Otherworld - Giving Applications a Chance to Survive OS Kernel Crashes

by

Alexandre Depoutovitch, B.Sc., M.Sc.

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Otherworld - Giving Applications a Chance to Survive OS Kernel Crashes

Alexandre Depoutovitch, B.Sc., M.Sc.

Doctor of Philosophy

Graduate Department of Computer Science

University of Toronto

2011

The default behavior of all commodity operating systems today is to restart the system when a critical error is encountered in the kernel. This terminates all running applications with an attendant loss of "work in progress" that is non-persistent. Our thesis is that an operating system 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.

In order to prove this thesis, we designed and implemented a new mechanism, called Otherworld, that microreboots the operating system kernel when a critical error is encountered in the kernel, and it does so without clobbering the state of the running applications. After the kernel microreboot, Otherworld attempts to resurrect the applications that were running at the time of failure. It does so by restoring the application memory spaces, open files and other resources. In the default case it then continues executing the processes from the point at which they were interrupted by the failure. Optionally, applications can have user-level recovery procedures registered with the kernel, in which case Otherworld passes control to these procedures after having restored their process state. Recovery procedures might check the integrity of application data and restore resources Otherworld was not able to restore.

We implemented Otherworld in Linux, but we believe that the technique can be

applied to all commodity operating systems. In an extensive set of experiments on real-world applications (MySQL, Apache/PHP, Joe, vi), we show that Otherworld is capable of successfully microrebooting the kernel and restoring the applications in over 97% of the cases. In the default case, Otherworld adds negligible overhead to normal execution. In an enhanced mode, Otherworld can provide extra application memory protection with overhead of between 4% and 12%.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Large software systems, such as operating systems, are extremely complex with internal state defined by many thousands of parameters. These systems can be in any one of a very large number of states at any given time. The system must be preemptible, be able to run concurrently on multiple processors sharing state, and must scale reasonably well. Moreover, these systems tend to be in constant flux with frequent bug fixes and addition of new features, so it is impossible to fully test these systems or predict how they will behave precisely in future scenarios. Modern operating systems typically contain third-party extension modules that are loaded into the kernel dynamically at run-time and that interact with the rest of the operating system. Often, those writing a kernel component or an extension use only a small part of the published interface and do not fully understand how other parts of the system work internally or interact with each other. As a result of this complexity, operating systems will likely always have bugs.

Patterson argues that faults in software are an unavoidable fact that we have to cope with [97]. He estimates that from 30 to 50 percent of computer system's total cost of ownership is spent on preparing for or recovering from system failures. This opinion is supported by others as well. For example, researchers from IBM call for switching focus from developing faster systems to developing more reliable and robust systems that

can recover from problems without human intervention [64]. Similarly, Microsoft put reliability at the top of its priority list with its trustworthy computing initiative [80].

While bugs in applications are important, they are less critical and easier to recover from than bugs in operating system kernels. When an application experiences a failure, the operating system kernel remains unaffected, and it is usually possible to restart the failed application or ensure that the system continues to provide services in some other way without affecting other applications. Unfortunately, it is not that simple when the operating system itself experiences a critical fault. In this case, there is no other code that can be trusted to run reliably. Thus, operating system kernels typically resort to the only option available to them, an unconditional and immediate system reboot.

While in most cases the reboot of the system allows the system to continue functioning, it introduces several problems. First, any system state that was not saved to persistent storage, including application data, is lost. Second, all running applications are terminated and have to be restarted after the system reboot has completed. Third, the system reboot and subsequent initialization after a crash may take a lot of time, from minutes to hours, for example, when a database server has to either rollback or roll forward a large transaction that has been interrupted by the reboot. During this time, the services provided by the system are unavailable. The cost of a failure ranges from frustration and lost work for an individual user to losses in millions of dollars for every hour of downtime of a system running financial services [48].

This dissertation addresses the issue of how applications can survive operating system failures. Throughout the dissertation, we follow the terminology established by Avizienis et al. [9]. A system *failure* or a *crash* occurs when the delivered service no longer complies with the agreed description of the system's expected function and/or service. The adjudged or hypothesized cause of a failure is a *fault* or a *bug*. In addition, we define a resource *resurrection* as a process of recreation of the resource after an operating system crash with the resource's state being the same as it had been before the crash.

The goal of tolerating operating system failures has been the focus of much prior research described in the next chapter. However, existing approaches to fault tolerance have significant deficiencies, e.g., high overhead [49, 114], inability to completely prevent data loss [46, 135], or increased system cost [60, 103]. Some approaches require complete redesign of applications or operating systems [15, 42], while others protect only from certain types of failures [36, 123]. The goal of our work was to create a fault tolerant solution that is free from the limitations mentioned above. The key considerations in our research are low or, ideally, zero run-time overhead, protection from faults in any part of a kernel, applicability to existing operating systems running on a conventional non-redundant hardware, and transparency of failures to user-mode applications.

1.1 Thesis

The thesis which is the foundation our work is that an operating system 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 operating system kernel. Following Candea et al. [28], we call such a reboot a *kernel microreboot* to distinguish it from a full system reboot.

In order to prove this thesis, we designed and implemented a new mechanism, called Otherworld, that allows us to perform a microreboot of an operating system kernel. It preserves the latest state of running applications and allows them to continue their execution after a kernel microreboot has completed. Hence, a kernel microreboot is a way to restore system functionality after an operating system kernel crash. This dissertation describes the design and architecture of Otherworld as well as its implementation in Linux. We also evaluate Otherworld's reliability and performance overhead.

Otherworld performs a kernel microreboot by passing control to an initialization func-

tion of a new kernel, the image of which was loaded in advance, while freezing execution and preserving the state of the original kernel and running applications. After initialization, the new kernel resurrects the kernel resources that were used by the applications running on top of the original kernel and continues execution of these applications.

Thus, Otherworld protects applications from kernel failures (crashes). When an operating system kernel crash occurs, instead of printing an error message and reinitializing the entire system (i.e., rebooting), we use Otherworld to microreboot the crashed kernel and continue application execution. After the microreboot, the kernel portion of the system state is completely reinitialized and is thus free from any corruption that may have been caused by the kernel fault that caused the failure. The application portion of the system state persists unchanged across such a kernel microreboot. Therefore, any application state corruption that may have been caused by the kernel fault may remain after the microreboot has completed. However, the probability of a fault inside the operating system kernel corrupting application state is low enough (1%-4%) for many applications [11, 29, 55, 63, 120], but may prevent Otherworld from being used with applications that require a high level of reliability, e.g., databases.

In order to address the possibility of an operating system fault corrupting application state, we complemented Otherworld with another mechanism that prevents unintended kernel writes to the application address space using standard memory protection hardware. This mechanism significantly reduces the probability of application state corruption. However, it introduces some run-time overhead (between 3% and 12%), so we designed the mechanism so that it can be used selectively only for the applications that require high reliability.

The Otherworld architecture is compatible with most existing applications, is capable of performing kernel microreboot completely transparent to applications running on the operating system at the time of a failure, and in many cases does not require modification of existing code or recompilation of the applications. However, microrebooting a

component as important as the operating system kernel may be difficult without support from some of the more complex applications. Nevertheless, even for such applications, we demonstrate that this is possible with minimal and straightforward changes to application code.

Otherworld is most closely related to fast checkpointing solutions, such as Discount Checking [77]. However, the key difference between Otherworld and checkpointing is that Otherworld does not require checkpoints of the system or applications to be taken at run-time, consuming additional system resources. For example, Discount Checking requires all changes to memory pages to be tracked, and original contents to be preserved using copy-on-write approach. The overhead is negligible for applications with small, slow changing memory footprint, such as interactive applications, but for applications with large memory footprints and frequent data modifications, such as database or web servers, the memory and CPU overhead will be quite significant. Instead, Otherworld is activated only when a failure occurs and uses the state of the system as it was at the time of this failure. However, checkpointing has the advantage of restoring the state as it was some time before the system failure, thus potentially reducing the probability of application data corruption. We provide a more detailed comparison of Otherworld with different checkpointing solutions in the Background chapter.

We implemented Otherworld in the Linux operating system kernel and tested it with different applications that represent different classes of applications, including interactive applications, a database server, a web applications server, and an application checkpointing solution. In our experiments we injected more than 100,000 artificial faults into an operating system kernel and were able to successfully execute a microreboot and recover from the failure caused by those faults in more than 97% of the cases.

While we implemented Otherworld in the Linux kernel running on Intel x86 processors, we didn't use any features that are unique to Intel processors or the Linux kernel. Therefore, we expect that the general architecture of Otherworld can be easily ported to

other monolithic operating systems, such as Windows, Solaris, BSD, and perhaps even operating systems based on a microkernel.

1.2 Contributions

The contributions of this dissertation are as follows:

- The design and implementation of Otherworld, the mechanism that allows a computer system to tolerate operating system kernel crashes. Upon a crash, the operating system kernel microreboots and applications continue their execution without interruption.

- The design and implementation of an application state protection mechanism that reduces the probability of a kernel bug corrupting application space.

- The experimental evaluation of the reliability of the microreboot mechanism using synthetic fault injection.

- The concept of an application-defined and application-level *crash procedure* that is invoked after a kernel microreboot, so that it can save application state even if kernel data structures (except those that are responsible for memory and process management) are corrupted as a result of a kernel fault. Implementing these functions inside applications, combined with the Otherworld mechanism, significantly increases application fault tolerance with respect to kernel failures.

Although, a number of techniques have been developed in order to tolerate faults in operating system kernels [15, 36, 42, 46, 49, 60, 103, 114, 123, 135], Otherworld has the following unique combination of advantages:

- It works with existing operating system architectures including monolithic- and microkernel-based architectures.

- It protects from faults in any part of the operating system kernel, including loadable kernel extensions.
- It does not require any specialized or other additional hardware.
- It does not introduce any run-time overhead (except for applications running with enhanced state protection enabled).
- It preserves the very latest state of applications; i.e., the state of the application at the time of the kernel failure.
- It requires minimal or no changes to applications.

Otherworld significantly increases the level of fault tolerance. Our experiments show that Otherworld is able to preserve application data in more than 97% of operating system crashes.

1.3 Limitations of Otherworld

The most significant limitation of Otherworld is that, by design, it is a best effort approach. This means that Otherworld cannot guarantee with hundred percent probability neither process resurrection nor absence of a process data corruption. Therefore, Otherworld is not suitable for in mission-critical systems, such as nuclear reactors or airplane control systems. For such systems, the N-version programming method can be used where multiple functionally equivalent programs are independently generated from the same initial specifications and executed in parallel on different hardware units [8]. Otherworld is also not suitable for systems that cannot tolerate any possibility of data corruption or inconsistency, e.g., systems that process financial transactions. Such systems can be built with fault tolerant clusters and storage solutions using persistent database transactions. However, the above systems are extremely expensive and complex to build and

maintain. Given the ubiquity of computer systems with much less strict requirements for reliability, and the millions of operating system failures such systems experience, a best-effort solution with a relatively simple implementation, wide applicability, and low or no overhead has practical value.

Other limitations of Otherworld are:

- Otherworld does not work correctly with device drivers that change during their initialization important device state (e.g., sound card drivers). It might also be incompatible with device drivers that require BIOS code to be executed prior to the driver initialization.

- Otherworld leaves the system vulnerable to operating system kernel failures for a period of time from the start of the microreboot process till the application resurrection is complete.

- Our specific kernel microreboot implementation does not automatically resurrect Unix domain sockets, pipes, pseudo terminals, System V semaphores, and futexes shared by two or more processes. It also lacks a mechanism for automatic reissue of system calls that failed because of the microreboot. This was not implemented not because of any limitations of our approach, but only because of time limitations. Currently, we rely on application cooperation to compensate for these deficiencies.

We discuss these and other limitations in more detail in the Limitations section of the last chapter.

1.4 Outline

In Chapter 2, we provide background on the types and frequencies of operating system kernel failures, failure models, and the consequences of failures, and then we present an overview of prior work on operating system failure isolation and recovery.

In Chapter 3, we give an overview of how a microreboot works and then present the Otherworld architecture. We also describe how users and applications can interact with and control the microreboot process. Chapter 4 discusses implementation details of Otherworld in the Linux kernel and describes the details of resurrection of different kernel resources used by user applications. In Chapter 5, we analyze different methods for automatically detecting kernel data corruption caused by a kernel fault and describe our mechanism for application state protection.

Chapter 6 demonstrates different ways in which applications may benefit from Otherworld. It also demonstrates changes that may have to be introduced to certain applications in order to survive kernel microreboots. In Chapter 7, we analyze the effectiveness of Otherworld protection against operating system kernel failures and effectiveness of application state protection. We also analyze the performance overhead caused by application state protection.

Finally in Chapter 8, we conclude this dissertation and indicate the directions of future research.

Chapter 2

Background and Related Work

The goal of our research is to make systems more resilient to operating system failures. For this, it is important to understand the characteristics of operating system failures. In the first section, we present available statistics on operating system crashes, their causes and tendencies in types and frequencies. The goal of this section is to demonstrate that existing efforts to make fault-free operating systems are insufficient, and software which anticipates operating system failures and knows how to cope with them is a necessity.

In order to properly recover from a failure, it is necessary to understand the amount of damage a failure can cause before being detected. In the second section, we review error propagation models and experimental data on propagation of data corruption.

The concept of a microreboot is a key part of our solution. It requires a separation of the software system into distinct components and protection of each of those components from faults from within other components. The third section reviews different methods of operating system component isolation.

After an operating system failure has occurred, been detected, and the extent of the damage is determined, failure recovery actions must be taken. In the fourth section, we describe available recovery techniques that try to ignore, minimize or, ideally, eliminate all consequences of the failure.

Finally in the last section, we summarize previous research in the area and compare our work with other similar techniques.

2.1 Failure Statistics

In this section, we present available statistics on operating system failures, their causes, and tendencies in types and frequency of failures in operating systems. Computer system failures can be caused by hardware, faults in operating system code, application failures, and human mistakes. According to Patterson, Gray, and other researchers, human mistakes are responsible for nearly half of the cases where a computer system becomes unavailable [54, 97]. Below, we review software-related failures, categorized by operating systems families. We also present data related to soft hardware failures because they behave in a way that is similar to software failures.

2.1.1 Digital Equipment Corporation Systems

Operating system crashes on the VAX/VMS platform were studied by Tang and Iyer [127]. They showed that the frequency of OS crashes ranges from 1.7 to 16 times per year on two VAX clusters. The authors also noted that OS failures often occur in bursts; once a system crashes then the probability that it will crash again soon is higher than it would be with a constant failure rate. These results are consistent with research on Nonstop-UX by Thakur et al. [128].

Murphy and Gent researched the reliability and availability of different operating systems used on computers manufactured by DEC [86]. Their study included OpenVMS, Ultrix, and Digital Unix. The authors showed that subsequent fixes to a new release of an operating system increased the time between failures 2-3 times during the first several months after the release. Subsequent fixes to the operating system do not increase its reliability, and the failure frequency remains constant.

2.1.2 Windows NT Family

A number of researchers have studied the reliability of the Windows family of operating systems. Murphy and Levidow showed that kernel crashes were responsible for 14% of system reboots [87]. Errors within driver code were the cause of 44% of the operating system crashes, followed closely with 43% by errors inside the kernel. Faulty hardware was responsible for only 13% of the crashes.

Later, Ganapathi et al. recorded data on Windows XP crashes [50, 51]. The average operating system crash frequency was more than 4 times per year. His study showed that the main source of errors were in components running in kernel space. Combined, they accounted for 88% of all system crashes, while the operating system itself was responsible only for 12% of the crashes. The most common sources of errors were unhandled page faults, page faults that occurred with disabled interrupts, unhandled kernel hardware exceptions, and threads stuck in device driver code. The most common type of driver that caused failures were graphic drivers (20% of all failures). Approximately the same percentage of errors was generated by application components that were loaded into kernel space. These results are not surprising considering the complexity of modern graphics drivers and the amount of work they have to do. Placing application components, such as antivirus checkers and firewalls, into the kernel is considered to be necessary by many manufacturers. Unfortunately, it unavoidably decreases the reliability of the operating system [95].

2.1.3 BSD and Linux

Chou et al. analyzed bugs in the Linux and OpenBSD kernels with a static code analyzer [37]. They inspected 21 different snapshots of a Linux kernel source tree from version 1.0 to 2.4.1. For each bug found, they identified the bug location, the kernel version in which the bug was introduced, and the kernel version in which the bug was

first fixed. The average lifetime of a bug in the Linux kernel was 1.8 years. Most of the errors were found in drivers; the number of bugs per line of source code in drivers was found to be 7 times higher than in the rest of the kernel. The authors found that the number of bugs in operating systems does not decrease over time. To the contrary, the number of bugs in Linux has increased as more and more functionality was added with each new version. While they found 80 bugs in version 1.0, version 2.4 contained more than 200 bugs. Results for OpenBSD resemble the Linux results with the exception that OpenBSD has a 1.3 times higher error rate than Linux [37].

2.1.4 Operating Systems Designed for Reliability

The operating systems considered so far were designed for general purpose computing, and reliability was not the primary goal. In this section, we review data available for systems designed primarily for reliability, such as the Tandem Guardian and Nonstop Unix operating systems.

Tandem is a fault tolerant system based on both hardware and software redundancy. It runs a custom operating system, initially called Nonstop kernel, later renamed to Guardian OS [15]. Analysis of this system by Lee and Iyer showed that 90% of the reported problems were caused by software [68].

Thakur et al. reviewed error statistics from the Nonstop-UX operating system designed specifically for high reliability and fault tolerance [128]. During the reviewed period of 3 years, this operating system experienced 389 unique faults (although the number of observed systems was not specified). Of these, 63% were caused by software errors and the cause of 33% remained unknown. Of the system crashes with known causes, most were caused by internal assertion statements. The second most frequent reason for system crashes were invalid pointer dereferences. Device drivers accounted for only 12% of the software errors.

2.1.5 Soft Hardware Bugs

Although the studies reviewed above indicated that hardware error rates have been decreasing, some researchers suggest that this may change in the near future [16, 83]. They predict that the rate of transient intermittent processor and memory faults, so called *soft hardware errors*, will increase if transistor sizes continue to decrease. The authors observed that cosmic rays caused crashes in large commercial sites, including eBay and AOL [16]. Similar observations are reported by Milojicic et al. [83]. They show that even with the most reliable ECC memory, soft hardware errors cannot be ignored: in 3 years, approximately 900 machines with 1GB of RAM out of 10,000 will experience soft hardware memory errors. This error rate is significantly higher for machines with less expensive RAM or with a larger amount of memory. The authors suggest that operating systems must include support to detect and recover from such errors with the help of specialized hardware. Because these errors are typically transient, they tend to manifest themselves similar to software errors.

2.2 Failure Propagation

Before trying to recover a system from an error due to a bug, it is important to determine if any data that persisted after the error during a recovery was affected by the error. The longer the system continues to function after the error has occurred, the higher the chances that some data may be corrupted as a result of the error. In this section, we review research on data corruption propagation as a result of an error. We show that, based on available studies, existing fault detection techniques are sufficient to detect errors in the operating system before data corruption propagates to application data in 82%-99% of the cases depending on the type of workload running.

2.2.1 Failure-stop Hypothesis

The *failure-stop hypothesis* was first defined by Schneider and describes an ideal case of fault detection: as soon as an error occurs, it is detected, and the system stops operating, preventing corruption of system state [112]. In the case where this hypothesis is true, we are guaranteed that no data corruption occurs and both isolation and recovery are greatly simplified. Unfortunately, it is difficult, if not impossible, to implement a system that fully satisfies the failure-stop hypothesis. Nevertheless, several attempts have been made. Schneider describes a solution for the immediate and reliable detection of faults in hardware [112]. He uses N processors executing the same code and voting on all output results that go to stable storage. This approach is prohibitively expensive and is not applicable to software bugs. Still, there have been attempts to apply it to software systems. Avizienis describes the concept of *N-version programming*: N different implementations of the same software are run in parallel with the expectation that they will fail independently [7, 8]. In work done by Gray, application code is divided into blocks, and verification code is inserted after every block to verify the results [54]. If verification fails, another version of the same code is invoked using the latest committed state of the application in the hope that the error will not reoccur. However, none of these approaches can guarantee that the failure-stop hypothesis will not be violated.

2.2.2 Limits of Generic Recovery

Assuming that the failure-stop hypothesis will be violated, Lowell et al. attempted to formulate principles of when system state should be checkpointed, so that in the case of a system crash, it could be restored with minimal losses [75]. They specified two principles for generic, user-transparent recovery from operating system crashes. The first principle requires system state to be checkpointed between every non-deterministic event (e.g., any input from external sources) and any external output (e.g., displaying message

to a user, sending any data over a network or to persistent storage). If the system satisfies the failure-stop hypothesis, this is sufficient to provide transparent recovery. But if the hypothesis is violated, and system state corrupted by an error is checkpointed before the error is detected, then a restart from the latest checkpoint will cause the system to crash again due to the corrupted state. As a result, the authors suggest a second necessary principle for recovery: application state cannot be saved between the time the error occurs and the time when the error is detected.

2.2.3 Application Data Corruption

A number of studies have estimated the probability of application data being corrupted by a bug in the operating system. These studies were done with both real bugs and artificially injected bugs.

Using several applications, including a text editor, a CAD tool, a game, and a Postgres database server, Lowell et al. tried to measure how often any of their two conditions for generic, user-transparent recovery from operating system crashes are violated [75]. When bugs were injected into the operating system, interactive application state was corrupted only in 15% of the cases, and in the case of the Postgres server application, only 3% of the crashes due to the injected bugs corrupted the application's memory. The authors came to the conclusion that checkpointing application state may be used to protect applications from operating system failures.

Research on the MVS operating system by Sullivan and Chillarege estimates the number of *addressing faults* to be 30% of the total number of faults discovered [120], where addressing faults are defined as faults that cause memory to become corrupt. Only 19% of the addressing faults corrupted memory which had an address far from the address at which the erroneous piece of code was supposed to write. Hence, only 6% of all the faults in the MVS system corrupted data structures manipulated by other components of the operating system or applications. This data, obtained from observing

real bugs, is consistent with the results obtained from injecting artificial bugs by Lowell et al. [75].

The number of addressing faults in BSD 4.1 and 4.2 was estimated by Baker and Sullivan to be 12% of the total faults within the operating system [11]. This is an upper bound on the probability of corrupting memory outside the faulty module of the operating system. Although the BSD study does not tell us how many of the addressing faults corrupt the memory not directly manipulated by the erroneous code, if we assume it to be approximately the same as in the MVS system, then only 2% of the operating system errors corrupt data that belongs to other parts of the system.

Bug propagation in the Linux operating system was investigated by Gu et al. by using bug injection techniques [55]. Approximately 95% of the crashes occurred because of one of the following reasons: a NULL pointer dereference, an unhandled kernel paging request, an invalid opcode or a general protection fault. About 90% of the injected bugs did not propagate outside the subsystem in which the bug was injected. These results are consistent with similar work by Kao et al., where faults were injected in Unix-like operating system kernels [63].

A thorough investigation of how often errors in the operating system corrupt application data was done by Chandra and Chen [29]. When application-specific recovery was used, corrupted state was saved in only 4% of the cases for the interactive applications and in 1% of cases for the Postgres server.

Application data, such as file buffers, may be stored not only in application space, but also in kernel space. A study conducted by Ng et al. determined that the file data stored in memory is corrupted only in 1.5% of the operating system crashes [92]. The number is only 0.4% larger than the percentage of faults that corrupt files on disk. The authors came to the conclusion that files in memory are approximately as safe as files on disk.

The research discussed above shows that the probability of application data being

corrupted by a bug in the operating system varies from 1% to 15%, depending on the operating system, running applications, and specific study. Data corruption in most cases is limited to a subsystem that contains a bug.

