as described in section 1 allows components to be easily restarted. Local cached state of a restarted server is re-initialized from the SSRs if necessary. Register bit-flip and memory abort fault injection experiments into the timer manager and scheduler components indicate that it is possible to transparently recover from 87-100% of errors by restarting them.

Lockup Errors: CuriOS relies on hardware watchdog timers to detect OS lockup errors. Processor resets caused by watchdog timers are converted to C++ exceptions and are dispatched to the locked up thread [5]. Request retries (without server restart) has been investigated as a possible recovery solution in Choices with about 70% effectiveness. Server restarts in CuriOS should further improve recoverability by avoiding inconsistencies in critical data structures. However, this has not been experimentally verified yet.

Some lockups are due to deadlocks. Checking for cycles in the wait-for graph is being investigated as a deadlock detection technique. This is expected to have better error detection latencies than watchdog timers. Transparent recovery can be attempted by restarting components to break cycles in the graph.

Code Corruption Errors: CuriOS uses periodic CRC checksums to ensure the integrity of executable code. Code is reloaded into RAM from stable storage if the checksum test fails. A reactive strategy retries execution of instructions after reloading them from stable storage if the processor signals an invalid opcode.

Fault injection experiments to test the effectiveness of code reloading in critical interrupt handling code show that 89% percentage of faults are detected and corrected by the periodic CRC checksum routine and the reactive invalid opcode exception handler [4].

3 Discussion

Standard fault-tolerance techniques for both hardware and software such as redundancy and majority voting can be applied to CuriOS to further enhance reliability. Redundancy can be used to protect critical information that cannot be recovered through restarts, such as the information used

by the state management framework.

If all attempts at recovery through component restarts fail, it should be possible to reboot the entire operating system as a last resort while continuing to preserve existing application state. There are several challenges that need to be overcome in order to accomplish this. A significant amount of OS state would need to be re-created based on the information in application SSRs. Dependencies between OS components needs to be tracked and state re-creation should proceed through the dependency graph. This is not yet supported by CuriOS.

There are some limitations in the current implementation of CuriOS. It does not yet tolerate errors in the C++ exception handling code or in the recovery routines. Also, this work only addresses the reliability of trusted OS components; malicious components can still cause failures or compromise the security of the system. These issues are the subject of ongoing research. Partitioning the OS into components has an adverse effect on performance. Performance optimization is yet another ongoing project.

4 Related Work

The isolation of OS components has been explored in microkernel operating systems such as L4 [11] and Minix3 [9]. However, these systems are susceptible to intra-component error propagation and can transparently recover only if the restarted server is stateless. Unlike CuriOS, restarts of a stateful service like a timer manager or a filesystem result in a loss of data and failures of dependent applications.

Several systems have been designed to provide isolation in commodity operating systems. Nooks [14] only focuses on hardware device drivers and does not address isolation of arbitrary OS components. OKE [1] requires special compiler support and also cannot be used to isolate critical OS components. SafeDrive [15] uses type-checking to improve error detection latencies and can complement the design of CuriOS.

Chorus [12] provides persistent memory regions to support hot restarts of components without loss of state. Minix3 also provides a data store service that can be used in a similar manner. But this support is not exploited by either OS and is provided as a service for applications.

Acknowledgments

This research is being performed under the guidance of Prof. Roy Campbell and incorporates significant contributions from Jeffrey Carlyle, Ellick Chan and Philip Reames. Part of this research was made possible by grants from DoCoMo Labs USA and generous support from Texas Instruments.

More information and source code for this project is available online at `http://choices.cs.uiuc.edu/`

References

[1] H. Bos and B. Samwel. Safe kernel programming in the OKE. In *IEEE Open Architectures and Network Programming*, 2002.

[2] 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.

[3] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – A Technique for Cheap Recovery. In *Symposium on Operating Systems Design and Implementation*, San Francisco, CA, December 2004.

[4] F. M. David and R. H. Campbell. Recovering from Operating System Errors. Technical Report UIUCDCS-R-2007-2831, University of Illinois at Urbana-Champaign, March 2007.

[5] F. M. David, J. C. Carlyle, and R. H. Campbell. Exploring Recovery from Operating System Lockups. In *USENIX Annual Technical Conference*, Santa Clara, CA, June 2007.

[6] F. M. David, J. C. Carlyle, E. M. Chan, D. K. Raila, and R. H. Campbell. *Exception Handling in the Choices Operating System*, volume 4119 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 2006.

[7] F. M. David, J. C. Carlyle, E. M. Chan, P. A. Reames, and R. H. Campbell. Improving Dependability by Revisiting Operating System Design. In *Workshop on Hot Topics in Dependability*, Edinburgh, UK, June 2007.

[8] J. N. Herder. Towards a True Microkernel Operating System. Master's thesis, Vrije Universiteit Amsterdam, 2005.

[9] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Reorganizing UNIX for Reliability. In *Asia-Pacific Computer Systems Architecture Conference*, pages 81–94, 2006.

[10] R. Kennell and L. H. Jamieson. Establishing the Genuinity of Remote Computer Systems. In *12th USENIX Security Symposium*, pages 295–308, 2003.

[11] J. Liedtke. On micro-kernel construction. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 237–250, New York, NY, USA, 1995. ACM Press.

[12] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. . Herrman, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the Chorus Distributed Operating System. In *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 39–70, Seattle WA (USA), 1992.

[13] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. In *Proc. Summer 1985 USENIX Conf.*, pages 119–130, Portland OR (USA), 1985.

[14] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering Device Drivers. In *Symposium on Operating Systems Design and Implementation*, pages 1–16, 2004.

[15] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harre, G. Necula, and E. Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *Symposium on Operating Systems Design and Implementation*, Nov 2006.