Recovering from Operating System Crashes

Francis David
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, USA
Email: fdavid@uiuc.edu

Daniel Chen
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, USA
Email: dchen8@uiuc.edu

***Abstract*— When an operating system crashes and hangs, it leaves the machine in an unusable state. All currently running program state and data is lost. The usual solution is to reboot the machine and restart user programs. However, it is possible that after a crash, user program state and most operating system state is still in memory and hopefully, not corrupted. In this project, we use a watchdog timer to reset the processor on an operating system crash. We have modified the Linux kernel and added a recovery routine that is called instead of the normal boot up function when the processor is reset by a watchdog. This resumes execution of user processes after killing the process that was executing when the watchdog fired. We have implemented this on the ARM architecture and we use a periodic watchdog kick thread to detect system crashes. If this thread is not scheduled periodically, the watchdog reset initiates recovery.**

I. Introduction

Reliability of computers is a very compelling requirement in the modern world where significant parts of our lives are governed by computer systems. Failures of large scale critical computer controlled infrastructure like the power grid have drastic effects on our way of life. Even small devices like personal cellphones, which have evolved to a never before seen level of complexity, need to be reliable because they are being used to manage increasing amounts of personal information. Operating systems provide a platform for users to run applications, share and store information. Reliability of operating systems is a topic that has been at the forefront of research in Computer Science for several decades [15], [17], [6], [11], [7].

There are several reasons why an operating system kernel can crash. Buggy code, bit-flips of kernel data caused by cosmic rays and faulty hardware can potentially cause a crash. Operating system crashes can be classified into two categories. Some crashes occur when the kernel detects that a serious error has occured and voluntarily halts the processor. These crashes are referred to as self-detectable crashes. Some crashes occur when the system enters a state in which no useful work is being performed and the kernel is unable to detect this condition. This is possible when for example, an interrupt service routine is in an infinite loop with interrupts turned off. Another example is when a non-preemtible kernel thread is in an infinite loop[1]. These types of crashes are referred to as self-undetectable crashes. External devices such as watchdog timers and processor support techniques [12] are required to detect such crashes.

[1]Software-lockup detection is possible in the latest versions of Linux if timer interrupts are still functional

It is accepted behavior that when an operating system crashes, all currently running user programs and data in volatile memory is lost and unrecoverable because the processor halts and the system needs to be rebooted. This is inspite of the fact that all of this information is available in volatile memory as long as there is no power failure. This paper explores the possibility of recovering the operating system using the contents of memory in order to resume execution of user programs without any data loss.

In this paper, we address recovery of operating system and user process state from self-undetectable crashes that are detected by an external watchdog timer. The watchdog timer is wired to the processor reset pin and asserts that pin on a timeout. A processor reset disables the MMU, turns off interrupts, and sets the program counter register to its reset value (0x0 on ARM). Recovery proceeds by bypassing the normal boot sequence and jumping to a recovery routine. The recovery routine re-enables the MMU, re-initializes the timers, kills the process that was running when the processor was reset, and finally re-enters the operating system dispatcher.

The rest of this paper is organized as follows. Section II describes our model for faults and our assumptions related to fault-propagation. We present our implementation platform in section III. Implementation details are presented sections IV and V. Our simple evaluation is described in section VI. We then present some related work in section VII and conclude in section VIII

II. Fault Model

The key to our recovery approach is exploiting the possibility of recovering the system directly from the contents of physical memory. This is motivated by the observation that the contents of physical memory are not affected by a watchdog induced reset of the processor. All user and kernel state is still available in memory. Thus, if the fault which caused the system hang is well contained, then recovery would simply entail killing the faulty thread and continuing to operate with the other non-faulty threads. For this reason, our operating system recovery technique is designed to work when the following conditions are true.

1) Errors are contained within the hanging kernel thread
2) Kernel task list is not corrupted

3) State of other kernel threads are not corrupted

The first assumption is reasonable for some kernel threads that performs specific services and does not access other data structures shared by other threads in the kernel space. Examples of such kernel threads are device drivers. Since the recovery of the system relies on continuing dispatching other good threads in the system after recovery, the second assumptions ensures the system can continue to operate after removing the faulty thread. The third assumption is to ensure the system is recovered to a stable and correct state. If other kernel threads are corrupted, the system would likely crash again.

III. Development Platform

The latest version of the Linux kernel at the start of this project was linux-2.6.16.7. We based our code off this version of the kernel. The target platform is the ARM processor based Integrator [1] board emulated by QEMU [5]. The property of memory preservation was initially observed on real hardware based on the TI OMAP [2] platform. Our current build environment is targeted for the Integrator.

The Integrator specification does not include a watchdog timer. As shown in figure 2 we added a watchdog timer modeled after the OMAP watchdog to the Integrator hardware emulated in QEMU. Linux already includes a driver for the OMAP watchdog. We used this same driver to manage the new Integrator watchdog.