2.3 Fault Isolation

The term *fault isolation* describes techniques that divide software logically or physically into smaller components to ensure that a bug in one component cannot affect any other component. Candea and Fox advocate a *crash-only design* where software is designed to be ready to crash at any time [27]. The software must be divided into isolated components that communicate through requests. Each component must be able to retry a request should the target component have crashed and then restarted. In this section, we describe attempts to apply this design to operating system kernels. With fault isolation, it may be possible to restart a component after it fails without affecting other components (assuming components can tolerate the restart of other components). Isolation can be achieved by placing different components of an operating system into different address spaces or different virtual machines or by careful run-time verification of the components to ensure there are no cross component memory accesses.

2.3.1 Isolation with Address Spaces

The first section of this chapter showed that device drivers and other third party software that runs within the kernel memory space are responsible for the majority of operating system failures. Therefore, much effort has gone into isolating kernel components and drivers from each other so that a faulty component can not damage other parts of the system. The idea of separating kernel components by isolating them into separate address spaces is the foundation of the *microkernel* operating system design. The microkernel design also dictates that only essential services run in privileged mode, while everything

else is run as user processes in separate address spaces. Essential services typically include memory and processor resource management as well as inter-process communication. The first operating system implemented with a microkernel was the Nucleus operating system [26]. There are many other microkernel-based operating systems, such as Thoth [34], Mach [1], Chorus [5], L4 [74], K42 [4], Minix [126], CuriOS [42], just to mention a few of them.

Cheriton and Duda consider even microkernels not to be small enough. They suggest moving functionality of monolithic kernels and even microkernels out of privileged mode and placing them into so called *application kernels* [35]. The application kernel is executed at a non-privileged level and is situated between a privileged mode kernel, called a *Cache kernel*, and user applications. Another extreme case of microkernel design is *Exokernel* [47, 62]. In this architecture, operating system kernel functionality is limited to protecting and multiplexing hardware resources and nearly all functionality, including inter-process communication, is placed into application level components.

Chapin et al. used an alternative approach to isolation by exploiting features of multiprocessors [31]. Instead of running a single small micro- or exo- kernel, they suggest running several instances of a monolithic kernel, each on a different set of processors. They developed an operating system, called Hive, which is based on IRIX, a Unix-like operating system developed by SGI. Hive targeted the CC-NUMA Flash multiprocessor that consists of several nodes, each with its own processor, cache, local portion of memory, and I/O devices. Special memory management logic was able to disallow writes to any local page from any remote node. All writes to memory pages of other nodes were intercepted by the local kernel and executed using remote procedure calling, completely transparent to applications. A node was allowed to write to a remote memory page only if this node was executing a part of application that owned the memory page. Failure of one of the kernels affected only applications that use resources (e.g. memory pages, disk resources) managed by that particular kernel. This is particularly important on large-

scale systems that contain hundreds of nodes, because the probability of a hardware or software failure is much higher with the large number of nodes.

A microkernel architecture provides a high level of fault isolation and simplifies recoverability. The main disadvantage of microkernels is the performance overhead associated with the increased number of address space crossings, making them less efficient than traditional, monolithic kernels (although this is perhaps arguable). As a result, most popular modern operating systems, such as Linux, Mac OS, and Windows, use a monolithic design and even have the tendency of moving an increasing number of services into the kernel address space.

Although microkernel systems usually are capable of isolating faults in kernel components or drivers, a common problem is that the crash of an important kernel component may still leave the system unusable. For example, the crash in a file system driver, even with a subsequent driver restart, may result in an inconsistent disk state and failure of all disk operations that have been executing at the time of the crash. To address those problems, designers of the CuriOS operating system proposed that operating system components store part of their critical state in the clients address spaces and access this state using a special interface provided by the kernel [42]. On a component failure, the component is restarted and obtains access to the state stored in the clients by its previous instance. The performance overhead of this approach is still to be evaluated. Also, the authors show that a microkernel design doesn't necessarily prevent an error in one component from propagating into other components. Their experiments determined that in 6% of the cases, the operating system ultimately crashes because of error propagation.

2.3.2 Isolation with Virtual Machines

Another way to prevent one software component from corrupting another is to execute the components in different virtual machines. Given that a virtual machine monitor does not have bugs, this can provide a reliable solution. This approach, called DD/OS, is

described by Levasseur et al. [71]. In their work, the authors suggest that each driver must run in its own virtual machine, which contains *(i)* an operating system for which this driver has been developed, *(ii)* the driver itself, and *(iii)* special code that maps requests to the driver originating from other virtual machines or from the virtual machine monitor to an interface provided by the driver.

The DD/OS approach provides not only complete driver isolation, but also allows reuse of existing drivers from any operating system as well as simultaneous usage of drivers designed for different operating systems. The cost for this solution is the need to write mapper code for each device class, as well as extra processor and memory usage.

Similar work by Fraser et al. suggests a new device driver framework based on the Xen virtual machine monitor [49]. The framework requires all devices to be categorized into several device classes. All devices of the same class communicate with the operating system through a strictly defined interface. The operating system contains one simple unified driver per device class that serves as a proxy between the operating system and a real device driver. The real device drivers run in separate virtual machines independently from the guest operating systems. When compared to the Linux operating system running on the same hardware, the overhead of this protection scheme was as high as 30%.

Isolation of kernel components by using virtual machines provides excellent protection from misbehaved kernel extensions, but it is slow and requires significant changes to the driver and operating system architecture. Also, it is important to note that it does not offer any protection from bugs within the operating system kernel.

2.3.3 Software-based Fault Isolation (SFI)

Microkernel architectures provide good fault isolation and simplified recovery. However, isolating kernel components using different address spaces results in higher overhead compared to monolithic kernel designs because of the required cross address space com-

munications and an increase in the number of context switches. In order to address this issue, several research groups have suggested implementing device drivers, kernel extensions, and other kernel components within the kernel address space using type-safe code. Type-safe data access can be achieved by writing code in a type-safe language or by adding run-time verification to the code written in a type-unsafe language. Both of these approaches are referred to as *software-based fault isolation* (SFI). With SFI, kernel components and extensions are protected from each other, and no context switches are required during calls between different kernel components.

One example of a system written in a type-safe language is the SPIN operating system [18]. In this system, the kernel itself and all modules that are executed in the kernel address space are written in Modula-3 that guarantees type-safety, and prevents drivers or kernel extensions from executing privileged instructions [89]. The authors compare performance of the SPIN operating system with that of the DEC OSF/1 microkernel and show that performance of SPIN is generally better.

Another example of an operating system kernel that uses language-based protection is the Singularity system [58, 118]. Only a small hardware abstraction layer is written in C and assembly language. All other components are written in a type-safe Sing# programming language that is compiled into byte-code. All kernel components are divided into trusted and verifiable categories. The trusted components include a garbage collector, a byte code compiler, and a static code verifier. Other kernel components, as well as loadable extensions, are untrusted and have to be verified. Verification is possible because applications are written in a verifiably type-safe language and stored in a form of intermediate byte code that can be statically checked for potentially unsafe memory accesses before execution. The static code verifier prevents execution of any unsafe code. Since the code has been verified to be safe, data of one kernel component cannot be corrupted by other faulty or malicious components, and hence there is no reason for protecting different parts of the kernel using other techniques.

Both SPIN and Singularity offer good protection of kernel data from erroneous or malicious code. Unfortunately their performance is still worse than that of traditional monolithic kernels (although often better than the performance of microkernel operating systems) [18, 58]. Moreover, a system based on type-safe programming languages require rewriting of existing software. To address this problem, Wahbe et al. and Small and Seltzer suggest binary instrumentation of device drivers and kernel extensions to ensure that they do not execute illegal instructions or access invalid memory regions [117, 132]. Small and Seltzer developed the VINO operating system [116, 117], in which extensions are written in C or C++ and compiled by a special compiler that is shipped with the operating system. Verification instructions are inserted into the code. These instructions check the validity of all memory and resource references at run-time. Also, verification instructions are inserted to verify the validity of function calls and to prevent the code from modifying itself. Later work, done by Seltzer et al., improves VINO's safety mechanisms with transactions[114].

2.3.4 Modifying Existing Operating Systems

Currently, operating systems with a traditional, monolithic design are installed on more than 98% of computers [90, 115]. The methods of fault isolation described in the previous sections either require a completely different architecture or the rewrite of the operating system in a different language. This prevents wide adoption of these methods, since creation of an operating system with a new architecture usually means porting existing software and, therefore, it is costly and labor-consuming task. Because of this, several research groups have targeted existing operating systems, trying to make them more reliable by introducing improvements to the existing code base.

Currently, popular operating systems running on x86 architectures do not make use of more than two out of four processor protection rings [113]. The Palladium system developed by Chiueh et al. is an attempt to make use of multiple protection rings offered

by the hardware [36]. Palladium is based on a Linux kernel and offers advanced kernel protection from misbehaved extension modules and drivers. Palladium loads the drivers into a separate segment with a limited size that belongs to a less privileged protection ring. Thus, the driver code cannot directly access kernel data. Palladium does not protect different parts of the kernel from each other or against erroneous DMA use. Palladium requires that the interface between the drivers and the kernel be modified and all drivers must be rewritten.

Nooks, developed by Swift et al., attempts to protect from many driver-based faults with few changes to the operating system and driver source code [125], by introducing an isolation layer between the operating system kernel and a driver. All calls between the operating system and drivers are intercepted with the help of proxies. The primary purpose of this layer is isolation: it forces execution of the driver to within a protection domain. When the driver code is executed, all memory that is not intended to be accessible by the driver is write-protected by hardware. A second purpose of the Nooks layer is to track all resources allocated by the driver. This allows the freeing of all resources in the case of a driver failure. It allows validation of resource references passed to kernel functions by the driver. Finally, the last and the most important purpose of Nooks is recovery. Whenever Nooks detects that a driver behaves incorrectly (i.e. performs an invalid memory access, passes an invalid argument, or executes an invalid operation), it releases all resources allocated by the driver and replaces the driver with a new instance. However the Nooks protection layer can add a significant performance penalty to executing applications. Processor utilization in a network benchmark grew from 39% to 81%. Compilation time increased by 10% on a system with only a single file system driver protected by Nooks. Performance of a kernel HTTP daemon decreased by 60% when it was executed within a Nooks protection domain.

While an operating system can, in many cases, survive a driver failure, a driver restart often results in applications crashing. Applications are not typically prepared for

driver calls to fail or for device state to be reinitialized. This problem is addressed by complementing Nooks with a *shadow driver* [124]. A shadow driver is device class specific and intercepts all calls from the kernel to the actual driver and from the actual driver to the kernel. It records all calls that might change the state of a device. When the operating system kernel reloads the failed driver, the shadow driver replays all previously recorded calls that may have changed the driver's or device's state. After replay of the requests, both the device and the driver are in a consistent state ready to serve new requests as if no failure has occurred.

Nooks and its shadow drivers protect from bugs in relatively simple device drivers. Sundararaman et al. target a much more complex kernel component, namely the file system [123]. They designed Membrane, a mechanism that detects file system failures and restarts the file system module preserving its state transparently to applications. The general approach is similar to Nooks with a few important differences. In order to reduce performance overhead, Membrane lacks protection domains and relies on the operating system for failure detection. State tracking was much more elaborate because file systems are significantly more complex and tightly integrated with the operating system than most device drivers. It involves the periodic checkpointing of the file system to disk and the logging of all file system calls between checkpoints. Nonetheless, the resulting overhead of Membrane on the set of benchmarks described in the paper is below 2%.

One of Nooks' disadvantages is that it targets only device drivers but doesn't provide protection from faults inside the operating system kernel itself or complex kernel extensions. Lenharth et al. address this limitation by treating a request served by the kernel, e.g., a system call or a hardware interrupt, rather then operating system component as a basic recovery unit [70]. All changes to system state resulting from the request are logged and can be rolled back if the kernel experiences a failure while serving the request. However, this approach is characterized by extremely high overhead of more then 500%

when running the Postmark benchmark.

2.4 Failure Recovery

Failure recovery tries to minimize or, ideally, eliminate all consequences of component or system failures. Several basic techniques are the foundation of many failure recovery methods. Below, we discuss failure recovery methods based on checkpointing, logging, redundant calculations, data retrieval, and a novel approach called failure oblivious computing, which advocates that sometimes the best way to recover from a failure is to ignore the failure or retry the request that caused the fault in a slightly different environment.

2.4.1 Checkpointing and Logging

One of the oldest, most versatile, and well researched failure recovery techniques is *checkpointing* [30, 65]. Checkpointing is the process of making a snapshot of the running system's state so that the state of this system can be restored later. Checkpointing has been used in many systems using different techniques, including a compiler based approach [72], user level libraries that are compiled with applications [98, 133], kernel level checkpointing [52], and the checkpointing of parallel applications running on a computational grid [43, 110].

Banatre et al. suggest storing checkpoints in a *stable transactional memory* (STM), a physical memory that allows saving data with transactional guarantees on multiprocessor systems [14]. Muller et al. describe a fault tolerant modification of the Mach operating system using STM [84]. Periodic, consistent checkpoints of processes are stored to STM in order to be able to retrieve the latest checkpoint after a failure. The authors prefer STM to a hard drive due to performance and access granularity reasons. They come to the conclusion that hardware-based STM is not effective due to the complexity of the hardware design and due to the conflicts between the STM protection mechanism and

internal caches of microprocessors. Because of the conflicts, the processor data caches have to be disabled. As a result, the authors implemented STM in software. They used two computers connected by a network. One computer served as the primary node, where all computations were executed. The second was used as stable storage, where all checkpoints are saved. Overhead for non-interactive, processor-limited applications was as high as 25%. The time to checkpoint the system, running an instance of the emacs editor, was about one second, during which time the system was unresponsive to the user.

The main drawback of checkpointing is that it usually has large overheads, especially for applications that consume large amounts of memory and change their data frequently. Because of this overhead, checkpoints cannot be done too frequently. For example, Wang et al. take checkpoints every 30 minutes [133]. Many techniques have been developed to increase the speed of checkpointing, e.g., incremental checkpoints that only checkpoint data that has been changed since the previous checkpoint [46, 98, 135]. Another technique is asynchronous checkpointing, where checkpointing and program execution occur in parallel [73].

Another drawback of checkpointing is that after a system crash and subsequent restoration from the recent checkpoint, all changes to system state made after the checkpoint are lost. A technique for restoring system state that complements checkpointing or can be used separately is *logging* [22, 61, 65, 119]. Logging is based on the fact that most computations are deterministic; it saves all inputs to the system, and, when it is necessary to restore system state, these inputs are resubmitted to the system and the system recomputes its state. Often, logging is done between checkpoints in order to prevent data loss. The problem with logging is that the results of calculations can depend on the exact sequence of external events, such as thread scheduling or signals. This sequence may be difficult to reproduce using logs.

Laadan et al. developed a continuous checkpointing low-overhead mechanism called

DejaView that targets desktop workloads [67]. DejaView is capable of producing incremental checkpoints of all user processes at the rate of up to 1 checkpoint per second incurring execution slowdown of less than 10% on the set of benchmarks described in the paper. DejaView delays writing checkpoints to a stable storage, thus requiring operating system to be stable. Also, DejaView produces significant disk I/O overhead ranging from 3 to 15 MB/sec even for workloads with memory footprint of several megabytes. This complicates use of DejaView to protect server applications with high I/O requirements and large memory footprints, such as databases.

A large survey of logging and checkpoint-based protocols was authored by Elnozahy et al. [45].

2.4.2 Redundant Calculations

Fault tolerance can also be achieved by mirroring software and/or hardware, where some or all computations are done multiple times, in parallel or sequentially. In the case where one computation fails, another computation can be used to produce the results. Redundancy can be obtained by executing the program on multiple hardware units, by a software-only solution, or by a combination of both.

The Nonstop kernel is an operating system designed to work on specialized hardware where most of the resources, such as processors, I/O buses, controllers, disks, power supplies, etc. are redundant [15]. The programming model of the Nonstop kernel uses *pairs of processes* in order to achieve fault tolerance. One of the processes in the pair is considered to be the primary and the other is the backup. The same approach was used in the Nixdorf [22], Auragen [21], and Sequoia [79] fault tolerant systems. A similar approach is used in air traffic control systems [41].

Both hardware and software-based computations as well as checkpointing were used in a Unix-compatible system called Integrity S2 [60]. This system runs on fully redundant proprietary hardware. All processors execute the same code. Each instruction's inputs

and results are voted on on a bit by bit basis. The operating system receives an interrupt if a discrepancy in processor execution results is found. In order to reduce the consequences of bugs inside the operating system itself, more than 1000 checks were inserted in its code. If any of the check conditions are violated, a check-specific recovery routine is invoked so that it can try to repair the damage caused to the system without rebooting.

Specially designed hardware for redundant systems can be extremely expensive. Because of this, several researchers have investigated the possibility of performing redundant calculations by means of software, without any specialized hardware. Parallel execution of the same code on two processors with hardware result comparison is probably the most effective way to detect transient hardware errors. This can be emulated to a certain degree by software.

Oh et al. describe the *error detection by duplicate instruction* (EDDI) technique where they target transient errors in particular [96]. Using a special compiler, processor instructions are duplicated and their results are compared by additional code injected by the compiler. The results and operands of duplicate operations are stored in different processor registers and memory locations. The authors estimate that this technique can detect up to 98% of transient hardware errors. No special hardware is required, but this method effectively halves memory size, cache size, and the number of processor registers available. In addition, their technique introduces more than 60% overhead on the benchmarks described in the paper.

Later work by Reis et al. introduced a modification to this method called SWIFT that lowers both memory and processor overhead [103]. They make use of ECC memory correction technology which is widely used in commodity hardware. Using ECC memory allows them to eliminate redundant memory stores and significantly reduces memory overhead. SWIFT also eliminates some redundant checks while still providing the same error detection guarantees. This reduces processor overhead to 40% on the same set of benchmarks.

An idea similar to that of the Nonstop kernel was implemented using the Mach operating system by Accetta et al. [1]. The authors do not require specialized hardware, but instead run a modified Mach kernel on a set of computer nodes that are connected with each other through a network [10]. Instead of running the backup process, the system periodically checkpoints the state of the primary process and sends the checkpoint to another node where the checkpoint can be resumed quickly.

Bressoud and Schneider tried implementing redundancy using hypervisors [25]. In their solution, there are two identical physical machines, each running a hypervisor with a single virtual machine. Both virtual machines run identical operating systems and software. They share the same physical storage unit connected to both physical machines. One of the machines is considered to be primary, while the second is the backup. Keeping two identical replicas of the same software without synchronization is impossible because of non-deterministic events, such as device interrupts. The authors solve this problem by masking all hardware interrupts from the operating system running on the backup system. Instead, the hypervisor running on the primary machine notifies the hypervisor on the backup machine about all interrupts it received from hardware. The hypervisor on the backup system sends these interrupts to its virtual machine, keeping the order and notifying the hypervisor on the primary computer when interrupt processing has completed. This ensures that the execution flow on both systems is exactly the same. Reading data from external devices such as a timer is handled in a similar manner by forwarding results from the primary to the backup node.

Performance measurements show that application execution speed on a cluster of primary and backup nodes is up to two times slower than the execution of the same application on a stand-alone computer of the same configuration. Also, both operating systems execute the same code in the same identical way. As a result, should the operating system fail, there is a high probability that both operating systems fail in the same way at the same time.

In other work, Bressoud moved the layer that synchronizes replica execution from the hypervisor to a middleware layer located between the operating system and the applications [24]. This approach has the benefit that the operating systems on both replicas function independently and have a much lower probability of failing at the same time. In order to make both computations identical, it is only necessary to eliminate sources of non-determinism. All events raised by the operating system, such as asynchronous exceptions or callbacks are intercepted and delivered to applications at the same points of execution. When loaded in memory, applications are binary instrumented with instructions that call the middleware layer at fixed points. At these fixed points, the middleware layer takes control of application execution and delivers any pending asynchronous events. The overhead of gzipping a file in this replication scheme is 25-50% compared to a non-replicated setup without any replication layer.

Another example of a system that uses virtual machines for fault tolerance without specialized hardware was developed by Cox et al. [40]. Two or more virtual machines are run on top of the Xen virtual machine monitor. Both virtual machines run the same operating system and execute the same software. The authors modified Xen so that it delivers the same interrupts in the same order to all virtual machines. Voting is done to ensure that all I/O results generated by the applications and guest operating systems are identical.

Replica synchronization using an isolation layer (hypervisor or middleware based) works well only on single processor systems. On multiprocessors, redundant execution of a threaded application may produce different outputs even if external inputs are duplicated identically between replicas due to the race conditions between threads that execute in parallel on different processors [40]. With today's desktop computers having multiple-core processors, this becomes a real issue. Pool et al. address this issue using *determinism hints* provided by programmers [99]. The determinism hint marks a section of code that produces results sensitive to the order in which different threads execute it. The hint

tells the system that all threads must execute this section sequentially and in the same order.

2.4.3 Memory Recovery

In the previous sections, we reviewed methods of recovering from an operating system crash by reproducing program state at a location not affected by the crash (e.g., another machine or a stable storage). This was done by either performing redundant computations or doing periodic checkpoints. The former is based on a repetition of all computations and has the disadvantage of a high equipment cost and performance degradation, while the latter requires continuous saving of all application data and has significant overhead. It is thus desirable to design a low overhead solution capable of restoring application state after a system crash without high run-time overhead; i.e., to allow the application state to survive an operating system crash. Application state usually consists of files located on persistent storage, data located at remote networked locations and data stored in the system's volatile memory. There is a large body of prior work on reliability and fault tolerance of file systems and distributed storage, which we do not report on here, since this is already widely covered [12, 32, 101, 134]. Instead, we present approaches here that concentrate on application state stored in volatile memory. While there is no generic software-only solution to this problem, some work has been done in this direction. Available solutions either utilize specialized hardware or try to solve some specific cases.

Some network cards allow direct *remote DMA* (RDMA) access to computer memory, bypassing the processor. Upon receiving a signal from another host, these cards are able to read a region of physical memory without any processor participation. Zhou et al. use such cards for system failure recovery [137]. Checkpoints are used to save the state of applications and the operating system in local memory. These checkpoints are either propagated to a remote node and saved in its volatile memory storage or remain

in the local memory and are retrieved later from it using RDMA in case of a crash. The authors found that without RDMA, checkpointing overhead was as high as 21%, while with RDMA checkpointing overhead was as low as 12% because the checkpoint did not have to be copied to the remote host. Disadvantages of the RDMA recovery include performance overhead, the need for special hardware, and a second node to store or retrieve checkpointed data.

Baker and Sullivan introduced the notion of a fixed sized, pinned region of memory called a *Recovery Box* accessible through a simple API [13]. Applications can save their state in this region. Memory of this region is not reinitialized during a soft reboot, so applications can restore their state after a system fault using the data from this region. Sultan et al. and Bohra et al. used a similar approach but used RDMA-enabled network cards to retrieve the saved application data [20, 122]. They used another computer to monitor a computer with an RDMA network card installed. The monitoring computer, using the RDMA card, detects when the monitored system hangs or crashes, extracts the saved state of the applications, and passes it to another computer, which then continues to execute the workload of the failed node.

While novel and useful, both of these approaches have several common shortcomings:

- Applications must be rewritten to save their state continuously and this introduces a constant overhead. In experiments with Recovery Box it was as high as 5%.

- Constant memory overhead is introduced. A region of memory is reserved for the saved application data and can not be swapped out to a disk. As a result, this solution effectively involves a trade-off between how much physical memory can be used directly by applications and how much data can be saved by these applications. In typical setups only a small amount of data can be saved (e.g., session identifiers or IP addresses). Although, the overhead for the examples provided by the authors is relatively small, in other applications, such as applications that have a large, constantly changing dataset, this overhead may be prohibitively large both from

processor and memory points of view.

- The RDMA-based approach requires special hardware and an additional computer system for monitoring. Security aspects of the RDMA still require investigation.

The idea of allowing application memory to survive an operating system crash was developed further by Chen et al. [33] and later work by Ng and Chen [93]. They targeted reliability of file system caches. Currently, two file system caching algorithms are used: write-back and write-through. The write-back algorithm is usually much faster, but less reliable, because in case of a system crash, all unsaved data is lost. Therefore, the write-through algorithm is much more prevalent.

Chen et al. suggest using a write-back cache and preserving its contents during a system reboot in the case of a system crash. The authors claim that it is possible to design the write-back cache as reliable as the write-through cache. The authors designed a caching subsystem called *Rio*. To ensure that the file cache is not damaged by operating system faults, Rio protects caching pages from unauthorized writes and uses checksums. Immediately after reboot, at the early stages of system start-up, the entire physical memory image is saved onto disk. Once the system completes its reboot, the memory dump is analyzed, the unsaved data from the cache is extracted and saved to disk. In order to assist in the memory dump analysis, the kernel does additional book-keeping of memory pages that contain unsaved cached data. After taking all measures to protect the file cache, reliability of Rio was even higher than that of the standard caching mechanism of FreeBSD and DigitalUnix. (The reliability became better partially because the authors modified the operating system to detect deadlocks and infinite loops inside the kernel.)

Lowell and Chen suggest using the Rio file cache as a foundation for a recoverable memory library [76]. They used the recoverable memory to store the transaction log of a database instead of using persistent storage. This improves performance of TPC-B and TCP-C benchmarks 150-500 times. Since this approach uses the Rio file cache, the size of recoverable memory available is limited by the physical memory size and memory

directly used by applications. A slower, but less memory consuming, implementation of recoverable memory was done by Satyanarayanan et al. [111].

In subsequent work, Lowell and Chen developed a fast application checkpointing system called *Discount Checking* [77]. The authors noted that checkpointing is similar to memory transactions in the way that both make a snapshot of application memory. The authors suggest that in order to do a checkpoint, process memory space must be mapped to transactional memory and, when a checkpoint is required, a new transaction is started. Checkpointing applications from the SPEC95 benchmark every second introduced overhead of only 10%. Since Discount Checking uses Rio Vista, it can work only with applications that fit completely in physical memory, thus doubling application memory requirements.

Ng and Chen describe an attempt to the use the Rio file cache and Rio-Vista for databases, effectively using memory to store database buffers, and then assess the reliability of this approach [94]. They compared three methods: (i) using the Rio file cache without changes to the database engine, (ii) using Rio-Vista by directly mapping regions of reliable memory to the database application space, (iii) a modification of the previous method that uses memory protection to prevent accidental writes in order to improve reliability. The authors found that all three designs achieved approximately the same degree of reliability when using fault injection. Only 2.3%-2.7% of the faults resulted in corruption of persistent application data. This is comparable with the reliability of commercial databases (IMS and DB2), which corrupt data in 1.8-2.1% of the crashes [121].

An interesting idea to significantly reduce the time required to reboot Linux is described by Biederman [19]. This project is called *KExec*. When the operating system kernel receives a command to reboot the system, instead of calling BIOS, it loads a new, uninitialized kernel image into memory and directly passes control to the initialization routine of the new kernel. Thus, the hardware initialization is skipped, and the BIOS and the boot-loader are not involved.

Later work by Goyal et al., called *KDump*, is based on KExec and provides a reliable and convenient crash dump facility for Linux [53]. During the system startup, a region of physical memory is reserved and is not used by the operating system. The image of another special-purpose, lightweight kernel, called *crash kernel*, is loaded into this region and stays there uninitialized. Upon a crash of the running kernel, control is passed to the initialization routine of the crash kernel. The crash kernel initializes itself using only the reserved region of physical memory and leaving the the rest of the physical memory unchanged. After the initialization of the crash kernel is complete, the crash kernel saves the contents of the whole physical memory to disk, so that it can be studied later. In contrast, previous memory dump solutions relied on the crashed kernel code and drivers to do the work, but this is unreliable, since the kernel has already experienced an unrecoverable error.

KDump is used only for obtaining memory dumps after a system crash. However, in subsequent chapters, we will show that the second, undamaged kernel in memory can make operating system failure recovery much more robust than it is now. Having the second uncompromised kernel in memory, ready to take over control, allows us not only to analyze the data of the main kernel but also to recover data from the main kernel.

2.4.4 Failure Oblivious Computing

Most system designers attempt to follow the fail-stop principle in that they try to detect a failure as soon as possible and either stop processing immediately or analyze the cause of the error and fix its consequences. But a few authors propose tolerating failures as long as possible. They argue that many errors do not affect execution results and that ignoring such errors can result in successful execution of applications with high probability. While this approach is inappropriate for mission-critical systems that require guaranteed validity of results, for many other applications used by ordinary users, this may be an effective solution. For example, users usually would not care if 1 second of their music is skipped,

if the only alternative is a crash of the music playing application. If a received email message can crash a mail client, then the user may not object to having this email be displayed incorrectly, since an email client crash renders all messages unavailable. Web sites would benefit from scenarios whereby some requests are dropped or incorrectly responded to if the alternative would be a failure of the entire server.

Rinard et al. were the first to introduce the term *Failure-oblivious computing* [109]. The usual practice is to terminate an application with an error message when an invalid memory access is detected. Rinard et al. suggest that invalid writes should be discarded and if the application tries to read from an invalid location, specially crafted, fake data is to be returned. The authors tested their approach on a number of programs, including mail clients, file managers, as well as web and mail servers. The authors reintroduced known memory-related bugs, and, in all tested cases, applications did not crash and continued to produce satisfactory outputs for most user requests.

Qin et al. noticed that many bugs are triggered only under specific, rarely occurring conditions caused by the application environment [102]. They developed a system called *Rx* that creates periodic lightweight checkpoints during application execution. In the case of a failure, execution of the application is terminated, the application is restored from the latest checkpoint, and an attempt is made to change the application environment, e.g., memory allocation can be done at a different address or of a greater size. Rx intercepts all requests to the application, and, in the case of a crash and subsequent rollback to the checkpoint, ensures that the application receives all the requests that it had received after the most recent checkpoint.

DieHard is a randomized memory manager for type unsafe languages [17]. The primary purpose of this memory manager is to allocate memory so that the probability of memory corruption is minimized. This is achieved by making the heap size a multiple of the size requested and placing objects randomly within the heap. When compared to the standard Linux allocator, DieHard introduced processor overhead of 8%, and memory

overhead of more than 100%. Instead of trying to detect faults, DieHard tries to mitigate their harmful consequences and allow a faulty application to finish its execution without interruption.

2.4.5 Recovery in Production Operating Systems

Failure recovery methods that have found their way into production operating systems are primitive at best; modern operating systems, such as Unix, Linux, and Windows, do not do much except to detect a fault with the help of hardware and then reboot the system.

History of failure recovery in operating systems is described by Auslander [6]. Early computer systems were able to detect hardware faults and provided no methods for failure recovery. With improvements in hardware reliability, software defects including defects in operating system code started to play a more significant role.

IBM was one of the first companies to develop an operating system capable of recovering from errors. In order to detect and survive software and hardware errors, IBM's MVS operating system was designed to include recovery procedures that try to assess damage and to recover or, in the worst case, remove ongoing work from the system [6]. Software modules can define recovery routines that are called if a given software module crashes. In the case where software does not provide a recovery routine, the recovery routine provided by software at a lower level is called. As shown by Sullivan and Chillarege, this approach did not work in many cases: 6.3% of the errors encountered within the operating system caused a reboot and 12.6% caused a system outage [120].

A similar approach was taken with the Multics operating system. In the end, half of the code written for Multics was for error recovery [129]. In order to avoid such complexity, developers of the UNIX operating system, according to Tom Van Vleck, removed all kernel error recovery code and replaced it with calls to a *panic()* routine that halted and then rebooted the system [129].

Current versions of the Linux and Windows kernels use a similar approach. Whenever an unexpected situation is discovered, the kernel calls a special routine (*panic()* in case of Linux or *KeBugCheckEx()* in case of Windows) that prints debug information to the screen and reboots the system.

Most modern production operating systems rely on a combination of hardware and software asserts to detect faults and try to protect applications and the kernel from them. Hardware can assist in protecting an operating system kernel and applications from each other by providing *protection rings* as described, for example, by Schroeder and Saltzer[113]. Process memory space is divided into domains called rings. Each ring has a level associated with it. When the processor executes code stored in a memory associated to ring at level n, it cannot read or modify data or branch to code stored in a segment associated to a ring at level m>n. Currently, most operating system use only 2 rings: kernel code is stored in a privileged ring and application code is stored in a non-privileged ring.

2.5 Summary

In this chapter we argued that large complex software, such as operating systems, will never be free of bugs. Despite (or maybe partially because of) many years of development of commodity operating systems, they still crash several times a year and the evidence shows that this situation is unlikely to change in the near future. Operating system failures are the most severe type of software failure because with today's systems they result in application data being lost. However, in the vast majority of operating system crashes, application memory remains unaffected and contains correct data. This makes it theoretically possible to continue running applications after an operating system crash if the failed operating system kernel can be rebooted without destroying the state of running applications. Contemporary operating systems, however, do not have this capa-

bility. Otherworld, which we present in subsequent chapters, is a technique that recovers application state from a crashed kernel so that applications can survive operating system crashes.

Currently, there is no single perfect approach for recovering from operating system failure. There are a number of different techniques that try to prevent or minimize data loss, each with their own strong and weak points. In Table 2.1 we summarize those techniques, comparing them with Otherworld using metrics, such as processor and memory overhead, extra hardware cost, types of operating system failures those techniques protect from, extent of the application data loss after the recovery, and amount of required changes in the kernel (including drivers) code and applications.

Technique	Overhead	Extra hardware costs	Failure types addressed	Application data loss	Changes to the software
CuriOS [42]	Some (no data available)	None	Faults in base services and drivers	None	Rewriting kernel and drivers
Drivers in VM [49, 71]	High	None	Faults in drivers	None	Kernel modifications
Type-safe languages [18, 58, 118]	Low	None	Invalid memory accesses in the kernel and drivers	None	Rewriting kernel and drivers
Compile-time instrumentation [114, 116, 117, 132]	High	None	Invalid memory accesses in the kernel and drivers	None	No
Palladium [36]	Low	None	Invalid memory accesses in drivers	None	Kernel and driver modifications
Nooks and Shadow drivers [123, 124, 125]	High	None	Faults in drivers	None	Kernel modifications
Recovery Domains [70]	High	None	Faults in the kernel and drivers	None	Kernel and driver modifications
Checkpointing	High (depending on frequency and workload)	None	Faults in the kernel and drivers	Some (depending on frequency)	Some require kernel and/or application modifications
Redundant calculations [15, 22, 25, 40, 41, 96, 99, 103]	None or low	High	Hardware faults and/or faults in the kernel and drivers	None	Rewriting or modifying the kernel
Recovery box [13]	Low (depending on application)	None	Faults in the kernel and drivers	Some (depending on application)	Kernel and applications modifications
RDMA network cards [20, 122, 137]	Low (depending on application)	Low	Faults in the kernel and drivers	Some (depending on application)	Kernel and applications modifications
In-memory checkpointing [33, 76, 77]	High memory overhead	None	Faults in the kernel and drivers	Low	Kernel modifications
Otherworld	**Negligible or low (depending on reliability level required)**	**None**	**Faults in the kernel and drivers**	**None**	**Kernel modifications, may require minor application modifications.**

Table 2.1: Comparison of recovery techniques.

Chapter 3

Otherworld Design: Kernel Microreboot

3.1 Overview

Currently, most production operating systems rely on a full system reboot as the only way to recover from an unexpected failure within the operating system kernel. This approach is effective because there is a lot of evidence that most failures are caused by intermittent, transient bugs [54, 86, 85, 102]. Bugs that manifest themselves consistently are easier to reproduce and are typically fixed during the testing stage of operating system development. In addition, a reboot is arguably the best way to deal with system aging problems including resource leaks. The effectiveness of a reboot is ensured by its simplicity and the fact that subsequent system recovery occurs under control of freshly initialized software and does not rely on the correct functioning of the code that experienced the failure.

A full and immediate system reboot on a system failure is costly because it results in system downtime and application data loss. In order to address this problem, Candea et al. suggest the concept of a *microreboot*, the individual rebooting of fine grained components [28]. In order to support a microreboot, a component, according to Candea

et al., must be *(i)* well isolated from other components and *(ii)* be stateless.

In this dissertation, we propose extending the concept of microrebooting to the operating system kernel and treat the kernel as a single component, the only one that can and should be microrebooted in response to its internal failure. There is a third pre-requisite for a microreboot, which was implied but not explicitly mentioned by Candea et al.: *(iii)* there should be a software component that can manage the microreboot process. If the microreboot was triggered by a failure, the component that performs the microreboot process should not be affected by the failure.

Operating system kernel code naturally satisfies the first condition of the microreboot: it is well isolated from the rest of the software running on the same system. At the physical level, it is isolated with hardware privilege protection rings. At the logical level, other software components communicate with the kernel only through a well defined, strictly enforced system call interface.

Unfortunately, existing operating system kernels fall short of being stateless. Also, the operating system kernel resides in a privileged layer underneath all applications, so there is no other software component that can manage a kernel microreboot without destroying all applications running above the kernel. In order to deal with the kernel state, our kernel microreboot mechanism does not destroy the state of the failed kernel, but preserves it in memory during the microreboot and extracts it after the microreboot is complete. Although kernel state tends to be complex and may include hardware state, the interface between the kernel and other parts of the software system tends to be relatively simple and conceals most of the kernel complexity. In order to perform a kernel microreboot, we only need to preserve the part of the kernel state visible from outside of the operating system kernel, e.g., the part of the state visible to running applications or other computer systems on a network.

The third pre-requisite, a software component that can manage the microreboot process, also presents some difficulties. The kernel resides in a privileged layer underneath

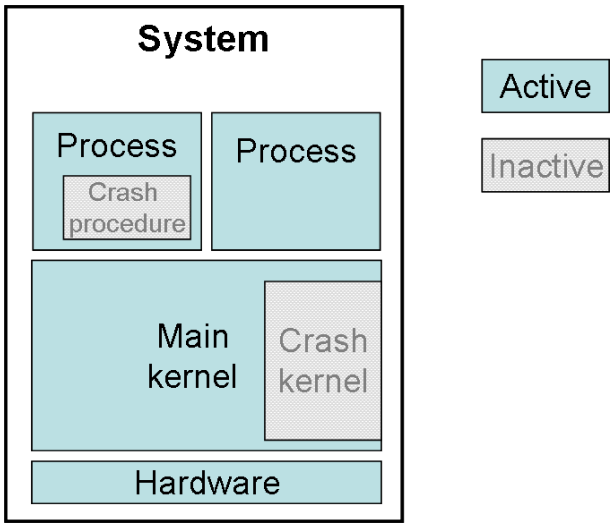

Figure 3.1: Main and crash kernels

all applications. The system BIOS code is not aware of running applications, so it cannot manage a kernel microreboot without destroying all applications running on top of the kernel. Applications, running on top of the kernel, lack knowledge of how to manage hardware resources and do not have sufficient privileges.

We have designed an operating system kernel microreboot mechanism called "Otherworld" that addresses the issues mentioned above. In order to perform a kernel microreboot, we propose having two operating system kernels resident in physical memory. The first, (*main*) kernel, performs all activities an operating system is responsible for. The second, (*crash*) kernel, is passive and is activated only when microreboot is required, e.g., the main kernel experiences a critical failure (Fig. 3.1). When the main kernel boots, it reserves a region of physical memory for the crash kernel, loads the crash kernel into that memory region, and then protects the region from being modified using standard memory protection hardware. From that point on, the main kernel continue execution as normal, while the crash kernel remains passive until it is passed control.

When the main kernel crashes (i.e., "panics"), instead of rebooting, it passes control to the crash kernel. The crash kernel is not affected by the error because it has been passive

and hence has not yet accumulated any state, and because it was protected by standard memory protection hardware. After obtaining control, the crash kernel initializes itself using only the limited region of memory reserved for this purpose by the main kernel. All kernel state related to running applications as well as the application data still exists in memory and is accessible to the crash kernel. This allows the crash kernel to reconstruct the state of each application and continue application execution without losing data. We refer to this reconstruction process as application *resurrection*.

An application can optionally register a special user-level function, called *crash procedure*. This function is called by the crash kernel notifying the application that a kernel microreboot has occurred and that the application has been resurrected. The crash procedure is called in a way similar to the way application exception (signal) handlers are called. The crash procedure can optimally:

- verify consistency and correctness of application state,
- restore relevant parts of the system state that are not automatically restored by the crash kernel, and/or
- save application state to persistent storage,

and then decide whether it should

- instruct the crash kernel to continue application execution from the point at which execution was interrupted by the kernel failure,
- restart the application, or
- it can decide not to continue execution (after perhaps having stored critical application state to disk).

For example, the crash procedure may reestablish a network connection, if the crash kernel was unable to do so, before requesting the crash kernel to continue its execution.

Otherworld consists of three parts. First, there is a modified operating system that microreboots itself without destroying running applications. Second, there is code inside the crash kernel that extracts the main kernel state and resurrects application processes after the microreboot. Third, for each application there is an optional *crash procedure*, an application-defined, user-level function called by the crash kernel after the resurrection.

The microreboot process is composed of 5 distinct stages

1. At first boot, the system configures itself as the main kernel, loads the crash kernel image into a region of physical memory reserved for the crash kernel, and protects that region from accidental writes.

2. The system runs normally under the control of the main kernel. At any time, applications may register a crash procedure with the kernel.

3. On a kernel failure, control is passed to the crash kernel, and the crash kernel initializes itself within the memory reserved for this purpose.

4. After initialization, the crash kernel resurrects applications and calls their crash procedures, if registered.

5. The crash kernel takes control over all remaining system resources, morphs itself into the main kernel, and installs a new crash kernel.

We implemented our mechanism in Linux, but our architecture is generic, and we believe it can be applied to other operating systems as well. Below, we describe each of the stages in more detail but leave Linux-specific implementation details for the next chapter.

3.2 Setup and Normal Execution

Initially, the computer system is booted normally by loading and initializing the main operating system kernel. In order for the system to be capable of performing a microre-

boot, the main kernel reserves a special region of physical memory, which is large enough for the crash kernel to boot itself. The crash kernel image is loaded into this region and is left there untouched and uninitialized, protected by memory hardware. As long as the main kernel operates without a failure, the crash kernel image is left intact in this region of physical memory, and its code is never executed.

As we will describe later, we use different swap partitions for the crash and the main kernel. Therefore during the configuration stage, the administrator needs to create two identical swap partitions: one to be used by the main kernel, the other by the crash kernel.

Any user process that wishes to be notified after a kernel microreboot can register a crash procedure with the main kernel. The address of this procedure is stored in the process descriptor maintained by the main kernel and serves as an entry point to be called by the crash kernel after it has resurrected the user process.

In order to simplify resurrection and reduce the number of main kernel data structures that we have to retrieve and rely on, we modified the main kernel so that the state necessary to recreate the resources belonging to the applications is easier to access. For example, in order to recreate an open file, only the file location, name, open flags, and current file offset are required. Thus, we modified the file descriptor to also store the location, name, and open flags specified by the application during the *open* call in addition to the file offset it is already storing. As a result, we only need to rely on one structure to recreate the kernel open file state.

In our current implementation, we use the same kernel source to build both the main kernel and the crash kernel. Common source code for both kernels has advantages and disadvantages. One advantage is that it is easier for the crash kernel to access the main kernel data structures since they both use the same structure layout. In addition, modifications to one of the kernels are automatically applied to the other, as both kernels are built simultaneously.

On the other hand, the crash kernel could be different from the main kernel. Although this approach is more complex to implement, it has one important advantage: if the kernel fault that triggered a failure is not intermittent (e.g., was caused by some particular combination of system call arguments) resurrection of the application that triggered the fault could cause the same fault to be triggered again, since the application will retry the system calls when run under the crash kernel. Using different kernel versions would allow us to successfully recover from this situation.

One of the key benefits of Otherworld is negligible run-time overhead. Because the crash kernel is passive, there is no processor overhead associated with it. [1] As we will show in the next chapter, physical memory overhead constitutes only 2% of the memory size of the typical desktop computer. No additional disk or network I/O is generated by Otherworld.

3.3 Response to Kernel Failure

When an unexpected failure happens when executing operating system code, the processor that was executing the code was in kernel mode; the other processors may have been executing either user or kernel mode at the time. Today's operating systems, such as Linux or Windows, respond to such a failure by halting all processors except the one that triggered the failure. This remaining processor executes the code that prints out an error message and either halts execution or jumps to the BIOS reboot code.

Otherworld responds to this situation differently. When the main kernel experiences a critical error, instead of rebooting, it issues non-maskable interrupts to all processors except the one that executed the code that triggered the failure. Upon receiving a non-maskable interrupt, each processor interrupts its execution, saves the current thread context on the stack, and jumps to the kernel's interrupt service routine. This service

[1] An optional application state protection mechanism, which we will discuss in Chapter 5, does incur some overhead.

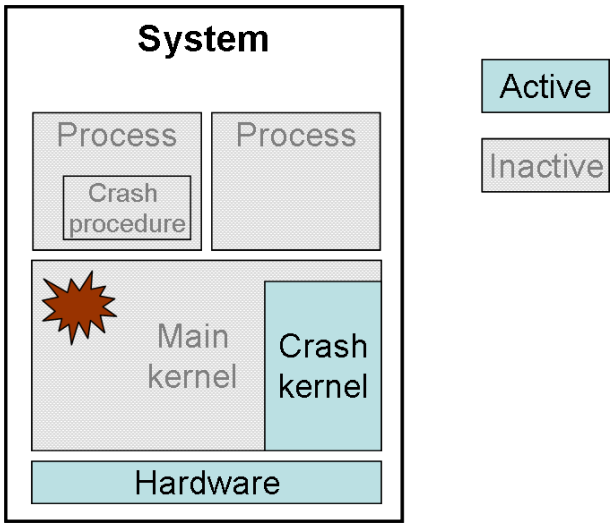

Figure 3.2: Failure is detected in the main kernel

routine sets a global per-processor flag acknowledging the interrupt and brings the corresponding processor to a halt. The processor that issued the interrupts waits for all other processors to halt before proceeding to the next step. This ensures that the context of all threads is saved on the thread stack for all user threads, so that the crash kernel can later retrieve this context and continue user mode thread execution.

The next step is to remove the memory protection from the crash kernel image and jump to the initialization point of the crash kernel. From this point on, main kernel code is no longer executed, and the crash kernel controls the system. The crash kernel initialization does not rely on the state of the main kernel (which may be corrupted), but initializes itself using only data contained in the BIOS, hardware, and stored on disk. Because of this, the crash kernel initialization is as reliable as a regular kernel initialization after a full reboot. The weak link in this design is the code that executes in the context of the failed main kernel between when the failure is detected and when the crash kernel initialization procedure is called. Although, we attempted to make the implementation of this part as small and simple as possible, as we will show in our evaluation, it remains by far the largest source of microreboot failures.

The state of the main kernel may be stored in physical memory, the swap partition on disk, the file system, and remote network-accessible devices. The crash kernel must initialize itself so that it does not modify any state of the main kernel necessary for application resurrection. In order to do this, the crash kernel initializes itself the way the main kernel does with several exceptions. First, it only uses the physical memory region originally reserved by the main kernel for this purpose (Fig. 3.2).

Second, in order to not corrupt any pages 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. Before choosing which partition to use, start-up scripts[2] query the kernel to determine if it is the main or the crash kernel. Based on which kernel is booting, the start-up scripts choose the appropriate partition. In our implementation, we did not encounter any kernel state required for resurrection to be stored on local disk or remotely. In fact, operating system developers are explicitly discouraged from doing this [66]. However, it is possible over time that more and more complex code, such as network file systems, antivirus systems, web servers, and/or virtual machine managers, that access state in the file system or remote servers, are moved into the kernel. In this case, the kernel start-up scripts must detect when the crash kernel is running and preserve the main kernel state stored on disk before the crash kernel tries to modify it (similar to the way we switch swap partitions).

Apart from the differences described in the previous paragraph, the crash kernel and the main kernel initialization process is exactly the same. They share the same start-up scripts, load the same device drivers, and mount the same file systems at the same mount points. As a result, the application environment of the crash kernel is the same as that of the main kernel, which makes kernel microreboots more transparent to applications and simplifies the resurrections.