The memory map of the Integrator in figure 1 shows the location of the bootloader, the compressed kernel and the uncompressed kernel. It also shows the location of the RAMdisk that stores the filesystem required by the kernel.

IV. Detection

Hangs in operating systems can be detected in both hardware and in software. Software detection mechanisms, however, are not completely foolproof because they usually rely on the working of the timer interrupt. If interrupts are disabled when the system hangs, the software detection mechanisms cannot detect the hang.

Hardware detection is possible using processor support techniques like the Reconfigurable Reliability and Security Engine (RSE) [12] or hardware watchdog timers.

For this project, we use a hardware watchdog for operating system hang detection. Processor reset by the hardware watchdog also provides the benefit of a known reset state to launch the recovery routine.

Linux provides a native soft-watchdog for hang detection. It displays an “Oops” message if the watchdog thread has not been scheduled for more than 10 seconds. A much better scheme that detects hangs in software is based on using processor performance counters to ensure progress of user applications. This design is still being developed by the DEPEND research group at the University of Illinois ar Urbana-Champaign.

The OMAP watchdog timer described in section III is controlled through memory mapped registers and requires specific write patterns to enable and disable the watchdog. The timeout period is configurable through a register write. A watchdog kick involves writing to a memory mapped register and this results in the timer reloading the timeout.

When recovery is enabled, we start the watchdog and spawn a kernel thread which periodically kicks the watchdog. If there is an error causing an infinite loop in some kernel thread which is not preemptable, then the kernel thread that pings the watchdog will not be scheduled, resulting in a watchdog timeout which resets the processor. Once the processor resets, the recovery routine will start.

Placing the watchdog kick in a kernel thread cannot detect

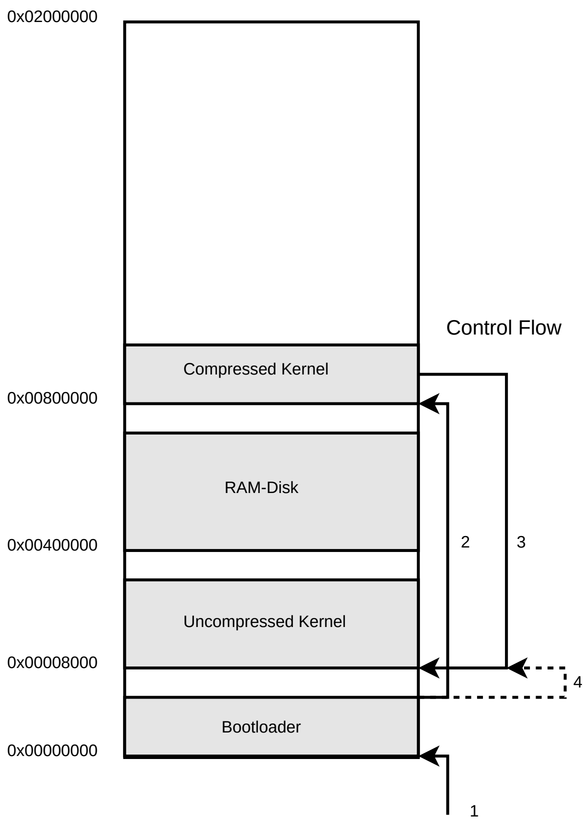

Fig. 1. Integrator Memory Map: 1-reset, 2-bootloader jumps to compressed kernel, 3-jump to uncompressed kernel, 4-bypass normal boot for recovery

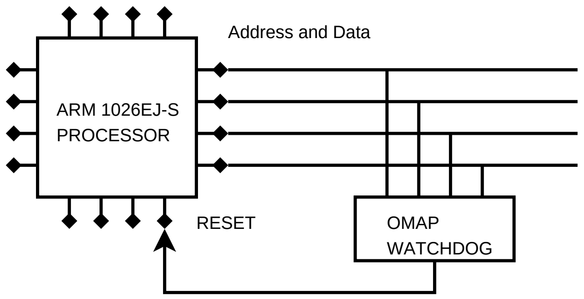

Fig. 2. Integrator Watchdog Architecture

infinite loop in preemtable kernel threads. Preemptable threads will be preempted even if infinite loops exist in the thread, and thus the periodic watchdog kick thread would still be scheduled. Adding progress indicators to all kernel threads and monitoring them in the recovery subsystem allows coverage for these type of errors as well. This can be done by instrumenting each thread to indicate its progress by setting a progress bit. These bits are periodically cleared and monitored to ensure that all threads are setting it and thus making progress. This is however not implemented in our work.