[2]In most Unix implementations, after the kernel finishes its internal initialization, it starts the first user-space process, usually called *init*. The *init* process finishes the system initialization by calling various start-up scripts provided by the operating system vendor and system administrator.

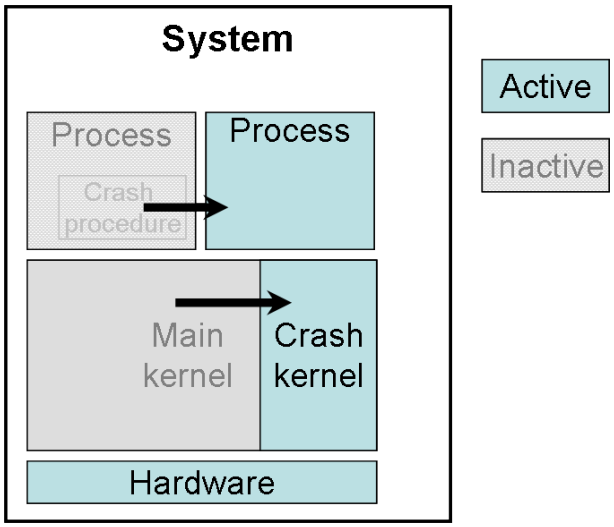

Figure 3.3: Crash kernel retrieves information from the main kernel

In our implementation, we did not modify any drivers, and all drivers initialize the devices the same way during crash kernel initialization as during the initial system boot. More robust device drivers or loadable kernel modules may maintain state visible to applications and may want to preserve the state across a kernel microreboot. Also, a driver developer may want to use, as an optimization, different logic to re-initialize a device after a crash occurred. For such cases, we envision the crash kernel providing an API to drivers to allow them to extract their state from the main kernel and the possibility for the driver to ask the kernel if it is a main or a crash kernel. If a driver, in its initialization function, determines that a microreboot has just occurred, it can access the memory of the main kernel through the functions provided by the crash kernel and extract its own state as it was before the microreboot. However, we have not tested this functionality with real drivers.

3.4 Application Resurrection

After the crash kernel completes its initialization, it starts a recovery phase, during which it accesses main kernel data structures and application memory pages in order to resurrect applications (Fig. 3.3).

Since most of the kernel functionality can be called from an arbitrary context, e.g., when serving an interrupt or an application system call, most of the kernel state also has to be accessible from an arbitrary context. For this, the kernel state is organized in tree-like structures; e.g., the process descriptor contains a reference to the list of files opened by this process, each open file contains references to in-memory buffers that contain this file's data. The root references to those structures are usually kept in global variables at well known locations. The important assumption that we make is that all application state that the kernel maintains can be retrieved by parsing such trees. This assumption is true for Linux, and we believe that it has to be true for most other operating systems.

The crash kernel recovery code does not have the goal of restoring the full operating system kernel state, as it was at the time of the crash. In fact, this would be undesirable because it would just increase the probability that the same failure will occur again.[3] Rather, our objective is to reproduce only the part of the kernel state that is visible to applications. And this part is only a small proportion of the entire state maintained by the kernel. To illustrate this, consider the Linux kernel. From an application point of view, the memory available to an application (if it is not mapped to a file) is described by a list of memory regions with the following information:

- Starting address
- Length
- Protection flags

[3] In the extreme case of reproducing the kernel state exactly as it was at the time of the crash, the system would experience exactly the same failure again immediately after restoring the system state.

- List of physical pages that contain the actual data

However, the Linux kernel maintains much more state for managing memory:

- Each memory region descriptor, besides containing starting address, length, and protection information for each region, also contains more than 20 additional fields not visible from user space.
- Memory regions mentioned above, but organized in a red-black tree structure [23].
- List of the memory regions recently accessed by applications.
- Page descriptor structures for each physical memory page in the system.
- Hardware page tables that provide virtual to physical address mapping.
- Reverse page mapping structures that map every physical memory page to all memory region descriptors, which this page belongs to.
- Zone descriptor structures that contains several lists of page descriptors (LRU page lists, free page list, per-cpu page lists), and more than 20 fields with different statistics.
- Slab allocators which contain lists of free chunks of memory organized by frequently used sizes.
- Memory pool consisting of a list of memory pages to be used in low-on-memory emergencies.
- Other data structures not mentioned here.

A portion of the state maintained in the memory region descriptors and the hardware page tables is sufficient to recreate the application virtual memory space not mapped to disk. The situation is similar with the other resources managed by the kernel. For

example, for open files, only a small and relatively simple subset of the kernel state needs to be extracted from the main kernel in order to be able resurrect the files successfully. This allows us to recreate application-visible kernel state from a relatively small amount of main kernel state.

3.4.1 Restoring Application Resources

By knowing the location of the list of process descriptors in the main kernel, the crash kernel can retrieve the process descriptor of each individual process that was running at the time of the kernel crash. For each process that is to be resurrected, the crash kernel creates a new process. Attributes of the process visible to the application, such as signal handler descriptors, signal masks, and process priority are used to initialize the newly created process. Not all of those attributes are stored in the process descriptor itself, but they can all be located using references to other data structures available through the process descriptor. Process descriptor attributes that are not visible to the application, e.g., statistical information used by the CPU scheduler, are recreated from scratch.

The kernel portion of the virtual address space of the newly created process is the same as for any other process running on the crash kernel. The user portion of the virtual memory space of the newly created process is created to be a copy of the user portion of virtual address space of the process being resurrected. To recreate the user virtual memory space, the crash kernel obtains the boundaries and protection flags for all of the memory regions of the process being resurrected from the main kernel memory. For each memory region belonging to the process being resurrected, the crash kernel creates a new memory region in the newly created process with the same attributes. Some of the memory regions may have been mapped to a disk file. Names of those files and mapped section locations are extracted from the main kernel memory, and the resurrection code creates a new mapping with the same parameters.

Some of the processes' pages being resurrected may be shared with other processes.

The sharing can occur either by two or more processes mapping the same section of a disk file to their virtual address spaces or by using the Linux IPC shared memory mechanism. The latter reuses the generic file mapping mechanism, but instead of mapping a memory region to a regular disk file, it maps the shared pages to a pseudo-file in a special shared memory file system called *shm*. The current implementation of Otherworld can restore shared memory mappings that are based on regular disk files by recreating file mapping with the same parameters. Resurrection of shm-based shared memory is not implemented yet, however, the implementation should be straightforward because references to all shm files descriptors are stored in a single structure pointed to by a global variable.

The next step is to retrieve the contents of each virtual memory page within the memory region. For each page, the crash kernel retrieves from the main kernel memory the corresponding entry of the hardware page table of the process being resurrected. If the entry references a physical memory page, a new page is allocated in the crash kernel and the content from the corresponding page of the main kernel is copied into it.[4] For each entry that corresponds to a page that was swapped out to disk by the main kernel, a new page is allocated in the crash kernel's swap partition (which, as we mentioned earlier, is different from the main kernel swap partition). The contents of the newly allocated page is copied from the corresponding page of the main kernel swap partition. This fully restores the user-level memory space of each target process.

After the application memory space of a process has been restored, the crash kernel restores the files that were open for the process. The crash kernel gets the list of descriptors of all files opened by the process, reads the name, location, open flags, and the current offset from each open file descriptor, and reopens the files accordingly. In order to make the reopening of the files transparent to the application, the crash kernel assigns the same file descriptor number that was used in the main kernel and restores the current

[4] As an optimization, one can directly map the physical page instead of copying it, which would significantly increase the speed of resurrecting large processes.

offsets. The last step of resurrecting an open file is the extraction of file's dirty buffers from the main kernel memory and flushing them to disk.

With respect to terminals, the current Otherworld implementation can only restore the state of physical terminals. In order to do this, the crash kernel checks the type of the terminal attached to the process being resurrected, and if it is a physical terminal, its state is restored as a part of the application resurrection process. The crash kernel attaches the current user's physical terminal to the newly created process, retrieves the terminal settings from the terminal descriptor that the process being resurrected used when the main kernel failed, and initializes the current terminal with these settings and the screen contents of the terminal of the process being resurrected.

The process descriptor references signal handler descriptors, that contain pointers to the application-defined signal-handling functions. Since the user portion of the virtual address space of the newly created process is created to be the exact copy of the process being resurrected, these pointers can be directly copied from the process being resurrected to the newly created process. Bitmasks of pending and blocked signals are copied as well. We also copy the pointer to the crash procedure (which we will describe in detail later in this chapter).

Resurrection of network-related resources is much more complex for several reasons. First, the resurrection code does not have direct access to or any control over remote hosts. Thus, the crash and subsequent resurrection have to be done completely transparent to the remote host. Second, the network code arguably composes the largest and most complex kernel subsystem. For example, the source code size for the TCP/IP protocol implementation inside the Linux kernel measured by the number of lines of code is only 25% smaller than the sizes of process, memory, and generic file management subsystems combined. Third, although nearly all network protocols in Linux use network sockets as an interface with user-mode applications, the data associated with each socket differs, depending on the protocol for which the socket was opened. As a result, the resurrection

of sockets has to be protocol dependent.

However, network communications are inherently unreliable because network communication failures occur frequently. This has to be expected and dealt with by network protocols, applications, or, more frequently, both. Most applications use TCP over IP or UDP over IP for communications over the network. Because IP and UDP protocols are unreliable, i.e., they do not guarantee packet delivery, it is safe to discard any IP and UDP data packets the main kernel was processing at the time of the crash. Only connection parameters associated with a socket need to be resurrected. These parameters include source and destination IP addresses, socket options, and others. For TCP, many more connection parameters need to be resurrected, including source and target ports and the current sequence numbers of the packets sent and received. Moreover, all outbound data packets as well as acknowledged inbound data packets need to be recovered. Also, the resurrection time is an important factor and has to be smaller than the various TCP timeout intervals.

In order to demonstrate the resurrection of network resources used by a process, our current crash kernel implementation fully resurrects network sockets that use raw IP, ICMP, or TCP protocols. Implementation details of TCP socket resurrection will be covered in the next chapter. We do not expect socket resurrection for other network protocols to be significantly different from resurrection of the TCP/IP protocols.

We have not yet implemented the resurrection of the various IPC resources, such as UNIX domain sockets, System V semaphores, pipes, or pseudo terminals. Hence, at this time, application crash procedures have to be added to programs that use these resource types in order to restore them in an application-specific manner or in order to at least shutdown the application gracefully after having saved the application state to persistent storage. But, as we will show in the Application Case Studies chapter, even our limited prototype is applicable to a wide range of applications.

	Crash procedure defined	**No crash procedure defined**
All resources were resurrected	The crash procedure will be called. It can either save data to disk and restart the process or instruct the crash kernel to continue the execution of the process.	The crash kernel will continue the execution of the process.
Some resources could not be resurrected	The crash procedure will be called. The crash procedure can either restore resources itself and continue execution or save application state and restart the process.	The resurrection will fail.

Table 3.1: Interactions between the crash kernel and the application being resurrected.

3.4.2 Crash procedure

After the resurrection of kernel state used by a process is complete, the crash kernel is ready to start executing the resurrected process. Process execution begins either by calling the registered crash procedure of the application or by simply resuming the application threads if the application did not register a crash procedure and the crash kernel was able to resurrect all of the application resources. This is summarized in Table 3.1. If a process did not register a crash procedure with the main kernel, and some kernel resource associated with the process could not be resurrected automatically (either due to a main kernel data corruption or due to implementation restrictions of the crash kernel), process resurrection will fail. Whenever a crash procedure for a process is registered, the crash kernel calls it and lets it decide the further course of action. The crash procedure

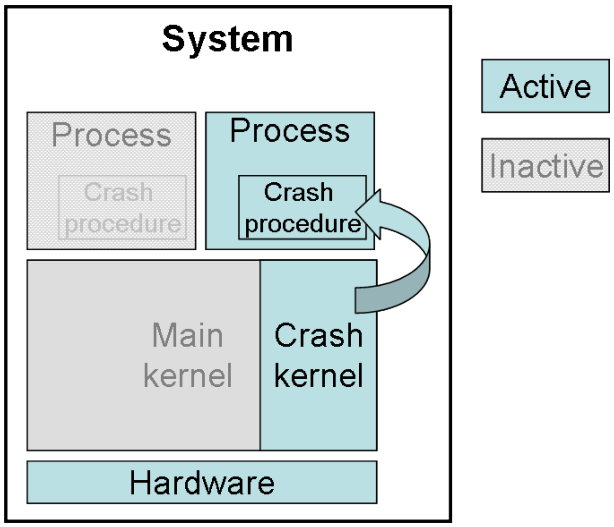

Figure 3.4: Crash kernel retrieves information from the main kernel

can use application-specific logic to resurrect resources that were not resurrected by the crash kernel and instruct the crash kernel to continue process execution, or it can choose to save the application state and restart the process. We provide more details on crash procedures in the next subsection.

The crash procedure provides the resurrected process the opportunity to execute recovery code with all of the process's global data available (Fig. 3.4). The declaration of a crash procedure is as follows:

```
int ow_crash_proc(unsigned long failed_types);
```

Each bit in the *failed_types* parameter corresponds to a resource type, for example, bit 0 corresponds to open files, bit 1 corresponds to sockets, etc. If any of the kernel resources used by the process being resurrected cannot be resurrected by the crash kernel, then the bit that corresponds to the type of the resource is set to 1. If all resources of a certain type were resurrected successfully by the crash kernel, the bit that corresponds to this resource type is set to 0.

If the crash procedure returns zero, the crash kernel will continue process execution from the point at which it was interrupted by the kernel microreboot. If the return value

is non-zero, the crash kernel will terminate the process.

The crash procedure can issue the *exec()* system call to replace the currently executing program with a new one. This is typically used when the crash procedure chooses to restart the process after having saved application state to the persistent storage.

Crash procedure registration typically happens when the application starts up. The registration of a crash procedure is done through a new *otherworld()* system call that we introduced for communication with Otherworld. We describe this system call in more detail in Section 3.6. In order for an application to register a crash procedure with the main kernel, it must fill and pass to the main kernel the following structure:

```
struct ow_crash_params
{
    int (*ow_crash_proc)(unsigned long failed_types);
    unsigned long* stack;
    size_t stack_size;
};
```

The *ow_crash_proc* field provides the address of the crash procedure. When the crash procedure is called it does not reuse an application thread stack, but rather uses its own. This stack must be allocated by the application at crash procedure registration time, and its address is passed to the main kernel in the *stack* field. The size of the stack is fixed and passed in the *stack_size* field. We found that stack size of 64KB was enough for all applications with which we tested Otherworld. For complex crash procedures that do a lot of nested function calls the stack size may need to be increased. The crash procedure registration parameters are stored in the process descriptor but are not used by the main kernel. The crash kernel retrieves them from the main kernel memory during resurrection and uses them to call the crash procedure.

When the crash procedure is called, none of the process's threads paused by the microreboot are executing yet. To execute a crash procedure, the crash kernel uses the

process and memory descriptor of the process's main thread and the stack provided when the crash procedure was registered. The crash kernel schedules execution of the crash procedure as a normal application thread. Because the kernel microreboot will happen unexpectedly for a process, the crash procedure may be called at any time during process execution, and the only guarantee that the crash kernel provides is no other application thread will be running while the crash procedure is executing.

Crash procedures serve several purposes. First, they can be used to detect potential data corruption in an application-specific way. Since a kernel failure is an infrequent event, the application can afford to do elaborate data consistency checks. The problem of detecting data corruption is complex and interesting by itself, and we will leave it for future work.

Second, crash procedures are used for application-specific resource resurrection. The more resources the crash kernel attempts to resurrect, the more main kernel data it has to use, and the higher the probability of encountering data corruption. If data required for resurrection of some application resource is corrupted, the corresponding application resource cannot be resurrected. Also, some resource types can not be resurrected due to implementation limitations, as was outlined in Section 3.4.1. An advanced crash procedure can resurrect these resources using application-specific logic, for example reopening network connections.

Finally, the crash procedure can be used to allow the application to decide whether it wishes to continue executing as is (e.g., if all resources were successfully resurrected), whether it wishes to save application state to persistent storage and restart, or whether it simply wishes to exit.

If a crash procedure was not registered for an application then the application can be resurrected only if the crash kernel succeeds in resurrecting all of the application's resources. On the other hand, if the application registered a crash procedure then the crash procedure is called after the microreboot even if some resources could not be res-

urrected. Thus, while many applications can be resurrected without a crash procedure, having a crash procedure that saves application state to persistent storage and restarts the application, significantly decreases the probability of the application data corruption due to the kernel failure.

Also, as shown by Chandra and Chen, application-specific recovery is much less likely to be affected by faults within the operating system than application generic recovery [29]. In many cases (e.g., for non-critical interactive applications) continuing the execution so as to minimize inconvenience to the user at the expense of a slightly higher risk of data corruption is acceptable. For more mission critical applications, such as databases, one may prefer to always save application state to persistent storage and restart the application to eliminate all potential side-effects of the fault.

3.4.3 Resuming Application Execution

Referring to Table 3.1, if a process did not register a crash procedure, or if a crash procedure was registered and called, and it decided to continue process execution, the crash kernel will continue process execution from the point at which the process was interrupted by the kernel microreboot. If no crash procedure was registered before the crash, and the crash kernel was not able to resurrect all of the kernel resources consumed by the application, the resurrection will fail and the process is terminated.

As discussed in Section 3.3, the last thing the main kernel does before switching to the crash kernel is to issue non-maskable interrupts to all processors in the system, except the one that was executing the microreboot code. After the non-maskable interrupts are received on all processors, every thread in the system is either:

- Not scheduled (sleeping or ready to run). In this case, its user thread context has been saved previously by the scheduler code.

- Running in the kernel (serving a system call or an interrupt). In this case, its

user thread context has been saved when the switch to kernel mode occurred. As described in the next subsection, the thread will have to reissue the system call.

- Running in user mode. In this case, the user thread context has been saved while switching to kernel mode to serve the non-maskable interrupt.

This ensures that the user thread context of all threads is saved on their stack, and that the crash kernel can continue execution of each application thread after the microreboot.

As discussed in Section 3.4, because the primary goal of a kernel microreboot is to recover from faults inside the operating system kernel, we do not attempt to recreate the kernel state exactly as it was at the time of the failure. Instead, we try to preserve the application state and recreate the kernel state from scratch so that the applications can potentially continue their execution after the microreboot.

3.4.4 Reissuing system calls

If, at the time of a crash, a process was in the middle of a system call, then the crash kernel will cause the system call to return an error code that signals to the process that the call was aborted in the middle due to a kernel microreboot. Some applications or application libraries may have to be modified to handle this error code correctly, e.g., by re-executing the system call. For example, as we will show in the Chapter 6, the JOE text editor treats any failure of console read operation as a critical error, and we had to change this behavior in order for JOE to be successfully resurrected.

The naive policy of reissuing system calls which were aborted due to a kernel microreboot is simple but may not always work for some types of system calls, because it is unknown how much of the system call had been executed before the microreboot. For example, the *bind()* system call assigns a port number requested by an application to a socket when this port number is available. If a kernel crash occurs after the port had

been assigned to the socket but before the *bind()* system call returned, then the crash kernel restores the socket with the port number assigned to it, but the system call returns an error. In this case, the reissued *bind()* system call will also fail because the socket already has a port assigned.

System calls belong to one of two categories: *idempotent* system calls are the calls that can be safely reissued without breaking application expectations, while *non-idempotent* system calls, if being reissued after microreboot, result in a behavior different from the case when these system calls are completed without being interrupted.

In total, the Linux kernel version 2.6.18 has 317 system calls. Out of them, 215 system calls are idempotent. Examples of idempotent system calls include *getpid(), getcwd(), time(), poll()*. 45 system calls are not idempotent. Examples of non-idempotent system calls include *write(), bind(), recv()*.

In addition, there are 55 system calls used for creation and deletion of different resources that are not strictly speaking idempotent, but reissue of these system calls usually do not affect application execution. For example, the *unlink()* system call deletes a file. If a process issued the *unlink()* system call, the file was successfully deleted from the file system, but the system call did not return when a kernel microreboot started, then, when the process reissues the system call after the resurrection, *unlink()* will return that the file does not exist, but nonetheless, the file will be deleted. Another example is the *socket()* system call that creates a new network socket for a process. If at the time of a kernel microreboot, the socket was successfully created, but the system call did not return, then the newly created socket will be resurrected, but the crash kernel will cause the system call to return an error code. This results in a one time resource leak, but if the process reissues the *socket()* system call, it will succeed and a new socket will be created for the process.

Some system calls may be idempotent or not depending on parameters passed or the context in which they are used. For example, semantics of the *ioctl()* system call depends

on a 3rd party kernel extension that implements it.

There are several possible solutions to the problem of handling non-idempotent system calls:

1. Consider using crash procedures. If a process execution was interrupted in the middle of a non-idempotent system call, after the resurrection, do not continue process execution, but just call its crash procedure to allow the process to save its data and restart.

2. Modify application or library code so that it correctly handles non-idempotent system calls failures due to the kernel microreboot. In fact, some Linux system calls require such handling irrespective of microreboots. For example, if *read()* or *write()* system calls fail for whatever reason, it is left unspecified whether the file position changes or not [78]. As a result, any correctly written application that handles I/O errors has to have functionality to handle situations when the current file position is unknown.

3. Use the transactional system call mechanism described by Porter et al., which automatically preserves all information necessary to roll back any changes to the system state made during the system call execution using a copy-on-write approach [100].

4. Use the transactional system call mechanism described by Lenharth et al., which automatically preserves the information necessary to rollback any changes to the system state made during the system call execution using compile-time instrumentation [70].

5. Modify the kernel so that before proceeding with non-idempotent system calls, it saves the information necessary to revert all changes that might affect correctness of application execution if the application reissues the interrupted system call after a kernel microreboot.

The first and the second options do not require any kernel code changes but reduce the applicability of Otherworld, since many applications would require modifications in order to be able to survive an operating system kernel crash.

The third option is to use the transactional system call mechanism developed by Porter et al. [100]. Although, it requires significant changes to the Linux kernel (23,000 lines of changed or added code), the overhead of system transactions is under 20% for workloads that run for more than one second. Write intensive workloads even demonstrate performance improvement by as much as factor of 5x.

The forth option is to use a system call rollback mechanism, but in this case, the mechanism is based on the compile-time instrumentation developed by Lenharth et al. [70]. This approach requires minimal changes to the Linux kernel (100 lines of code) but requires a special compiler and may introduce performance overhead ranging from 8% to 560% depending on a benchmark.

The last option does not require any application modifications, but requires kernel modifications for each non-idempotent system call. However, we expect the modifications to be straightforward in most cases because we need only to preserve the portion of the modifiable kernel state that is visible to applications. System calls usually modify the state of a certain resource. e.g., a file or a socket. Therefore, generic per-resource type functions can be added to the Linux kernel to preserve user-visible data of the resource, e.g., the current file position for a file or the sequence number of the last byte written to the user space from socket read buffers. Each non-idempotent system call before modifying a resource should call the corresponding function.

During resurrection, if the crash kernel detects that a thread of the process being resurrected was interrupted while executing a system call, the crash kernel executes a code that retrieves the information necessary for a system call rollback and restores the corresponding resource using this information. For example, for the file *read()* or *write()* system calls, only the current file offset needs to be rolled back. The function that

preserves the user-visible state of the file resource saves the original value of the current file offset. During resurrection, if the crash kernel detects that the crash occurred when the main kernel was executing *read()* or *write()* system call, the crash kernel will recover the preserved value of the file offset instead of the default one. Once the correct file offset is restored, the corresponding read or write operation can be safely reissued. Another example is the *recv()* system call that copies information received by a network socket from the kernel to an application-supplied buffer. In this case, the kernel has to be modified not to discard the kernel buffer contents that were copied to user space until the next *recv()* call is issued or the socket is closed.

The last option requires adding extra code to all non-idempotent system calls. However, only a few bytes of information needs to be preserved for the most popular system calls. As we will show in Section 7.3.2, the overhead of this code is negligible. Therefore, we consider this option to be the most optimal solution to the problem of reissuing non-idempotent system calls that were interrupted by microreboot.

Rolling back partially executed system calls, as required by options 3, 4, and 5, poses an interesting problem that requires careful synchronization. While the crash kernel can detect whether a kernel crash occurred when a process was executing a system call, and the crash kernel can retrieve saved rollback information, it can not without additional synchronization tell whether the rollback information was saved by the currently executing system call or by the previous invocation of a system call that modified the same resource. To address this, we make use of the Linux kernel property (also common to other kernels, e.g., Windows NT) that all system calls start with code common to all system calls and also end with code common to all system calls. This code is located at a fixed address known to the crash kernel. In addition, we introduce a per-thread *"rollback"* flag. This flag is cleared by the common system call start code and set by non-idempotent system calls. When set, this flag signals the crash kernel that the last system call that the thread was executing is a non-idempotent system call and that rollback information

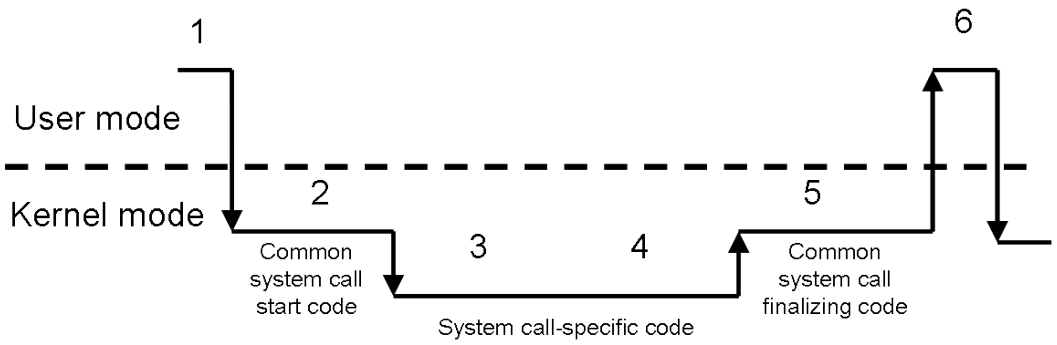

Figure 3.5: Application execution flow

is available.

Execution flow of an application issuing system calls is shown on Figure 3.5. Application execution can be in one of the following six stages (stage numbers in the list below corresponds to the numbers in the figure):

1. Application code executes and issues a system call.

2. The common system call start code executes and at some point clears the *"rollback"* flag.

3. If the system call is non-idempotent, information necessary for the system call rollback is saved. The last step of this stage is to set the *"rollback"* flag indicating that the current system call is non-idempotent and needs a rollback if crash happens.

4. A kernel system call-specific non-idempotent code executes.

5. The kernel common system call finalizing code executes.

6. Application code executes.

After the crash, when the crash kernel resurrects a process, it checks the address of the last instruction that was executed before the crash occurred and determines if it needs to use the rollback information or not based on the following algorithm:

- If the address of the instruction is in the user space, then the crash happened at stage 1 or stage 6, which means that the thread successfully completed the execution of the previous system call and no system call rollback is required. In this case, we ignore the *"rollback"* flag.

- If the address of the instruction is within the common system call start code, then the crash has happened at stage 2, which means that at the time of the crash, the execution of the system call was in progress, but no non-idempotent code has been executed yet, so no rollback is required. In this case, we also ignore the *"rollback"* flag.

- If the address of the instruction is in the kernel, not in the common system call code, and the *"rollback"* flag is not set, then the crash occurred at stage 3, but no rollback is required since no non-idempotent code was executed.

- If the address of the instruction is in the kernel, not in the common system call start code, and the *"rollback"* flag is set then the crash has happened in stage 4 or stage 5, and a rollback is required. When the flag is set, we know that all appropriate rollback information was collected.

3.5 Final Recovery Steps

After a kernel failure, it is important not only to restore application state but to also restore the full functionality of the system and protect the system from failures that may occur in future. After all required application processes are resurrected, the physical memory that belonged to the main kernel is no longer needed. The crash kernel thus reclaims all of the available physical memory and adds it to its free memory list. One region of the reclaimed memory is reserved, and another kernel image is loaded into this region. As soon as this is done, the crash kernel starts playing the role of the main kernel,

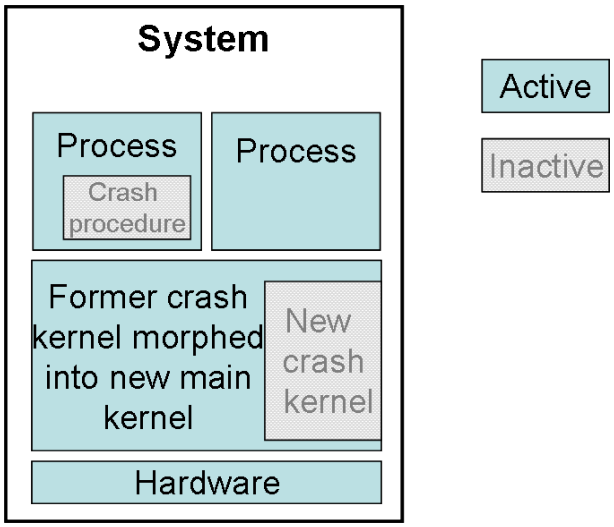

Figure 3.6: The crash kernel takes over the system and morphs into the main kernel

and the newly loaded kernel becomes the crash kernel. As a result, the system is running with a fresh kernel, which is free of state corruption caused by the fault, the applications that were running at the time of failure are able to continue their execution or at least preserve their data, and the system is again protected from failures (Fig. 3.6). This last action finalizes the microreboot process and restores the system to its full functionality and capacity.

3.6 User and Program Interface

In the previous sections, we described each of the individual steps required for system recovery after an operating system kernel failure: prepare the system for kernel microreboot, perform microreboot, and resurrect each process. Each of those steps can be controlled and customized through a set of interfaces provided for system administrators and applications to help automate and tune the microreboot process for a specific environment. In this section, we describe these interfaces.

There are four levels at which a system administrator or a program can communicate

Level	Mechanism	Capabilities
User	Command line utilities	• Load crash kernel • Detect if current initialization is part of the microreboot
Scripts	Command line utilities	• Show processes interrupted by the microreboot • Show information about interrupted process
Application	System calls	• Resurrect a process with given process id • Reclaim remaining resources
Kernel modules	Kernel APIs	• Detect if current initialization is part of the microreboot • Read memory state of the rebooted kernel

Table 3.2: Otherworld interfaces.

with Otherworld, as summarized in Table 3.2. The top three levels are used to protect the system with the crash kernel and resurrect processes. The lowest level is used by the process resurrection code itself and may also be used by crash kernel modules that might want to optimize their initialization using state stored in the memory of the main kernel.

Command line utilities are used to configure Otherworld and manage the microreboot and resurrection process, from start-up scripts, or manually from the command line by a system administrator. For more fine-grained control over the resurrection process, we provide developers with an application programming interface through calls to the operating system kernel. Finally, we provide a kernel-level interface to be used by kernel modules. Below, we describe each of these interfaces in more detail.

Two command line utilities are provided: *kexec* and *ow_exec*. *Kexec* instructs the main kernel to load the crash kernel image into memory from a specified location on disk. *Kexec* is typically run at the beginning of the system start-up scripts.

The *ow_exec* utility allows a system administrator to manage process resurrection after the crash kernel is initialized. It has several sub-commands that are specified through

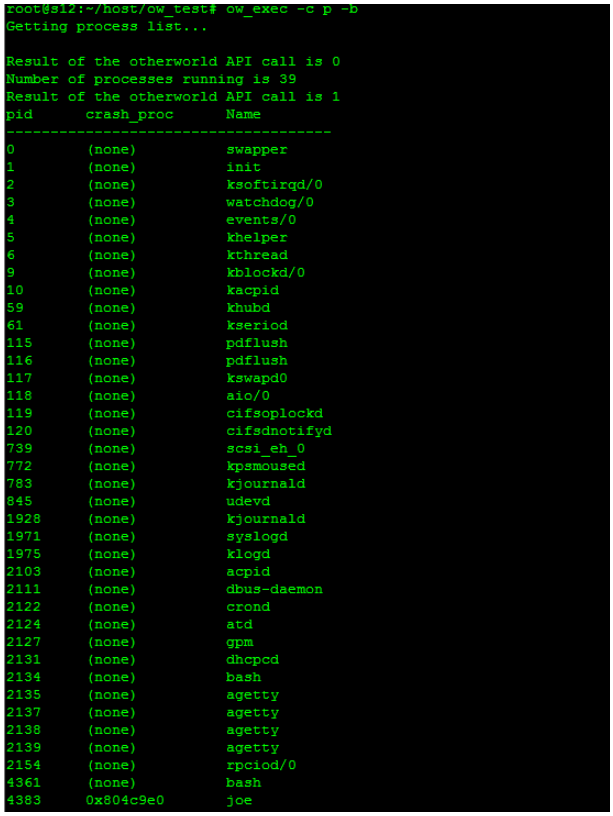

Figure 3.7: List of processes running at the time of the microreboot displayed by the ow_exec utility

command line parameters. The first sub-command displays the list of processes that were running on the main kernel at the time when the kernel microreboot occurred (Fig. 3.7). This sub-command can also display a variety of process information, such as the name, id, and whether the process had registered a crash procedure. The second sub-command resurrects the process with a given id. Finally, the third sub-command tells the crash kernel that it can reclaim and use all resources that belonged to the main kernel.

We believe that in most scenarios, the end user is interested in resurrecting only a few important processes that were running at the time of the failure, such as perhaps a database server, a web server, or a text editor. The other processes, such as the window manager, the mouse server, or the cron daemon, do not hold important state and can be safely restarted without resurrection in most scenarios. Not resurrecting these

Request	Description	Parameter
SET_CRASH_PROC	Registers the crash procedure for a process	Pointer to an *ow_crash_params* structure
GET_CRASH_PROC	Returns the process crash procedure parameters	Pointer to an *ow_crash_params* structure
GET_TASKS	Returns the attributes and number of processes that were running on the main kernel at the time of a crash	Pointer to an *ow_get_task_params* structure
OCCUPY_MEMORY	Tells the crash kernel to reclaim all resources used by the main kernel	Ignored
GET_STATISTICS	Asks the crash kernel for statistical information collected during process resurrection	Pointer to a *ow_statistics* structure

Table 3.3: Parameters of *otherworld()* system call .

processes eliminates the possibility of any side effects on these processes that might have been incurred by the kernel crash failure. The *ow_exec* utility is typically invoked after the crash kernel finishes its initialization. Towards the end of the system microreboot, start-up scripts run the *ow_exec* utility to present the interactive user with a list of the processes that were running on the system at the time of the crash. The user can then select the processes that should be resurrected.

Alternatively, a system administrator can create a resurrection configuration file that identifies which processes are to be resurrected automatically based on specific process name, terminal, or user name. The startup scripts call the *ow_exec* utility to list all processes that were running at the time of the crash and consult this file to determine which processes to resurrect. Finally, the start-up scripts call the *ow_exec* utility again for each process that matches the criteria specified in the configuration file telling the utility to resurrect this process. The latter option is intended to be used by server systems for

autonomic recovery, and we used it during our automated fault injection experiments (see Chapter 7). After all processes that need to be resurrected have been resurrected, the start-up scripts or user instruct the crash kernel to reclaim all resources by calling the *ow_exec* utility. Finally, the start-up scripts or user loads another crash kernel by calling the *kexec* utility.

In order to interact with Otherworld, applications, such as the *ow_exec* utility, can use three system calls: *kexec_load()*, *otherworld()*, and *clone()*. The first system call is used to specify the crash kernel image to be loaded into the reserved region. The second system call is used for several commands and has the following syntax:

```
long otherworld(int request, void* param);
```

The first parameter specifies the Otherworld request that the kernel has to invoke, and the second argument specifies associated parameters. Possible requests and parameters are listed in Table 3.3, and a detailed description of the parameters is given in Appendix A.

Both process resurrection and the cloning of an existing process have a lot in common: in both cases the copy of another process is created. Because of this, in our implementation, we modified the existing *clone()* call to handle process resurrection as well. In order to instruct the crash kernel to resurrect a process with a given process id that was running on the main kernel, an application must call the *clone()* system call, specifying as parameter the id of this process and the flag that resurrection is required.

For loadable kernel modules, Otherworld provides an API function that allows reading of memory from the main kernel:

```
ssize_t ow_read_oldmem(char *buf, size_t count, unsigned long *ppos);
```

This function reads *count* bytes from the main kernel memory at location **ppos* into a buffer *buf*, and its syntax is similar to the standard file *read()* function. Using this function, a loadable kernel module can preserve data across a kernel microreboot,

e.g. routing tables, ip address leased by a DHCP server, or hardware configuration parameters.

3.7 Summary

This chapter presented the architecture and design of Otherworld that allows microrebooting an operating system kernel in response to an unexpected kernel failure (crash). After a microreboot, a new, freshly initialized kernel controls the system. Applications can survive the operating system kernel microreboot and continue their execution after the microreboot is complete. The kernel microreboot may go completely unnoticed for relatively simple applications but may require some cooperation for more complex applications.

A flexible programming interface helps Otherworld adapt to the requirements of a specific system and makes an operating system crash nearly transparent to end users allowing running their applications to continue to run without any data loss. In the next chapter, we will discuss specific implementation details of Otherworld within the Linux operating system kernel.

Chapter 4

Implementation of Otherworld

4.1 Overview

In the previous chapter, we described the architecture and design of Otherworld. Otherworld does not rely on unique characteristics of any operating system, but in order to show its viability, we implemented it in the Linux operating system kernel. We have chosen Linux because of its popularity and source code availability. In this chapter, we describe implementation details specific to the Linux kernel.

We have implemented Otherworld in Linux kernel version 2.6.18. The implementation required changing fewer than 600 lines of existing code and adding 3,000 new lines of code, as summarized in Table 4.2. To put this into perspective, the 2.6.18 Linux kernel consists of approximately 4.9 million lines of code, and the Debian Linux 4.0 distribution consists of about 280 million lines of code [2]. Modifications were required to the start-up code, the file management code, and to the *clone()* system call. The start-up code modifications were required in order to reserve a memory region for a new crash kernel and to add memory to the crash kernel after the resurrection process has completed. File management code had to be modified in order to simplify the restoration of open files. Both process resurrection and the cloning of an existing process have a lot in common,

Purpose	**Number of lines**
Modified code	
Kernel initialization	53
File management	74
Network sockets	11
KDump fixes	34
Startup code	80
clone() system call	254
Other:	20
Added code	
Process resurrection	2947
Total	3473

Table 4.1: Modifications to the Linux kernel introduced by Otherworld.

since in both cases the copy of another process is created. Because of this, we modified the *clone()* system call to handle both operations. Most of the new code was added for retrieving and recreating process information from the failed main kernel. In our implementation under Linux, we use the KDump mechanism, which is part of the Linux kernel, to load the crash kernel into memory and pass control to it after a failure is detected [53].

In the next section, we discuss the changes that we made to the stock Linux kernel. The second section describes the details of the resurrection process.

4.2 Kernel Code Modifications

The current implementation of Otherworld requires minimal changes to the stock Linux kernel. Most modifications execute only at initialization or microreboot time. This

contributes to one of the key benefits of Otherworld: negligible run-time overhead.[1] Indeed, the only extra code that is executed during normal system operation is the saving of the open file and socket creation system calls parameters, which significantly reduces the complexity of the resurrection code and reduces the number of main kernel data structures we have to rely on. In the Linux kernel, information relating to open files is located in the *file*, *inode*, and multiple *dentry* structures, but in order to recreate an open file resource, only the file location, name, open flags and current file offset are required. We modified the *file* structure to also store the location, name, and open flags specified by the application during the *open* call. As a result, we only need to rely on one structure to recreate the kernel open file state.

The layout of many kernel data structures that describe network sockets depend on the type and the family of the protocol the socket was created for. These structures are created based on the protocol type and the protocol family specified by the application through the *socket()* system call at socket creation time. We save the parameters that were passed to the *socket()* system call in the socket descriptor structure, so that the socket can be easily recreated by the crash kernel.

We also had to modify the Linux memory allocator so that it allocates physical memory below 1 MB only for DMA transfers. This was necessary because the Linux kernel uses the physical memory region between 0 and 1 MB during its boot process, which is also the case for the crash kernel. Without this special precaution, this memory, if used by the main kernel, would be corrupted by the microreboot process. A possible alternative would be to copy the first megabyte of physical memory to some other memory location before the microreboot, but the copy code would have to be executed in the context of the main kernel after a failure is detected, which increases the likelihood of a microreboot failure, which we wish to avoid.

[1] Additional, optional application state protection does add overhead, and will be discussed in the next chapter.

4.2.1 Startup code modification

Some user applications and kernel code might depend directly or indirectly on system uptime. For example, the TCP protocol adds timestamps based on the system uptime to each packet sent. Therefore, we modified the crash kernel's startup code to retrieve the main kernel's uptime and number of timer interrupts (*jiffies*) and initialize the corresponding crash kernel's variables with these values. We talk about TCP timestamps in more detail in Section 4.3.2.

In order to be able to dynamically increase the amount of physical memory allocated to the crash kernel, the start-up code of the crash kernel has to allocate extra page descriptors that are not used by the crash kernel during the resurrection process but will be used when the resurrection process is complete.

Pre-allocation of these extra page descriptors consumes approximately 10% of the memory reserved for the crash kernel by the main kernel. A more elegant and effective solution would be to allocate page descriptors dynamically as required. Unfortunately, the current Linux kernel implementation relies on page descriptors for all memory pages to be allocated continuously and at the fixed, predefined location. Changing this behavior would have required multiple modifications to the Linux kernel code and our goal was to minimize changes to the stock Linux kernel.

Two other startup code changes are related to TCP network sockets resurrection. Each TCP socket uses a port number, a unique two-byte integer assigned to any open TCP connection. These port numbers are either specified explicitly by the application or are assigned at runtime by the kernel from the pool of available numbers. Two open TCP network connections cannot share the same port number on the same machine. This can be a problem for resurrecting client applications, because client applications typically use ports assigned by the kernel, which chooses them randomly from the fixed range of dynamically allocated ports. The port, once assigned, cannot be changed for a connected TCP socket. When resurrecting an application, the crash kernel can find that

the port number of a socket that belongs to the application it is trying to resurrect has been already assigned to a different application running on top of the crash kernel. In order to avoid such a conflict, the crash kernel during its startup checks the main kernel memory for port numbers that were in use and reserves them until the resurrection of all processes is complete. The crash kernel can retrieve the list of all allocated ports, because all ports allocated to open sockets are listed in the *inet_hashinfo* hash table referenced by the global variable *tcp_hashinfo* inside the main kernel memory.

The second change modifies the way the crash kernel processes unexpected TCP packets. By default, the Linux kernel sends back a TCP reset packet for each incoming TCP packet that is destined to a port that does not have an associated open socket. If, at the time of a crash, a process had an open socket, and the remote party sends a packet, which reaches the crash kernel before the process is resurrected, then no process will be listening on the incoming packet's destination port, and a reset packet is sent back. This forces the remote party to terminate the connection. We changed this behavior to silently ignore all incoming packets until all processes are resurrected. In this case, the remote party continues to resend dropped packets until we resurrect the application that listens to the target port, or the maximum number of retransmissions is reached. After resurrection of all processes is complete, we resume the default policy of sending reset packets. We cover further details of the TCP network sockets resurrection in Section 4.3.3.

4.2.2 Stall Detection

In some cases, a kernel fault results in the system getting into a stalled state, when it stops scheduling application code for execution and does not respond to external events, such as interrupts. This can happen, for example, when the kernel blocks interrupts and starts waiting on a spinlock, which is never released. In other cases a kernel failure can result in a recursive sequence of failures. In the first case, CPU is busy looping in a spinlock wait

cycle and the failure detection code does not have a chance to detect this situation. In the second case, the failure detection code experiences a failure before or while trying to transfer control to the *panic()* routine because of the significant kernel state corruption. We address both cases by enabling Linux software lock detection (which is disabled by default in the Linux kernel) and using a hardware watchdog timer driver. The hardware watchdog timer is a simple electronic circuit that issues a non-maskable interrupt to a processor after a specified amount of time, unless the processor resets it. The hardware watchdog timer driver is invoked by the kernel periodically to reset the timer. If the kernel becomes stalled or there is a cascading sequence of failures, the driver will not be invoked and will not be able to reset the timer, causing a non-maskable interrupt to be issued. The interrupt handler then responds to this interrupt by microrebooting the kernel.

Although not widely used in practice, the hardware watchdog timer is a common component of many modern x86 and ARM chipsets. Intel has included a watchdog timer in all of its x86 architecture chipsets since 2002 [59]. AMD also includes a watchdog timer in many of its chipsets. Both Linux and Windows are shipped with watchdog drivers.

4.2.3 KDump Modifications

The original purpose of KDump was to provide error information and a physical memory dump of the kernel for developers. In Otherworld, we use KDump to load the crash kernel into the memory and to transfer control to the crash kernel when a failure inside the main kernel is detected. In order to improve reliability, we simplified the code that transfers control to the crash kernel and extended KDump to handle additional types of failures, such as double fault interrupts and stalls. We also modified KDump to dynamically allocate the reserved region for the crash kernel.

We briefly describe our modifications to KDump. After a failure in the kernel is detected, the code that transfers control to the crash kernel runs in the context of the

main kernel, meaning that it uses the main kernel stack and data structures. This code can be affected by corruption that was caused by the fault that led to the failure. It is therefore important that this code be simple and be able to handle data corruption. In order to achieve this we did several modifications to the original KDump code. Because the modified code is executed only after the failure was detected, none of our changes affect application performance.

- The original KDump preserves the thread context of the thread that caused the failure in a reserved region of memory. However, Otherworld only uses global kernel state and thus discards local thread context of threads executing kernel code. We, therefore, disabled the code that preserves the kernel thread context.

- KDump walks the current thread's stack to create a stack trace. The existing KDump code assumes that the stack is not corrupted. We added additional checks to detect stack corruption so that we can avoid infinite loops and invalid memory accesses[2].

- KDump attempts to access the process descriptor of the currently executing process. This may result in a microreboot failure, when the main kernel fails within an interrupt context and the descriptor of the current process is not accessible. We removed any references to the currently executing process from the code that transfers control to the crash kernel.

There are several places in the Linux kernel that may detect kernel failures. We found that not all of them trigger a panic - some just stall the system when a failure is detected. The first such place we encountered was in the *double fault* interrupt handler. The processor triggers a *double fault* interrupt if it encounters a problem while trying to service a pending interrupt or exception. A situation where a double fault interrupt may

[2]We limit the number of stack frames the stack walking code tries to parse to 1000 and verify that the virtual addresses that the stack walking code reads from the stack and tries to access are valid and reference memory that belongs to the kernel.

occur is when an interrupt is triggered but the segment in which the interrupt handler resides is invalid. The Linux kernel simply stalls the system by blocking interrupts and going into an infinite loop instead of invoking a *panic()* routine in response to double faults in the kernel code. We modified the Linux double fault interrupt handler to start the microreboot process instead. We also found a similar problem in the Linux software stall detection code and fixed it as well.

4.3 Process Resurrection

A process is resurrected by traversing the main kernel data structures that contain the kernel state used by the process, retrieving from those structures the required state so that the crash kernel can recreate the process resources. In Figure 4.1, we show the kernel data structures (except the network sockets-related structures, which will be discussed later) and global variables that we use for process resurrection.

After the main kernel is compiled, the location of all global variables is known and hardcoded into the crash kernel. One of these variables, *init_task*, points to the process descriptor of the first process in the system. In Linux, the process descriptors are organized in a circular, doubly-linked list. Thus, knowing the location of one process descriptor, the crash kernel can access the process descriptor for any process. Another global variable points to the swap area descriptors stored in a fixed size array. Each array element describes one swap partition and contains a pointer to the *file* structure that corresponds to a regular file or a device file that stores the swap area. Since the symbolic name of the device is stored in this structure, the crash kernel can reopen it. In the following sections, we describe the implementation details of process resurrection.