V. Recovery

When the watchdog timer resets the ARM processor, it behaves similarly to when it is first powered on. Interrupts are disabled, the MMU is disabled, the caches are disabled, and the program counter is set to the value 0. On the Integrator, the reset signal is also wired to the external interrupt controllers and causes all interrupts to be disabled at these controllers as well.

In order minimize data loss when the watchdog bites, the processor cache is set to write-through. This impacts performance significantly. Thus, this contributes to the cost of increased reliability.

On the Integrator, a small bootloader is present at address 0. During a normal power-on, the bootloader first loads the Linux kernel, which is usually stored on disk as a compressed file, into RAM. It then jumps to the compressed kernel. The compressed kernel has a header which uncompresses the rest of the kernel to a machine dependent fixed location in physical memory (0x8000 on ARM Integrator). The compressed kernel is then discarded and the memory reclaimed. We modified the bootloader so that on a processor reset, it does not reload the compressed kernel and instead jumps directly to the uncompressed kernel in physical memory. An alternative option that was also considered was to build the Linux kernel with support for eXecute In Place and place it in non-volatile memory instead of the bootloader. Eliminating the bootloader in this case requires relaxed assumptions about register contents and processor state in the kernel boot code. However, we could not get an XIP kernel to run userspace programs on the Integrator and we were unable to determine a reason. Therefore, we used the smart bootloader approach in order to jump back into Linux kernel code.

Once execution resumes at the beginning of “head.s” in the Linux kernel, a check is performed to see if the current boot is due to a watchdog reset or a normal power-on. If the reason is a watchdog reset, the normal boot sequence is bypassed. The MMU is turned back on in assembly code and a jump is performed to the C function “recover_kernel” in “init/main.c”. This routine is responsible for all other recovery actions that are performed.

Locks present a serious problem during recovery. If the process that was interrupted by a watchdog reset was holding locks, those locks will need to be released when the process is killed. During the initial phase of our work, we realized this problem when a process that was printing characters to the console was interrupted by the watchdog reset. The recovery routine was unable to print anything to the console because the console lock was not released.

To solve this problem, we have built a framework to track owners of locks and present the ability to forcibly unlock locks held by any process. This allows us to unlock all resources owned by the process that was deactivated. Tracking lock owners is another source of overhead that affects our implementation.

The recovery routine first unlocks all the locks held by the process that was running when the watchdog timer reset was issued. This process is tracked by a variable called “running” which is updated whenever there is a context switch. It then dequeues the task from the kernel runqueue so that it is never scheduled again. The interrupt controllers are reprogrammed to enable all interrupts that the kernel expects. Interrupts are then renabled on the processor. The watchdog is re-enabled and the process dispatcher is then started. The dispatcher resumes execution of processes in the runqueue.

There are several possible scenarios which can cause recovery to fail. Violations of any of the three conditions mentioned in section II can result in a state of the system that our design cannot recover from. For example, if a fault in a kernel thread causes corruption of the list of tasks maintained by the kernel, recovery is impossible because there is no way to determine the user level state of the system. Simply not scheduling a kernel thread that resulted in a hang is also definitely not the best solution. Some kernel threads are critical to the operation of the system. For example, “ksoftirqd” on Linux is responsible for running interrupt bottom-halves. If this thread dies or is no longer scheduled, interrupts cannot be processed correctly by the kernel. “kswapd” is another example of an important kernel thread that is critical to the continued operation of the system. It is responsible for managing memory paging to and from disk.

Our recovery framework is controllable from user space applications through the proc virtual filesystem on Linux. Entries in “/proc/sys/recovery” enable control of the watchdog and recovery.

VI. Evaluation

We did not have much time to perform a complete evaluation of our implementation. However, we have tested our implementation with several manually inserted infinite loop hangs inside several running non-critical kernel threads and other dummy device driver threads. Our implementation was able to recover the kernel for all of these manually inserted hangs. These tests however, ensure that the conditions described in section II are maintained. A more thorough evaluation using fault-injection and/or real kernel bugs should paint a much better picture of the extent of system recoverability through our design.

We have demonstrated in class that our implementation is able to recover from a hang in a a dummy device driver and complete execution of the bzip2 uncompression algorithm, producing correct output. We have also demonstrated to a smaller subset of the class, the recovery of a vi editing session after a kernel hang.

VII. Related Work

The Microsoft Windows operating system allows roll-backs to previous checkpoints of system configuration using a tool called "System Restore". If the system crashes repeatedly, Windows can be booted in safe mode and a previous checkpoint can be restored. This checkpointing is however only limited to system configuration and checkpoints versions of files. Unlike our work, it does not provide recoverability to currently running user processes.

Researchers have explored recovery after operating system crashes in many different ways. The recovery box approach [4] uses a region of non-volatile memory to store application specific state that is used when the system is restarted after a crash. Recently, researchers at Rutgers have investigated the use of intelligent network processors with support for Remote-DMA in order to access the memory of a crashed system and recover application state [16].