Whenever a process attempts to use or create some resource, the kernel checks the privileges of the user under which the process is running. Some processes start with an elevated privilege level, perform actions that require these elevated privileges, and then

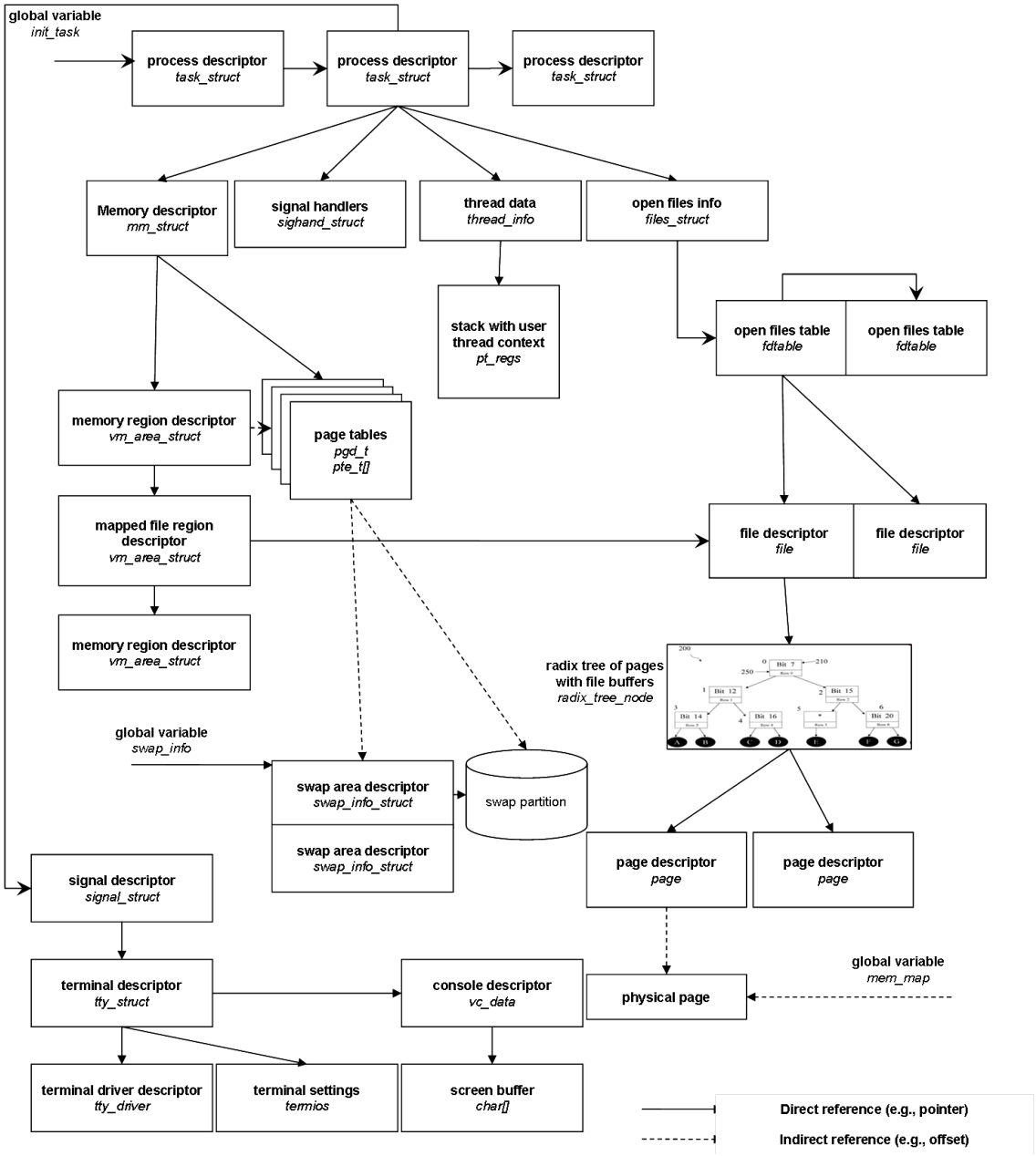

Figure 4.1: Main kernel data structures used for resurrection and their interdependencies

lower their privilege level to a normal level. This is done to reduce the consequences of potential security vulnerabilities. For example, the *ping* application uses raw IP sockets and always starts with superuser privileges because creating a raw IP socket can only

be done by the superuser. After the socket is created, *ping* explicitly lowers its privilege level to the level of the user that started it. The main kernel only knows the current privilege level of a process; therefore, if during resurrection the crash kernel tries to use the resurrected process's privileges when recreating the raw IP socket, the recreation will fail due to not having the appropriate privileges. To solve this problem, we perform all resource resurrection with superuser privileges. This should not constitute a breach of security because the resource access was already granted to the process being resurrected by the main kernel. However, this might confuse some access auditing applications. Alternatively, the main kernel might record in the resource descriptor the privileges at the time of the resource creation. The crash kernel can reuse these privileges when it recovers the resource.

4.3.1 Recovering Memory Space

Each process descriptor contains an entry that identifies the location of a memory descriptor for the process. The memory descriptor is created together with the new process descriptor and contains a reference to the list of virtual memory regions that the process has allocated. Each virtual memory region is described by the memory region descriptor structure. Memory region descriptors for each process are organized in a list, and the memory descriptor contains a field which points to the beginning of this list.

In Linux, a virtual memory address space is split into two portions. The kernel portion of the address space belongs to the kernel and is not accessible by application code. In Linux, the kernel portion of the address space is shared between all processes running on the same kernel, i.e. it has the same virtual to physical memory mapping. When Otherworld recreates the kernel portion of the address space, it reuses the logic of the regular *clone()* call to share the kernel portion of the address space of the process it is trying to resurrect with the other processes running on top of the crash kernel.

The second, application portion of the address space belongs to the application and does not contain any data that belongs to the kernel. This portion of the address space is made to be an exact copy of the application portion of the address space of the process being resurrected. To do this, Otherworld reads from the main kernel the list of the memory region descriptors that describe the application portion of the address space and creates new memory region descriptors with the same parameters. This duplicates the memory layout of the process we are trying to resurrect. The next step is to copy the contents of the process's memory.

The memory descriptor also points to hardware page tables that define the mapping between physical memory pages and virtual addresses. When recreating the application portion of the address space, Otherworld reads these tables to get all physical memory pages with application data. For each hardware page table entry that points to valid physical memory page in the main kernel memory, the resurrection code asks the crash kernel to allocate a new physical page and copies the contents of the page from the main kernel's memory to the newly allocated page. The newly allocated page is mapped at the same virtual address as the original page.[3]

A hardware page table entry may indicate that the corresponding page has been swapped out. In this case, Otherworld also asks the crash kernel to allocate a new page and reads its data from the swap partition used by the main kernel.

Some of the memory regions may have been mapped to a disk file. In this case, there is a pointer in the memory region descriptor to the corresponding *file* structure that describes this file. This file is reopened (see the next subsection) and mapped to the same region. In this case, the mapped data does not need to be read from disk immediately, the corresponding page table entry of the process that will replace the target process being resurrected is marked as invalid, and the Linux kernel page fault

[3]We suggest a possible optimization of this where physical pages are not copied but remapped in Chapter 8, when we discuss the future work.

handler will load it when application tries to access the page.

4.3.2 Recovering Open Files

The process descriptor contains a pointer to a file descriptor table (*files_struct*), which describes all files that were open for the process when the main kernel failed. It contains an array of open file descriptor pointers. As we described in the previous section, we modified the Linux code that opens files, to store file open parameters, including the file name and flags, in the file descriptor. Thus, using file descriptors alone, we are able to reopen the same files for the newly created process that will replace the target process being resurrected.

An open file may contain data modified by the application and stored in memory file buffers (file cache) but not yet saved to disk. File buffers are organized as a radix tree [23]. The root of this tree for each open file is accessible through the file descriptor. Each leaf element of the tree contains a pointer to a descriptor of a physical page with file data. The page descriptor contains a *dirty* flag, which is set by the main kernel when the page is modified (to indicate that the corresponding page needs to be saved to disk), and the offset of the data relative to the start of the file. When the crash kernel reopens a file, it retrieves all file's pages with a set dirty flag from the main kernel memory and saves these pages to disk.

While some file data needs to be read from the main kernel and synchronized to disk, namely the modified blocks, file metadata does not need to be retrieved. Modern file systems, such as ext3, reiserfs, NTFS, and JFS, use journaling techniques [56] for all metadata updates. Journaling implies usage of an auxiliary log to record all metadata operations. When the crash kernel mounts a file system, the file system driver replays or rolls back all actions recorded in the log ensuring atomicity and consistency of all metadata operations.

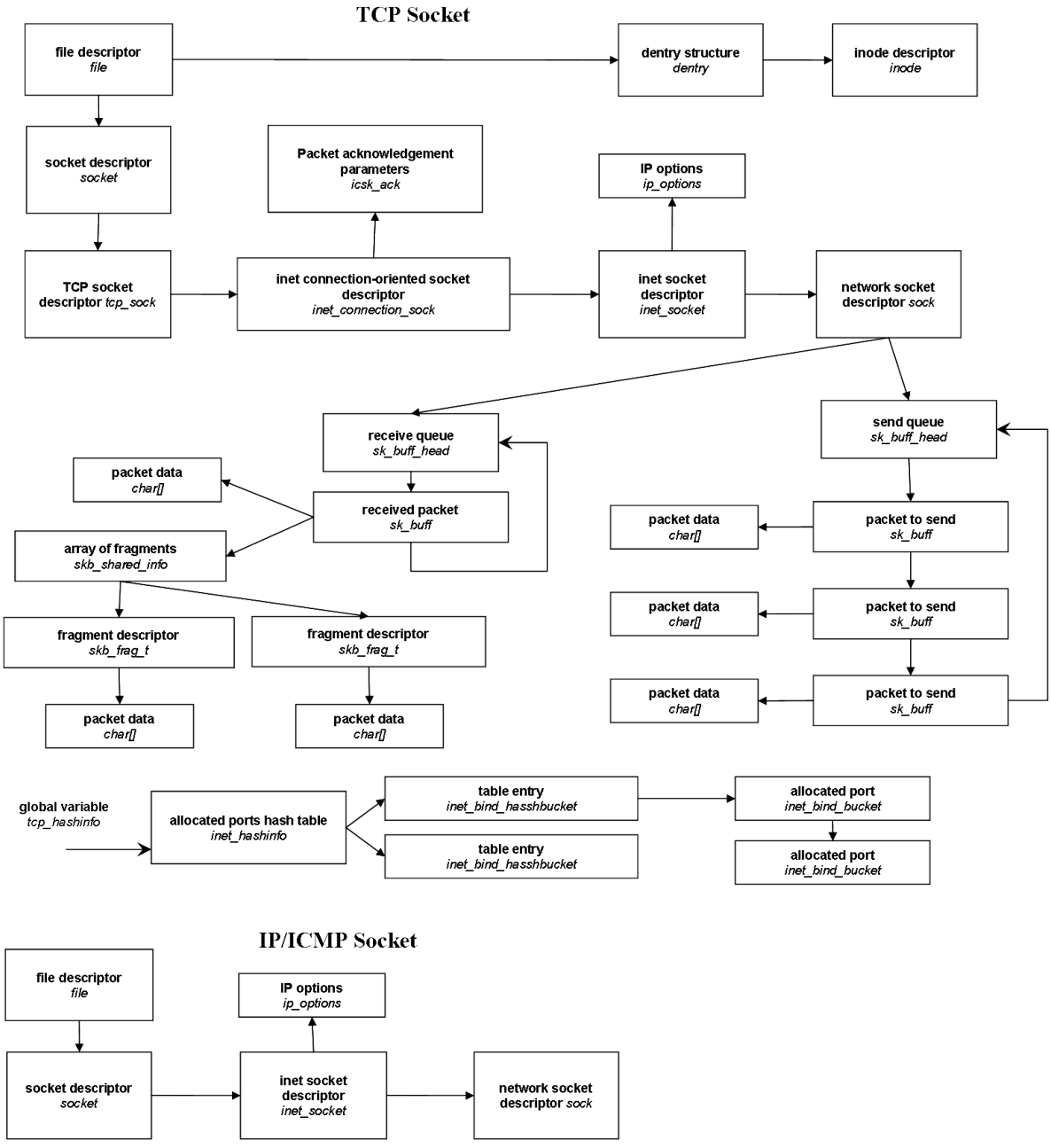

Figure 4.2: Main kernel data structures used for network sockets resurrection

4.3.3 Recovering Open Network Sockets

Probably, the most common application interface for communication over a network is the network socket interface. The networking subsystem is arguably one of the largest and

most complex subsystems in the Linux kernel. Although the network sockets interface is mostly the same for different network protocols, the kernel implementation of the protocols differs significantly because Linux lacks its own common network driver model, instead providing only a few common helper functions and structures. Moreover, even within these common structures, the same fields point to different data types at runtime when used by different protocols. Thus, the specific data type referenced by some pointers can be determined only at runtime. This is one of the reasons why the socket resurrection code has to be protocol dependent. Currently, our Otherworld implementation supports the resurrection of network sockets for TCP over IP, as well as resurrection of raw IP and ICMP sockets. These are arguably the most popular network communication protocols. The structures that have to be retrieved and analyzed for TCP over IP and raw IP sockets resurrection are shown in Figure 4.2.

Processes do not have a separate table of open sockets. Instead, for each open socket, there is a file descriptor in the process's file descriptor table. Each file descriptor contains fields that store pointers to function that handle file operation (e.g., *read()* or *write()*). For sockets, these fields contain values that point to special sockets functions, which are different from the functions that handle regular disk file reads and writes. By checking these pointers, the resurrection code can distinguish between the file descriptors of regular files and socket descriptors. In addition, the inode structure for a socket's file descriptor has special flags set, and inode operations' function pointers point to socket-specific functions instead of regular disk inode functions. These indications are redundant, but they are used for corruption detection.

File descriptors that describe sockets contain a pointer to a corresponding socket descriptor structure. The specific protocol for a socket is specified during the socket creation through the *socket()* system call. We modified this system call in the main kernel to save socket creation parameters in a socket descriptor to be able to easily recreate an open socket and resolve the pointer ambiguity mentioned above.

4.3.4 Recovering Raw IP sockets

Networks are inherently unreliable, and network packets can be lost or corrupted for a number of reasons. The IP protocol does not guarantee that a packet sent by a host will be delivered to its destination, or that a packet will not be corrupted [104]. This makes raw IP socket resurrection relatively straightforward because during the recovery of a raw IP socket, we can safely discard all network IP packets queued for sending or packets received by the kernel as if they were lost en route. Applications that use the IP protocol directly have to take the probability of packet loss into consideration and correctly process this situation.

To recover a raw IP socket, Otherworld creates a new raw IP socket in the crash kernel, retrieves the destination address for the socket from the main kernel's memory, and recalculates the route to the destination, i.e., the network interface to pass the packet for sending, gateway address, etc. Next, Otherworld recovers from the main kernel memory all socket options that were specified for the socket by the process Otherworld is trying to resurrect. Then, Otherworld creates a new file descriptor for this socket and assigns it the same file descriptor number as was assigned for this socket's file descriptor before the crash. All IP socket options are stored in the inet socket descriptor, the *ip_options* structure, and in the network socket descriptor referenced by the socket descriptor (Fig. 4.2). All incoming and outgoing network packets that the main kernel was processing at the time of the crash are discarded, and the application that owns the socket can deal with this after its execution continues.

4.3.5 Recovering TCP sockets

In contrast to the IP protocol, the TCP protocol is reliable, which means that applications can expect that no data is lost during transmission [105]. TCP interprets data passed through it as an ordered stream of bytes. For each TCP connection, there are two

streams, one in each direction. As soon as a connection is established, both streams function identically. Each party serves as a receiver for one of the streams and as a sender for the other.

The TCP protocol assigns to each byte it transfers a 32-bit *sequence number*, the position of the byte relative to the beginning of the stream. The sender adds the sequence number of the first byte of the data in a packet to all TCP packets it sends. The receiver uses this sequence number to correctly order packets, detect missing packets, and eliminate duplicates. Reliability is implemented by requiring the receiver to transmit back to the sender the sequence number of the next byte it expects to receive, called *acknowledgement number*. Transferring this number means that the receiver has received all bytes up to the one specified as the acknowledgement number. If the sender does not receive the acknowledgement for the packets it has sent within a timeout interval, it retransmits all the data it has already sent starting with the byte that has the sequence number equal to the last acknowledgement number received.

Since TCP is built on top of the IP protocol, when Otherworld recovers a TCP socket, it needs to perform all the actions necessary to recover a raw IP socket as described in the previous section before performing any TCP-specific recovery. When the crash kernel recreates sockets for TCP over IP, it does not discard any of the data packets stored in memory. After all IP-specific state is recovered, Otherworld binds the newly created socket to the same TCP port number the original socket was bound to. Then, Otherworld starts recovering the TCP-specific part of the socket state. TCP state can be divided into three parts: (*i*) connection state, (*ii*) incoming packets queue (i.e., data received by the TCP layer which have not been passed to the application yet), (*iii*) outgoing packets queue (i.e., data passed by the application to the TCP layer for sending to the receiver). We now describe each part in more detail below.

TCP Connection State

TCP connection state consists of multiple connection parameters. We divide connection parameters into two categories. The first category includes parameters that are critical for the correct functioning of the TCP protocol. This category includes the sender's and receiver's port numbers, the sequence numbers of the last byte sent, the last byte received, the last byte acknowledged by the receiver, the last byte passed successfully to the application, and the timestamp of the most recently received packet.

The second category includes parameters that affect TCP performance, but not correctness. These parameters usually receive default values when the connection is established and are then modified to optimize TCP's behavior for changing network conditions. This category includes the TCP window size, the maximum segment size of the receiver, the average transfer rate, and the packet round trip time. Modifying these parameter values to something different from what they were at the time of the microreboot affects the performance of the TCP but does not result in data loss or corruption. For example, the window size parameter specifies the amount of data that the receiver can currently accept. If the sender sends more data than the receiver can accept, the excess data will be discarded by the receiver, and the sender will have to send it again because the receiver does not acknowledge it. This results in more data being sent than necessary. As soon as the receiver receives some data from the resurrected socket, it sends a packet with an acknowledgement and specifies the current window size, thus setting the sender's window size parameter back to the correct value.

Otherworld restores all connection parameters from the first category to the values they had at the time of the microreboot and resets all parameters from the second category to their default values. This results in potentially less than optimal TCP performance immediately after resurrection, but allows us to reduce the amount of data from the main kernel that we have to rely upon.

There are several optional features of the TCP protocol that were introduced to im-

prove reliability and performance of the TCP protocol. In order to calculate a round-trip time and detect situations when sequence numbers wrap-around, TCP uses a *timestamp option* [107]. According to the standard, the TCP timestamp is a monotonous 32-bit integer value approximately proportional to the wall clock time. In Linux, a timestamp is simply the number of timer interrupts (jiffies) that occurred since the operating system started running. If the TCP sequence number reaches the maximum integer value and wraps-around, the receiver of the data has no way of distinguishing between a very old packet and the latest packet transmitted for the first time if they happen to have the same sequence number. With the help of timestamps, a packet can be discarded as an old duplicate if it is received with a timestamp less than some timestamp recently received through this connection. Because of this it is important to preserve jiffies during a microreboot.

The current implementation of Otherworld fully supports the TCP timestamp option. As we discussed in Section 4.2.1, during its initialization, the crash kernel reads the number of timer interrupts from the main kernel and initializes its own counter with this value.

Another optional optimization to the TCP protocol, called *selective acknowledgements*, allows acknowledgement of out-of-order packets, [108] thus reducing the amount of data TCP has to retransmit if one packet is lost or corrupted. Currently, Otherworld does not support the selective acknowledgment option.

Incoming Queue

After recovering the TCP connection state, Otherworld starts recovery of the incoming and outgoing packets queues. Recovery of each individual packet in these queues is relatively straightforward. Each packet is represented by an *sk_buff* structure. This structure contains the packet's properties such as a packet length, a checksum, and a sequence number. Each *sk_buff* structure references a mandatory data region pointed by

the *data* member as well as several optional additional data regions, called fragments. Fragments are described by an *skb_shared_info* structure, which is located at the end of the mandatory data region and is pointed to by the *end* member of the *sk_buff* structure. It contains an array of *skb_frag_t* structures. Each *skb_frag_t* structure describes an individual data region. It contains a field containing a pointer to a page in the main kernel memory with data, the offset within the page, and the size of the data.

The recovery policy of the incoming TCP queue is to discard all packets that have not been acknowledged by the main kernel since they will be resent later by the sender and to put all already acknowledged packets in the incoming queue of the socket, so that the process which we are resurrecting can read them later. In Linux, the TCP incoming queue is not in fact a single queue, but is composed of three distinct queues in order to improve TCP performance by reducing lock contention. These queues are called: the *backlog* queue, the *prequeue* queue, and the *receive* queue.

If, at the time the packet is received, there is some read or write operation in progress for the socket, then the socket receive queue is locked, and the received packet will be appended to the backlog queue. Later, when the socket receive queue is unlocked, packets in the backlog queue are moved to the receive queue. Only after having been moved to the receive queue are acknowledgement for these packets sent to the sender.

Packets are placed in the prequeue if the process has processed all data received by the socket and is waiting in the *read()* system call for new data to arrive. Packets in this queue are not acknowledged, and remain there until the process that waits to read from this socket starts executing, or until a certain timeout is reached. In both cases, packets are removed from the prequeue and placed in the receive queue. Only after packets have been moved to the receive queue, the acknowledgement for these packets are sent to the sender. When a packet is placed in the receive queue the acknowledgement is scheduled for the last in-sequence packet (i.e., the packet the sequence number of which shows that there are no lost or delayed packets).

Therefore, when recovering a socket, Otherworld only need to recover packets stored in the receive queue. Packets from the prequeue queue and the backlog queue are discarded because the main kernel has not sent acknowledgements for them, and consequently the sender will continue retransmitting them until the acknowledgement is received. Otherworld retrieves all the receive queue packet's from the main kernel memory and places them on the receive queue of the newly created socket.

After the recovery of the socket's receive queue, Otherworld sends an acknowledgement packet for the last in-sequence packet found in this queue or, if the incoming queue is empty, it retransmits the last acknowledgement number sent before the crash. Sending an acknowledgement packet serves two purposes. First, it tells the sender from which point it should start data transmission. Second, during its last transmission before the crash and subsequent kernel microreboot, the TCP protocol in the main kernel might have informed the sender that its TCP window size is zero, thus asking the sender to halt transmission. Even if the main kernel later attempted to send an acknowledgement packet with a new non-zero window size, the crash might have prevented the packet from being delivered. Upon receiving a zero window size message, the sender will stop the transmission and will wait for a message with a non-zero window size. Sending an acknowledgement message with a non-zero window size after resurrection instructs the sender to resume transmission.

Outgoing Queue

After recovering the incoming packets queue, Otherworld starts recovering the outgoing packet queue, which in Linux is called the *write* queue. The recovery policy of the write queue is to resend all packets found in the write queue in the main kernel. In order to do this, Otherworld copies all packets from the write queue in the main kernel memory to the write queue of the newly created resurrected socket and sets the sequence number of the last byte sent to be equal to the sequence number of the first byte in the write

queue minus one. This may result in unnecessary retransmission of some packets already received by the other party, but allows us to guarantee the delivery of each packet.

Finally, Otherworld starts all timers associated with the recovered socket. The most important timer is the *retransmission* timer. Triggering it, forces all packets in the write queue to be retransmitted to the other party. From this point on, the TCP socket resurrection is complete and data exchange is resumed.

Timeouts

Because the TCP protocol is layered on top of the unreliable IP protocol, timeouts intervals are used to detect broken connections. Whenever a timeout is reached, the connection is terminated, and the application has to reestablish it again. There are two types of timeouts that one needs to be concerned about when dealing with TCP. One of them is the retransmission timeout described in RFC 793 [105]. This timeout is applicable to data packets that have been sent but have not been acknowledged within the timeout interval. In this case, the TCP protocol tries to retransmit packets some fixed number of times. Initially, the retransmission timer is initialized to an implementation-specific value and later is adjusted on the fly based on a packet round-trip time. The retransmission timer value for a given packet is multiplied by an implementation-specific value after each retransmission of this packet up to a some implementation-specific maximum value. For example, in Windows NT, the initial value for the retransmission timer is 3 seconds and is doubled for each data packet retransmission, but limited to 240 seconds. The maximum number of retransmissions is 5 [82]. In Linux 2.6 kernels, by default, the initial retransmission timer value is 3 seconds but can drop down to 0.2 seconds on low latency networks. The retransmission interval is doubled for each data packet retransmission. The maximum interval of the retransmission timer is limited to 120 seconds and the maximum number of retransmissions is 15. Thus, by default, the time before TCP closes a connection due to a timeout ranges from 189 seconds for Windows to 684 seconds

for Linux even on low latency networks. As we will show in Chapter 7, the current microreboot time is around one minute even for relatively slow machines. This is less than the TCP timeout interval and allows us to avoid dropped connections due to a TCP timeout while the microreboot is in progress. However, applications may use their own timeouts, perhaps shorter than the TCP timeout. Therefore, it is important to minimize microreboot and resurrection times. This is one of our most important goals for the future work.

The retransmission timeout affects only the sender. The receiver by default must wait indefinitely until the new packet arrives [106]. However, keep-alive packets might be sent by TCP if the keep-alive feature is supported by the protocol implementation on both sides. However, if keep-alive is implemented, the application must be able to turn it on or off for each TCP connection, and it must be disabled by default. The TCP protocol keep-alive period by default is set to 7200 seconds, which is much longer than the kernel microreboot time.

4.3.6 Recovering Console Screen Contents

One way to restore the contents of a console screen would be to retrieve it from the video adapter's memory. The advantage of restoring the console contents this way is that we do not rely on any specific implementation feature of an operating system kernel. However, this approach has several disadvantages. First, it is hardware dependent - different video adapters have different protocols for retrieving the contents of video memory. Second, during initialization, the crash kernel outputs debug messages. Hence, the contents of video memory has to be preserved either by the main kernel when the failure is detected or by the crash kernel before printing any information to the screen. Preserving the contents of video memory by the main kernel contradicts our goal of minimizing the amount of code that is executed in the context of the main kernel after a failure is detected. Preserving the contents of the video memory by the crash kernel may not be

always possible because at initialization stage, the operating system may use a different video driver (typically a more basic one) than the one it uses during normal system operation (which is loaded later).

Fortunately, Linux duplicates the contents of video memory inside the kernel in order to support video consoles. Linux supports multiple virtual consoles, each of which can be displayed on the physical terminal screen at any given time. The user can switch between these consoles with the terminal's keyboard. Our current implementation supports resurrection of virtual consoles but only for those in text mode - graphical console resurrection has not yet been not implemented.[4]

Whenever we resurrect a process that outputs to a physical terminal, we replace the contents of the current terminal screen of the new process with the contents of the terminal screen attached to the process being resurrected as it was at the time of the microreboot. The descriptor of the terminal can be retrieved indirectly from the process descriptor of the process. The terminal descriptor references the descriptor of the virtual console, which, in turn, points to a memory area that contains all symbols displayed on the screen along with their attributes. This area is copied from the main kernel memory and is sent to the current terminal driver for displaying on the screen.

For the correct functioning of the resurrected process, it is important to resurrect not only the contents of the screen but also the terminal settings that had been set by the original process. These settings specify, for example, whether the inputs from the terminal are processed by the program on a per character or on a per line basis, whether local echo is enabled, or whether carriage return is processed by the terminal driver or by a program. The terminal descriptor points to the *termios* structure that contains

[4]Resurrection of a graphical user interface is a much more complex task, because the state of Linux's graphical user interface, X-Window, is distributed across several processes that communicate with each other through the sockets interface. In order to resurrect a graphical console, we would first need to be able to support automatic resurrection of Unix domain sockets, named pipes, and automatically resurrect groups of dependent processes running on the same machine. We discuss this in more detail in Chapter 8.

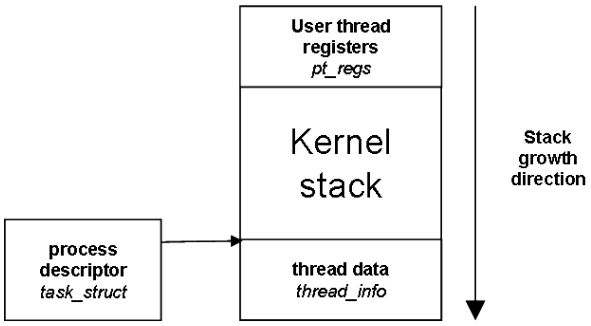

Figure 4.3: Layout of Linux kernel stack

all these terminal settings. Otherworld retrieves this structure from the main kernel and copies its contents to the current terminal's *termios* structure to make the terminal settings of the resurrected process exactly as it was for the original process at the time of the microreboot.

Our console screen resurrection code is hardware independent, but it depends on a specific feature of the Linux kernel, namely duplication of the screen contents in the kernel memory. In order to implement Otherworld with operating system kernels that do not have this feature, we suggest modifying these kernels to maintain the copy of video memory inside the main kernel memory.

4.3.7 Recovering thread contexts

In Linux, when a process is created, a new process descriptor is allocated. It contains data that describes both the process and the main thread of the process. For every additional thread created by the process, another process descriptor is allocated. This additional process descriptor shares most of the resources (e.g., process id, memory descriptor, open file table, console) with the first process descriptor that was created during process creation. Process descriptors of the other threads of the process are organized in a doubly-linked circular list and thus can be retrieved by the crash kernel.

Information unique to a specific thread, is stored in a *thread_info* structure, and the

thread's process descriptor contains a pointer to this structure. This structure is allocated at the bottom end of the thread's kernel stack (Fig. 4.3). The kernel stack has a fixed size, and so the stack's location can be deduced from the location of the *thread_info* structure. When an application thread enters the kernel, the processor registers are saved by the hardware and the kernel code at the top of the stack.

As described in the previous chapter, at the beginning of a microreboot, before passing control to the crash kernel, we ensure that all application threads have entered the kernel, either through a system call or an interrupt, thus saving the processor registers on the kernel stack of each thread. The crash kernel, when resurrecting a process, retrieves and recreates all additional process descriptors. Then, for every process descriptor of each thread of the process being resurrected, Otherworld retrieves the registers from the corresponding kernel stack inside the main kernel memory and puts them at the top end of the kernel stack of the newly created process's thread. When the thread continues its execution after resurrection, its context is as it was at the time the thread was interrupted by the microreboot.

4.3.8 Recovering Futexes

The Linux kernel provides *futexes* (Fast User-space Mutexes) as a building block for user-space synchronization primitives, such as semaphores, conditional variables, and mutexes. Futex is allocated in the user portion of the process's address space and has the semantics of a mutex. In the non-contended case, acquiring a futex is done completely by user-space code. If the user-space code finds that the futex is currently acquired by another thread, it passes control to the kernel, so that the current thread can be put to sleep until the the futex is released. Release of a futex is also done by a user-space code. If upon releasing a futex, the user-space code finds that there are other threads waiting for this futex, it makes a system call telling the kernel to wake these threads. Kernel code always rechecks futex state to avoid race conditions. User-space code does not notify the kernel about

futex creation. When the call is made to the kernel, an address of the user-space futex is passed to the kernel as a way to identify the futex. Upon encountering a futex address for the first time, the kernel automatically creates all necessary data structures, such as list of processes waiting for this futex.

Because futex data is accessed by the kernel as well as applications, the code that deals with futexes depends on kernel version and is implemented in system libraries, such as NPTL (Native Posix Thread Library) that provides Posix threads functionality for Linux. Posix semaphores, conditional variables, and mutexes are implemented in Linux through futexes by this library. Thus, supporting futexes is very important for multi-threaded programs, such as MySQL.

In the non-contended case, Otherworld does not change the way futexes are handled because the kernel is not involved. In the contended case, when the kernel is called by the NPTL library, changes were made to the library code to automatically retry to acquire the resource if the kernel call returns the error code that indicates the kernel microreboot, and the current thread is not marked as the owner of the futex.

Futexes also can be shared between different processes and/or have associated file descriptors to support asynchronous notifications. These features are not currently supported by Otherworld.

4.4 Implementation Limitations

In Table 4.2 we summarize which kernel resources used by applications at the time of a microreboot our current implementation of the crash kernel is able to resurrect automatically and which resources our current implementation does not support.

Our current implementation does not automatically resurrect resources that are used for exchanging data between processes running on the same machine, such as Unix domain sockets, pipes, pseudo terminals, System V semaphores, and futexes shared between two

Resources restored automatically by the crash kernel	**Currently implementation does not restore**
Application physical memory pages	Unix domain sockets
Pages swapped to disk	Pipes
Memory mapped files	Pseudo terminals
Shared memory	System V semaphores
Open files	Futexes[1]
File buffers	
Physical screen content	
Thread execution context	
Signal handlers	
IP, ICMP, and TCP sockets	

[1] Private futexes that are not shared between two or more processes are supported.

Table 4.2: Automatic resource resurrection.

or more processes. Resurrection of this resources would require a mechanism for joint resurrection of group of processes that share these resources. This limitation results in the need to create a crash procedure for the applications that use these resources. We believe these resources are resurrectable and on working on them next.

Another limitation of our implementation is that it relies on applications or application libraries to handle correctly the error code that is returned after the application resurrection by the system call that was executing at the time of microreboot. For most of the system calls, this handling consists of simple reissue of the system call that returned this error code. However, current implementation cannot guarantee that simple reissue will work for certain non-idempotent system calls, as described in Section 3.4.4. Possible generic solutions are described in this section as well but are not implemented in

the current Otherworld prototype. Currently, if an application is not able to recover from system call failure, a crash procedure is required to shutdown and restart the application gracefully without data loss after the resurrection.

The above limitation are not insignificant. But given that writing a crash procedure even for a complex applications is a relatively simple task (as we show in Chapter 7), and given that the alternative is an instant termination of all running applications and loss of all application in-memory state, a best-effort solution has great practical value.

4.5 Summary

This chapter presented implementation details of Otherworld in the Linux kernel. The Otherworld implementation in Linux required adding fewer then 3000 new lines to the stock Linux kernel code and required modifying fewer than 600 lines of existing code. We found that most of the code changes were relatively straightforward. Most changes focused on making the transfer of control from the main to the crash kernel simpler and more reliable. A few changes were made to reduce the amount of main kernel data necessary to be retrieved and analyzed for resurrection.

Our resurrection code relies only on 32 kernel data structure types. The structures are accessible through 4 global variables. Using fewer then two dozen data structure types, Otherworld is able to recreate most of the resource types used by an application so that the application can continue execution after the microreboot.

Chapter 5

Application State Corruption Protection

5.1 Overview

The practicality of microrebooting an operating system kernel depends to a large extent on the probability of the bug that caused the kernel to crash to also corrupt application memory and/or kernel structures needed for recovery. Most faults in the operating system kernel conform to the fail-stop model and cause an immediate crash, leaving application data intact [11, 55, 75, 120]. However, there are some faults that do not result in an immediate operating system crash, thus potentially leading to data corruption. Application data can be corrupted *(i)* when kernel data structures that describe kernel resources owned by application are corrupted or *(ii)* when application data itself is corrupted.

While a kernel microreboot may not be able to completely guarantee protection from memory corruption errors, it should be noted that alternative techniques also cannot provide such guarantees. For example, a fault in the kernel may corrupt application data before a checkpoint is taken, corrupting the checkpoint as well [75]. Verifying data consistency at every checkpointing event would add considerable run-time overhead.

In this chapter, we discuss the probability of kernel and application data structures being corrupted as a result of a kernel fault and present a new mechanism, called application state protection, that significantly reduces the probability of application data being corrupted.

5.2 Probability of State Corruption

In Otherworld, the memory and process management structures are the key kernel data structures required to resurrect a process so that it can save its state or continue execution.[1] In order for an application to be able to save its state after a kernel crash, it is necessary that these structures are not corrupted and that a crash procedure is defined. The memory and process structures may be damaged by *(i)* code containing a bug that manipulates them or *(ii)* by faulty code in some other part of the kernel that performs random (wild) writes. The code that manipulates the memory and process management structures in Linux constitutes less then 3% of the total Linux code. Moreover, the fault rate within the process and memory management code is 2-3 times lower than that of other parts of the kernel [55]. This indicates that only 1% of all non-wild write bugs may potentially corrupt the data structures critical for application resurrection.

The probability of wild writes corrupting kernel structures critical for resurrecting applications is proportional to the size of these structures relative to the size of the entire kernel virtual address space. The full list of the main kernel structures used for resurrecting a Linux process as well as their sizes is given in **Tables B.1** and **B.2** in Appendix B. As we will show in the Evaluation Chapter, the total size of the process and memory management structures necessary for resurrection is less than 0.13% of the virtual address space size even on 32-bit systems. This size will be many orders of magnitude smaller on 64-bit systems.

[1] The key kernel structures are: the process descriptor, the memory descriptor, memory region descriptors, page tables, descriptors of memory mapped files, and swap area descriptors.

OS	**Probability**	**Bug types**	**Sample size**
MVS [120]	6%	Real	Sample of 240 error reports out of 3000
BSD [11]	2%	Real	
SunOS [63]	8%	Artificial	500 injected bugs
Linux [55]	10%	Artificial	35,000 injected bugs
Linux [29]	Application-generic[1]: interactive: 18% non-interactive: 3% Application-specific[2]: interactive: 4% non-interactive: 1%	Artificial	400 injected bugs

[1] *Application-generic* means that the application space is considered to be corrupted when a single bit is changed as a result of the kernel fault.

[2] *Application-specific* means that only the memory regions required by the application for recovering its state are checked for corruption.

Table 5.1: Probability of application data being corrupted by faults in the operating system kernel using real and artificially injected bugs.

The probabilities discussed above give us an upper bound of the probability of kernel data used during resurrection being corrupted. We attempt to confirm this hypothesis later in the Reliability and Performance Evaluation chapter.

The second concern is that a kernel bug may have corrupted application data before crashing the operating system. Several research groups have estimated the probability of application data being corrupted by faults inside the operating system kernel for different operating systems [11, 29, 55, 63, 120]. Their research was based on injecting artificial bugs as well as reintroducing real bugs. We summarize the results in **Table 5.1**. All experiments, but one, show that the probability of application data being corrupted due

to a bug is less than 10%, and one experiment puts this probability at 18%. These numbers are high because these experiments consider the data to be corrupted even if a single bit within the application portion of the address space is modified in error as a result of the kernel fault, even if this bit is never used by the application. As was shown by Chandra and Chen, if we only consider the memory regions that contain data important for saving application state, the probability of application data corruption is 1%-4% [29].

In the next two sections we suggest measures that can be taken to increase the probability of corruption detection and reduce the probability of data corruption.

5.3 Consistency Verification

The probability of kernel data corruption going undetected can be significantly reduced by several simple, but effective, techniques. First, much of the state in the kernel is already duplicated in order to speed-up operations. For example, memory page information in Linux is stored in hardware page tables and in Linux *page* memory structures. Protection bits of every page table entry are duplicated in the corresponding memory region descriptor. Recovery code should verify that duplicated data is still identical. Any detected inconsistencies indicate that data corruption has occurred.

Secondly, many fields in most data structures used for resurrection must follow certain rules. If such fields have been corrupted by a wild write, they will violate these rules with high probability. Below are just a few examples of such rules for process, memory, and memory region descriptor data structures:

- Each process descriptor contains two fields that reference two other process descriptors connecting all process descriptors in a list. This list must be doubly-linked and circular.

- Process descriptors contain a pointer to a *thread_info* data structure, which must

be aligned to a memory page boundary.

- Process descriptors contain an integer field that represents the state of the process and can have only one of 5 possible values.
- Each memory descriptor contains two fields that reference two other memory descriptors, connecting memory descriptors of all processes in a list. This list must be doubly-linked and circular.
- Memory descriptors contain a field that points to a null-terminated list of memory region descriptors. The number of elements in this list must be equal to the number of regions that belong to the process, which stored in the memory descriptor as a separate field.
- The total size of regions allocated by the process is stored in a memory descriptor field and must be equal to the sum of the lengths of all memory regions.
- Memory descriptors contain two fields that must point to certain functions inside the kernel. These fields may have only one of two fixed values.
- Each memory region descriptor has a field that must point back to its memory descriptor.
- The start of a memory region must be less then its end, and the end of a region must be less then the start of the next memory region in the list
- Memory region descriptors contain fields that organize all memory region descriptors that share the same physical page (implementing shared memory) in a doubly-linked circular list. In case where a memory page is not shared, this list contains one item that points to itself.
- If memory region descriptor flags show that it is not mapped to a file, then the field that specifies the file descriptor must be null. If memory region flags show that it

is mapped to a file, then the field that specifies the file descriptor must point to a valid file descriptor in the process's open file table.

Rules such as these exist for most fields used for resurrection. By carefully analyzing data integrity using appropriate rules, kernel data structure corruption can often be detected. Such analysis only happens after a failure and thus does not add overhead to the normal operation of the system. Of course, satisfying these rules does not guarantee that there is no data corruption, but these rules allow us to detect corruption in many cases, without adding any overhead to the normal operation of the system.

As a final technique, one could add checksums or data duplication to the most important data structures, such as process descriptors and memory maps. This would introduce some run-time overhead but would ensure that corruption will not go undetected.

5.4 Application State Protection

As was shown by Chandra and Chen, if we only consider the memory regions that contain data important for saving application state, the probability of application data corruption caused by the faults within the operating system kernel is 1%-4% [29]. This probability can be further reduced by protecting the application portion of the application address space using standard memory protection features.

In most Unix implementations, applications have direct access only to the application portion of the address space, while kernel code has access to both application and kernel portions of the address space. This allows arbitrary kernel code to easily modify application data, even when this was not the developer's intention, but the result of a bug. One possible technique to prevent this from happening in most cases is to run the Unix kernel code in isolation, protecting the memory belonging to the application, similar to the way many microkernels do. There are several ways to achieve this:

1. Leave the Unix address space layout without change, but revoke the write per-

mission for the application portion of the address space on every transition from application to kernel code and restore permissions back when control is returned to the application.

2. Maintain two sets of page tables for each application. One page table set is used for executing kernel code, the other is used for executing application code. Both sets map both the application and kernel portion of the address space, as the stock Linux kernel does, but the page table set used for executing kernel code maps the application portion of the address space as read-only. Page table sets must then be switched on every transition from application to kernel and back.

3. Maintain a completely separate address space for kernel code, with its own page tables. Physical memory pages with application data are not mapped to this address space. This would require an address space switch on every transition between user and kernel code.

There are situations when the kernel legitimately needs to access the application portion of the address space; e.g., to copy results of an I/O operation. The Linux kernel convention is to do so through functions specifically designated for this purpose: *copy_to_user()* and *copy_from_user()*. Currently, these functions do nothing more than verify access rights to avoid invalid memory accesses. For the techniques discussed above, we would need to modify these functions so that they also do the appropriate page table management, e.g., mapping necessary application pages or switching page table sets while reading or writing to application pages.

The overhead of the above techniques results from *(i)* switching or modifying page tables on every system call or interrupt and *(*ii) switching or modifying page tables every time the kernel legitimately needs to read or write to application pages.

The advantage of the first two techniques is that they do not introduce overhead when the kernel needs to read from the application portion of the address space. The

first option requires changing many page table entries on every system call or interrupt. The overhead will be particularly high for applications with a large memory footprint and attendant page table entries. The second option, requires significant changes to the Linux kernel in order to synchronize both page table sets on every page table entry modification. This synchronization also introduces run-time overhead. The third option is relatively easy to implement since Linux already maintains for its own purposes a separate page table set that maps only pages that belong to the kernel [23]. The disadvantage of this method compared to other two is that it requires two address space switches every time the kernel code needs to read from pages that contain application data.

We have chosen the third option for our implementation of application state protection because it is the easiest to implement and because the application portion of the address space is accessed by fewer than half of all Linux system calls and interrupt handlers do not access the application portion of the address space.

Application state protection does not guarantee that application state cannot be corrupted under any circumstances. But with application state protection enabled, application data can be directly corrupted only with complex bugs that disable the protection and then change the application state, behavior more likely to be exhibited by malware, rather than bug. Also, application state may be corrupted indirectly by corrupting kernel data structures. For example, a bug may corrupt a page table entry, and as a result, the corrupted entry will point to a different physical page.

The third technique also improves kernel fault detection, since any attempt by the kernel to directly access the user portion of the address space, bypassing the functions specially designated for this purpose (which is always a bug), will result in an immediate kernel failure and a subsequent microreboot. As we will show in the evaluation section, the cost of the protection is less than 12% of overhead.

5.5 Summary

In this chapter, we have shown that the probability of a kernel fault corrupting structures important for application resurrection is low. In addition, there are numerous consistency checks that can be made in order to detect such corruption.

Application data also can be corrupted by a kernel fault, although the probability of such corruption is estimated to be less than 4%. In order to further reduce this probability, we proposed an application state protection mechanism. This mechanism protects application data from being corrupted by faulty kernel code. The overhead of this mechanism will be evaluated in Chapter 7.

Chapter 6

Application Case Studies

6.1 Overview

In this section, we describe the benefits that Otherworld can provide for certain types of applications and estimate the complexity of writing crash procedures for these applications, should they be needed. We show that even simple crash procedures are able to restore application state of many real-world applications and improve application fault tolerance. We used the Otherworld implementation as described in Chapter 4. We tested Otherworld with five different programs, representing different application classes: the vi and JOE text editors, the MySQL database server, the Apache/PHP web application server, and the BLCR checkpointing solution, used with many scientific applications [57]. The results are summarized in Table 6.1.

In some cases, resurrection required a crash procedure. We found that writing a simple crash procedure that saves application state to disk, restarts the application, and restores saved application state for the applications we considered did not require a deep understanding of application internal details. The applications that we considered all had functions that serialize and deserialize important state to and from a file or a byte stream. The task of writing a simple crash procedure then includes:

Application	**Crash procedure**	**Modified lines of code**
vi	Not required	0
JOE	Not required	1
MySQL	Not required	0
Apache	Required[1]	115
BLCR	Not required	0

[1] Because the Apache server uses the Linux kernel implementation of System V semaphores, Otherworld cannot currently resurrect it without a crash procedure.

Table 6.1: Modifications to the applications to support Otherworld.

1. identifying these functions,

2. adding a crash procedure that calls the serialization function to save application's state to persistent storage and restarts the application,

3. modifying the application startup code to call the deserialization function supplying it with the application state that was stored during the previous step.

We have found that for the purpose of writing a crash procedure it is not necessary to understand the implementation of the serialization/deserialization functions used or the internal format of the data they produce. We were able to easily identify and use such functions within 1-2 days for Apache/PHP and MySql without any prior knowledge of their internal design.

We describe each application we considered in the following sections.

6.2 Interactive Applications

For interactive applications, such as text and graphic editors or computer games, losing state due to a kernel crash results in all work since the last save operation to be lost. While some text editors typically autosave every few minutes, many other programs do not have this feature (e.g., graphic editors such as Photoshop or GIMP), in part because the size of the image being edited or game being played may be tens or even hundreds of megabytes and saving this much data has a significant performance impact. Moreover, we are not aware of any text editor that saves additional application state, such as the undo data, as a part of its autosave function. For such applications, Otherworld's ability to continue application execution after a kernel failure offers significant advantages, since this additional application state is also resurrected.

A text editor is a good example of an application that can be resurrected without having to be modified. We tested Otherworld with two popular text editors: vi and JOE, which are shipped as part of many Linux distributions. JOE, in particular, contains much advanced functionality, such as the support for multiple windows, macro execution and syntax highlighting. Vi did not require any modifications in order to be resurrected, and no document data or application state was lost across kernel failures during our testing. Initially, JOE failed after resurrection because it treated any error code returned by the console read function as a critical error and terminated itself. Changing one line of code to reissue failed console reads allowed JOE to be resurrected without any other modifications, making any kernel crash transparent to the user. In both cases, the applications were able to run across a kernel failure without requiring a crash procedure. After resurrection, the user was presented not only with the latest contents of all documents, but also with the undo buffer, relative window positions and other application state preserved.