Checkpointing can also be used to recover from crashed systems running in virtual machines. VMWare [3] and Xen [8] provide mechanisms to checkpoint currently running operating systems and restore them. When the operating system crashes, the checkpoint can be restored and thus providing limited recovery. Compared to this approach which loses all information after the checkpoint, our design can recover currently running processes.

VIII. Conclusions

Recovering an operating system in the manner described in this paper is controversial. The assumption of thread level fault-containment is debatable and system recoverability after the crash of a kernel thread is questionable as well.

We feel that our assumption of thread-level fault-containment is not unreasonable because the same assumption is made by the Linux kernel when it continues to operate despite "Oopses" in the kernel, which are thread attributable errors caused by serious faults like null-pointer dereferences. The large user base of Linux and thousands of critical kernel developers have not encountered any reason to change this behavior. But there has been recent research into fault propagation in the Linux kernel [10] that suggests that fault-containment is important. It is possible to enforce containment via mechanisms like Nooks [18]. But we have not yet had the opportunity to study the behaviour of our design with such as system.

Some application-independent kernel state can be reconstructed by re-running kernel code. Some examples of such state are interrupt management and physical memory layout data structures. Reconstructing such data eliminates corruption that might have occured before the crash.

After a crashing kernel thread is killed, through either an Oops or through our recovery design, it is possible that the system is still left in an unusable state because of corruption of other state or because the thread is crucial to the functioning of the OS. We feel that this is unacceptable from the viewpoint of user applications. After a careful analysis of different classes of kernel threads, we believe that recovery routines that are thread specific will provide improved reliability. A framework that allows threads to specify policies that govern recovery can be used to tailor the recovery effort. For example, the policy can specify a cleanup routine and request an automatic restart. This thread-specific information could potentially increase the chances of a successful recovery.

We have also explored solutions that can be adopted when all attempts at recovery fail. It is possible to checkpoint just user-process state to disk. This can be done in either an application transparent [13], [9], [14] or aware manner. This allows the operating system to start afresh and load all checkpointed user processes from disk, thus ensuring fresh kernel state.

As described in section V, supporting recovery through a watchdog timer induced processor reset requires that the processor caches are set to write-through. This has significant impact on performance. A small change in the architecture that provides a Non-Maskable Interrupt on the ARM that is wired to the external watchdog can benefit our design greatly by eliminating the the problem of losing cache contents and other processor state. The NMI handler would then jump to our recovery routine.

We intend to continue working on our current design and perform a more thorough evaluation using fault-injection studies. We have already modified QEMU to support injecting faults into Linux and expect to perform several fault-injection campaigns over the next couple of months. We expect to learn more about fault propagation from these experiments and evaluate the efficacy of our recovery approach.

IX. Acknowledgments

We are thankful to Professor. Ravi Iyer for insightful comments that helped shape the final version of our project and report.

References

[1] ARM Integrator Family. `http://www.arm.com/miscPDFs/8877.pdf`.
[2] Texas Instruments OMAP Platform. `http://focus.ti.com/omap/docs/omaphomepage.tsp`.
[3] VMWare. `http://www.vmware.com`.
[4] M. Baker and M. Sullivan. The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment. In *USENIX*, pages 31–44, Summer 1992.
[5] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, 2005.
[6] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An Empirical Study of Operating System Errors. In *Symposium on Operating Systems Principles*, pages 73–88, 2001.
[7] P. J. Denning. Fault tolerant operating systems. *ACM Comput. Surv.*, 8(4):359–389, 1976.
[8] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfi eld, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.
[9] 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.

[10] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang. Characterization of linux kernel behavior under errors. *dsn*, 00:459, 2003.

[11] I. Lee and R. K. Iyer. Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN90 Operating System. In *FTCS*, pages 20–29, 1993.

[12] N. Nakka, Z. Kalbarczyk, R. K. Iyer, and J. Xu. An Architectural Framework for Providing Reliability and Security Support. In *DSN*, pages 585–594. IEEE Computer Society, 2004.

[13] E. Pinheiro. Truly-Transparent Checkpointing of Parallel Applications. 1998.

[14] J. S. Plank, M. Beck, G. Kingsley, and K. Li. **Libckpt**: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, January 1995.

[15] B. Randell. Operating systems: The problems of performance and reliability. In *Proceedings of IFIP Congress 71 Volume 1*, pages 281–290, 1971.

[16] F. Sultan, A. Bohra, S. Smaldone, Y. Pan, P. Gallard, I. Neamtiu, and L. Iftode. Recovering Internet Service Sessions from Operating System Failures. *IEEE Internet Computing*, 9(2):17–27, 2005.

[17] 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.

[18] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *SOSP '03: Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*, pages 207–222, New York, NY, USA, 2003. ACM Press.