6.3 Databases

Another important class of application that can benefit from Otherworld are database management systems. Storing data in RAM instead of disk can improve server performance by up to 140 times [69, 91]. However, a key reason why in-memory databases are problematic is that data is lost when the operating system crashes. Otherworld significantly reduces the risk of data loss due to an operating system crash by preserving the in-memory data tables across kernel crashes. Database server reliability requirements are often very high. In some cases, database server developers may prefer using a crash procedure, as potentially cleaner and more reliable approach, comparing to continuing application execution from the point where it was interrupted by microreboot. By adding a simple crash procedure that saves the contents of the in-memory database to the disk and restarts the database server, we can improve its fault tolerance without introducing runtime overhead or architectural changes.

For our tests, we used MySQL. The MySQL architecture isolates the code responsible for maintaining data at the physical level into a separate component called *pluggable storage engine (PSE)*, which includes low-level functions that store and retrieve data. MySQL supports different types of PSEs. One of these, called MEMORY PSE, stores the table data in memory without saving it to disk, thus making the database memory-resident in order to improve performance.

By examining the MEMORY PSE source code, we found that all tables allocated by MEMORY PSE are organized internally in a linked list, which is accessible through a global variable. MEMORY PSE has functions for scanning and retrieving row data from the tables, returning them in an internal format. Also, it has a function that accepts data in the same format as an argument and inserts it as a new row in the table. For the purposes of writing our crash procedure, we did not need to know how these functions work or the internal format of row data.

Although, it is possible to continue running an unmodified MySQL server after the

```
1  int ow_crash_procedure(unsigned long failed_types) {
2          LIST *pos;
3          for (pos= heap_share_list; pos!=NULL; pos= pos->next)  {
4                  HP_SHARE *info;
5                  byte filename[256];
6                  File f;
7                  byte* record=NULL;
8                  HP_SHARE* share=(HP_SHARE*) pos->data;
9                  sprintf(filename,"%s/%s.ow",restore_dir,share->name);
10                 f=my_open(filename, O_RDWR | O_CREAT, MYF(0));
11                 record=my_malloc(share->reclength,MYF(0));
12                 info=heap_open(share->name, O_RDONLY);
13                 my_write(f,&share->reclength,sizeof(int),MYF(0));
14                 heap_scan_init(info);
15                 while (!(error=heap_scan(info,record)))
16                         my_write(f,record,share->reclength,MYF(0));
17                 my_close(f,MYF(0));
18                 my_free(record,MYF(0));
19         }
20         execv(ow_argv[0],ow_argv);
21         return 1;
22 }
```

Listing 6.1: Simplified crash procedure for MySql server

kernel microreboot, we also tried to estimate the difficulty of writing a simple crash procedure for MySQL. The crash procedure for the MySQL server is shown in Listing 6.1. For the sake of clearness, we omitted all error-handling code from the listing. We marked the functions and data types defined by MySql with bold font. Because the application memory space of the process is fully restored by the crash kernel's resurrection code, the crash procedure can use any function or data structure defined by MySQL. The first element of the list of in-memory tables is pointed by the *heap_share_list* variable, and the crash procedure iterates through this list (line 3). For each table, a separate file for temporarily storing the contents of the table is opened (lines 9-10). Next, the size of a table row is written to the file (line 13). After this, the crash procedure calls the appropriate MySQL functions to retrieve the data rows from the current table (lines 14-15) and save them to the file (line 16) in the directory *restore_dir* created for the purpose of temporarily storing resurrected tables. 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 to disk, it restarts MySQL (line 20), passing it the arguments with which MySql was originally started before the microreboot. Those arguments are passed to the *main()* function at MySQL startup and saved into global array *ow_argv* by the code that we added.

We also modified MySQL to *(i)* check during the startup the *restore_dir* directory for any files saved there previously by the crash procedure, *(ii)* read the content of these files, *(iii)* initialize the in-memory tables with this content, and *(iv)* delete these files after the initialization is complete.

It took us a day to write the crash procedure. Most of this time was spent on getting ourselves familiar with the MySQL architecture, since we did not have any prior experience with this product. Overall, MySQL has about 700,000 lines of code, and our modifications consisted of 70 new and 5 modified lines of code.

6.4 Web Application Servers

While the HTTP protocol is stateless, many web applications need a way to maintain session data, such as the contents of a shopping cart or user credentials, across a sequence of page accesses. Some Web applications need to be fault tolerant, which means that user session data cannot be lost when the system fails. To address this requirement, session data is typically stored on disk or in a database. The copying of session data between in-memory representation and persistent storage causes at least a 25% performance decrease [81]. By adding an Otherworld crash procedure to the Web application server, we can prevent losing session data on kernel failures without the overhead of going to persistent storage. Once a crash procedure is added to the Web application server, no changes are required to any Web application that runs on this server.

For our case study, we selected the Apache and PHP bundle. The session data is stored by the PHP code in shared memory and is available to applications through PHP functions. PHP session code stores session data in a hash table using the session id as a key and a serialized version of the session data as a value. The address of the hash table is stored in a global variable. The crash procedure that we wrote gets the address of the hash table and saves each element of the table to a file. After the session data is saved to disk, Apache restarts and initializes the session data table from the file.

As in the MySQL case, we did not need to know how session data is serialized or the details of the session hash table implementation because we reused the functions that already existed to retrieve and populate the session hash table. As a result, changes to the PHP code were limited to 110 new and 5 modified lines of code. All modifications were limited only to the PHP module code itself, so all PHP applications can benefit from improved fault tolerance without any changes. The logic of Apache/PHP crash procedure completely resembles the logic of the MySQL crash procedure.

6.5 In-memory Checkpointing

A popular mechanism for minimizing the consequences of application and operating system failures is checkpointing. There are several approaches aimed at reducing the overhead of checkpointing by saving checkpoints to memory rather than to a disk. Zheng et al. show that saving checkpoints to memory reduces overhead by more than a factor of ten [136]. However, in-memory checkpointing does not protect from operating system crashes because the memory is overwritten during a traditional system reboot. On the other hand, the advantage of checkpointing is that it does not necessarily require support from the applications. By combining Otherworld with existing checkpointing techniques, we can improve the reliability of in-memory checkpointing by protecting in-memory checkpoints from operating system crashes without changing the applications themselves.

We tested our technique with the Berkeley Labs Check-point-Restart (*BLCR*), a system level checkpoint/restart implementation for Linux used with many scientific applications [57]. BLCR consists of a kernel module and a user-level application that together checkpoint unmodified applications. The BLCR user-level application receives the id of a process for which it need to create a checkpoint, after which it requests the BLCR kernel module for the checkpoint data and writes this data to a disk file.

We modified the BLCR user-level application so that:

- Instead of writing checkpoints to disk, the BLCR application runs continuously and keeps checkpoints in memory.
- Upon user request, the BLCR application restores the target process from the latest in-memory checkpoint.

We measured the performance of in-memory checkpointing against the performance of the unmodified BLCR that writes checkpoints to disk. As long as the checkpoint done by the modified BLCR application fits in physical memory, checkpointing performance

improves approximately by a factor 10 compared to the original BLCR application. This is consistent with the measurements observed previously by other research groups [77, 136].

We were able to successfully recover the BLCR application and with it target application checkpoints from operating system crashes and continue running the target applications from the recovered checkpoints. We did not introduce any modifications apart from modifying BLCR to keep checkpoints in memory. That is, no crash procedure was needed.

6.6 Summary

In this chapter, we have demonstrated how Otherworld can be of use for five different programs as examples that represent popular classes of applications, such as interactive applications, database servers, web application servers, and checkpointing solution to be used with scientific applications.

We gave an example of a crash procedure and showed that the task of creating a crash procedure, even for an application as complex as a database or a web server, is not difficult or time-consuming and requires only basic knowledge of the application internals.

Chapter 7

Reliability and Performance Evaluation

7.1 Overview

The main goal of our Otherworld work is to significantly improve the reliability of computer systems. In this chapter, we address the following questions:

- Recovery: Can Otherworld successfully perform a microreboot responding to a kernel crash?
- Reliability: What is the probability of resurrected application data being corrupted after a kernel crash and subsequent microreboot?
- Performance: What is the performance impact of Otherworld and the application state protection mechanism during normal execution?

In Chapter 5 on protection, we theoretically estimated an upper bound on the probability of kernel and application data structure being corrupted. In this chapter, we experimentally confirm those theoretical estimations using synthetic fault injection techniques. We show that Otherworld can successfully perform microreboots and allow applications

to survive kernel crashes in more than 97% of the cases. Application data corruption was detected only in 1 out of 2,000 cases when application state protection was enabled. Finally, we show that Otherworld does not impose any noticeable performance overhead, while overhead from application state protection is moderate, at less than 12% in the worst case.

7.2 Reliability

In order to test the reliability of Otherworld, we injected faults into the operating system kernel running an application. This injected faults cause the kernel to crash, which in turn induces a microreboot and the resurrection of the application allowing the application to save its state; we then verified the correctness of the saved application state.

We tested the five applications described in the previous chapter: the vi and JOE text editors, the MySQL database server, the Apache/PHP Web application server, and the BLCR in-memory checkpointing system. For each application, we added a crash procedure to save its state to disk after the resurrection so that we can make a comparison between the application state saved by the crash procedure and the application state we would expect if no crash happened. In total, we observed 800 experiments that ended with a kernel crash for each of the five applications. Half of the experiments were done with the standard Linux address space management, and the other half with application state protection enabled, as described in Chapter 5.

7.2.1 Test Methodology

Experimental Setup

We automated our experiments to expedite the gathering of results. In order to simplify the automation of a large number of test runs, we conducted reliability experiments within a VMWare virtual machine. The machine had two virtual CPUs, 1GB of RAM, and 22GB

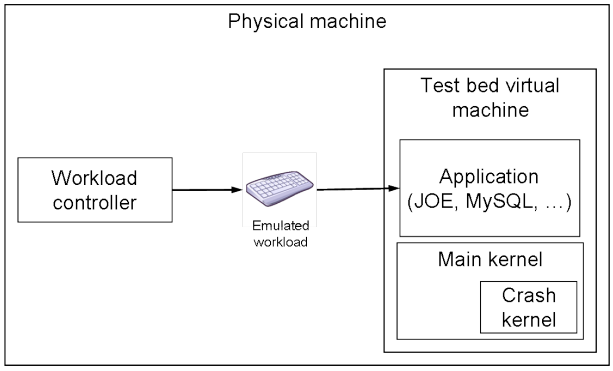

Figure 7.1: Testing environment

of disk storage. We installed the modified Linux 2.6.18 operating system with Otherworld on this virtual machine. The physical machine that hosted the virtual machine also ran a program, called workload controller, that *(i)* directed the workflow of the experiment, by starting a target application with a particular workload on the Linux kernel with Otherworld and then *(ii)* checked the application data after a kernel microreboot to detect for potential corruption (Fig. 7.1). More specifically, for each experiment, the workload controller started the virtual machine and waited for the main kernel to initialize itself and load the crash kernel. Next, the workload controller established an SSH session with the virtual machine and started one of five applications to run during the experiment. Then, the workload generator sent commands to the application modifying its state. The nature of the commands depended on the application being tested and is described in more detail in the next subsection. All issued commands were logged by the workload generator, so that we would independently know the correct state of the application for every point in time.

After a random amount of time ranging, from 1 to 3 minutes after application start, we injected faults into the kernel or kernel modules and observed the outcome. For each experiment we injected 30 faults at a time to increase the chance of generating a failure. Most of the failures occurred within few seconds after the faults injection. About 20% of the experiments did not result in a kernel fault within 5 minutes, and all applications

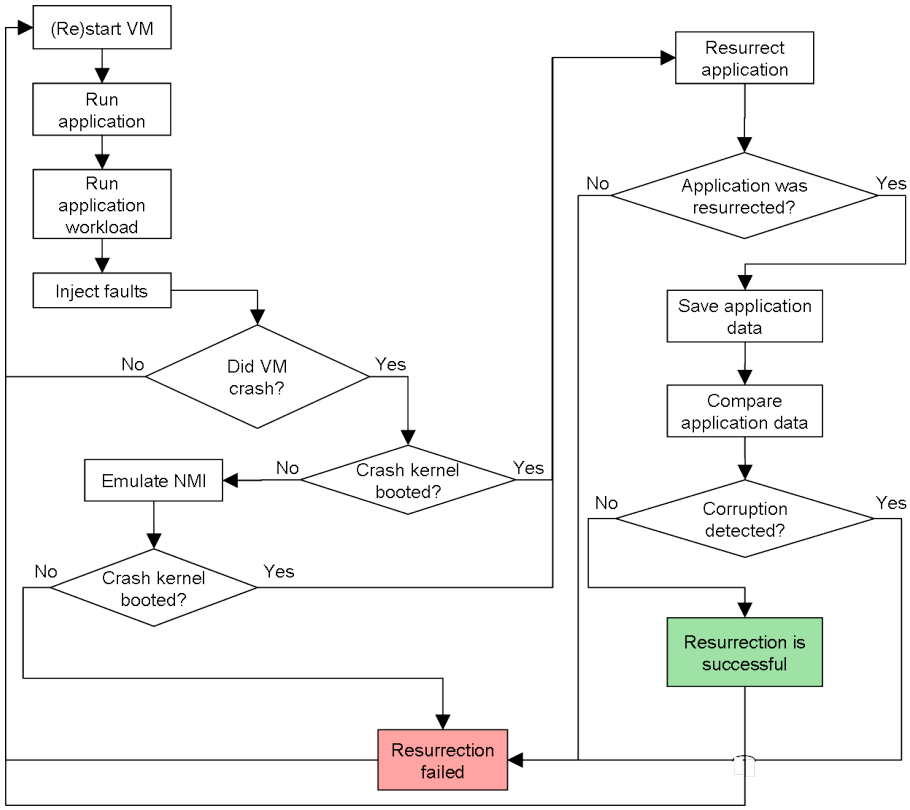

Figure 7.2: Experiment workflow

continued executing with no visible problems. In these cases, we discarded the experiment from our statistics and continued with the next experiment. If the system was unable to perform a microreboot, or the crash kernel failed to resurrect the application because of main kernel memory corruption, we considered resurrection a failure.

After the application was resurrected, it saved its data to disk using a crash procedure designed specifically for this purpose, and the workload controller checked this data against the commands it had sent to the application. If any data was missing or incorrect, we considered resurrection to have failed because of data corruption. Otherwise, we considered resurrection to be a success.

The workflow of the experiments is shown in Figure 7.2.

Applications and Workloads

The vi and JOE workload consisted of replaying a sequence of keystrokes that emulated a working user. After resurrection, the text editors saved the created document to a file, and the workload controller compared the saved document with the sequence of keystrokes it had sent to the editors.

When testing MySQL, the workload controller connected to the running server and issued a sequence of SQL queries which inserted, deleted, and updated entries in the database with numerical and character data. All issued queries were recorded by the workload controller. As soon as one query was completed another was issued. After resurrection, the workload controller reconnected to the MySQL server, retrieved the table data and checked for correctness of the data against the issued queries.

When testing the Apache/PHP web application server, the workload controller continuously requested a PHP page that recorded the number of times it had been requested in the session variable and displayed this number on the generated HTML page. After resurrection, the workload controller continued issuing page requests, checking that the request count embedded in the page is correct.

The BLCR workload consisted of periodic in-memory checkpointing of a test application. The application was specifically written for the purpose of testing. It dynamically allocated 800MB memory array, filled it with sequential numbers, and periodically verified that the array data was not changed. After resurrection, we ensured that the application could be restored from the checkpoint and let the application verify that its data was not corrupted.

Fault Injection

Reliability testing can done either by injecting synthetic faults or by reintroducing real faults to the program and observing the resulting behavior. Ideally, we would perform reliability experiments with real faults, but in practice, real faults usually take too much

Fault type	**Description**
Stack fault	Corruption of a single value on a stack
Interface fault	Corruption of a function argument
Branch fault	Deleting a branch instruction
Condition fault	Inverting a conditional instruction
Instruction fault	Flipping a random bit of an instruction
Source fault	Changing an instruction source register
Destination fault	Changing an instruction destination register
Pointer fault	Changing the address for a memory instruction
NOP fault	Deleting a random instruction

Table 7.1: Synthetic fault types injected into the kernel.

time to be triggered and to crash the system. A typical operating system installation running on commodity hardware crashes on average four times a year [51]. It would take us about 1,000 machine-years to observe 4,000 crashes. Hence, we instead used the synthetic fault injection mechanism originally developed at the University of Michigan for evaluating the reliability of the Rio File Cache [93] and later used for evaluating Nooks reliability [125]. We modified this fault injector mechanism to use it on Linux 2.6 kernels.

The fault injector tries to generate faults that closely resemble real programming errors common for kernel code [38, 120]. There are several types of faults that can be generated. *Stack faults* change a single integer value on the kernel stack of a random thread. *Interface faults* corrupt function arguments. *Branch faults* replace a branch instruction with no-op instruction. *Condition faults* invert the condition in a conditional instruction. *Instruction faults* flip a random bit in a random instruction. *Source faults* change the source register, while *destination faults* change the destination register of a random assignment instruction. *Pointer faults* change the address portion of instructions

Application	Successful resurrection	Failure to boot the crash kernel	Failure to resurrect application	Data corruption with / without application state protected
vi	97.5%	2.5%	0%	0% / 0%
JOE	97.75%	2.25%	0%	0% / 0.25%
MySQL	97.25%	2%	0.5%	0.25% / 0.5%
Apache/PHP	97%	3%	0%	0% / 0%
BLCR	97%	2.75%	0.25%	0% / 0.5%

Table 7.2: Results of fault injection experiments.

that access operands in memory. Finally, *NOP faults* replace a random instruction with a no-op instruction. These faults are summarized in Table 7.1. They emulate many common errors, such as stack corruption, uninitialized variables, incorrect testing conditions, incorrect function parameters, wild writes, and others.

Unfortunately there is no numerical data on prevalence of different fault types; therefore in our experiments, we injected an equal numbers of each fault type.

7.2.2 Results

The results of our fault injection experiments are summarized in Table 7.2. The second column contains the percentage of cases in which Otherworld successfully preserved application data through resurrection after a failure. The third column lists the percentage of cases where Otherworld failed to boot the crash kernel. The fourth column lists the percentage of cases where corruption in the main kernel structures was detected, preventing resurrection. The last column lists the percentage of cases where applications were successfully resurrected, but subsequent data verification detected data corruption. This

Application	**Kernel memory**	**Page tables**
vi	116 KB	60%
JOE	137 KB	61%
MySQL	711 KB	70%
Apache	844 KB	83%
BLCR	941 KB	83%

Table 7.3: Size of the data read by the crash kernel during the resurrection process.

column contains two numbers. The first represents the percentage of cases where application data was corrupted while running with application state protection enabled, while the second is without application state protection. For each application, Otherworld was able to recover application data successfully in 97% or more of the cases.

The major source of resurrection failure is the inability to transfer control from the main kernel to the crash kernel. Although the amount of code involved is minimal, Otherworld still requires coordination between CPUs on multiprocessor systems and is sensitive to the corruption of certain kernel page entries and the interrupt descriptor table. Since the crash kernel is kept uninitialized and is protected by the memory protection hardware, we found that once we succeeded in passing control to the crash kernel, it successfully boots itself in 100% of the cases.

When application state protection was disabled, 5 experiments out of 2,000 (less then 0.3%) ended with application data corruption. We did not encounter any application data corruption for Apache/PHP, but we encountered data corruption on one occasion for JOE and on two occasions for BLCR and MySQL each. Protection of application state, as described in Chapter 5, introduces overhead (measured below) but significantly reduces the probability of undetected corruption. With protection enabled, application corruption was observed only in one MySQL experiment, due to a undetected corruption of a page table entry.

Resurrection failed due to kernel data structure corruption in only 3 cases, out of a total of 2,000 experiments. This result is perhaps to be expected, since the amount of data needed for resurrection from the main kernel is relatively small. Table 7.3 shows the size of the data that the crash kernel needs to read from the main kernel for resurrecting the applications we tested. We found this size to be less then 1 MB for all of the examples considered. The last column of Table 7.3 lists which proportion of the main kernel data structures required to resurrect the process contained page tables, illustrating that page tables constitute the largest portion of the main kernel data retrieved. The amount of memory used for kernel data needed for resurrection relative to size of the virtual address space gives us a rough estimate of the probability of wild writes corrupting data important for resurrection. Even for an application with the largest possible memory footprint on a 32-bit systems - 3 GB, the amount of data retrieved will be approximately 5 MB, which is less than 0.13% of the total address space.

7.3 Performance

In this section, we describe experiments that show the overhead of our Otherworld implementation relative to unmodified Linux. The first type of overhead is the slowdown that applications experience under normal conditions while the system is running with no failures. The second type of overhead is service interruption time; i.e. the time during which recovery takes place and during which user requests cannot be processed.

7.3.1 Application State Protection Overhead

When application state protection is disabled, the current implementation of Otherworld does not execute any extra code unless a crash occurs, except the minimal changes described in Section 3.2, and we did not observe any run-time processor or I/O overhead. Because the unmodified Linux kernel already maintains for its own purposes a separate

page table set that maps kernel pages but does not map application pages, the application state protection also adds only a few extra instructions to execute. However, switching page table sets on each transition from application to kernel and back and every time the kernel needs to read or write to an application's portion of the address space introduces overhead mainly due to TLB flush operations that occur on every page table switch.

In order to estimate the performance impact of protecting user memory space while executing code in the kernel, we ran the MySQL benchmark suite and the Apache benchmarking tool with and without user space protection enabled. MySQL benchmarking suite comes with a MySQL database server and is meant to compare performance of individual database operations, such as inserting, deleting, and updating large amount of data, creating, altering, and dropping tables [88]. The benchmark also includes the Wisconsin database query benchmark [44]. The Apache benchmarking tool comes with the standard Apache source distribution and shows how many requests per second the Apache server is capable of serving [3].

Since the in-memory checkpointing system and the text editors do not have a high rate of system calls, they were not affected by page switching overhead and were not further considered for this evaluation. However, we added the Volano benchmark to our tests [131]. This benchmark simulates a chat server with multiple client sessions. The benchmark creates client connections in groups of 20 and measures how long it takes the clients to take turns broadcasting their messages to the group. At the end of the test, it reports a score as the average number of messages transferred by the server per second. It is a highly parallel and system call intensive application, the type of workload that should be the most sensitive to system call overhead.

In order to get reliable results, all experiments in this section were run on a physical machine with a single dual core CPU, 4GB of RAM and 120GB of disk storage.

The results of these experiments are presented in Table 7.4. The second column of the table shows the system call rate that the benchmark generates on the unmodified

Benchmark	**Increase in TLB misses**	**System call rate (thousands calls per second)**	**Performance overhead**
MySQL	30	22%	3.4%
Apache	40	51%	4.8%
Volano	100	55%	11.6%

Table 7.4: Performance overhead of enabling user memory space protection while executing system calls.

Linux kernel. The third column shows the increase in TLB misses that application space protection introduces compared to the unmodified Linux kernel. Finally, the last column shows the increase in time required to complete the benchmark which runs with application address state protection. As we can see, the overhead of protecting the application state depends on the number of system calls issued by application and ranges from 3%-5% for Apache and MySQL to 11.6% for the Volano benchmark.

7.3.2 Non-idempotent system call handling overhead

Although the current implementation leaves handling of non-idempotent system call issues described in Section 3.4.4 to applications, we estimated the potential overhead of being able to handle such calls automatically in the crash kernel. As described in Section 3.4.4, this can be achieved if, before proceeding with non-idempotent system calls, the main kernel saves the information necessary to revert (rollback) all changes that might affect correctness of application execution if the application reissues the interrupted system call after a kernel microreboot.

The overhead occurs whenever an application running on top of the main kernel executes a non-idempotent system call. We identified 45 non-idempotent system calls, for which rollback information has to be saved, out of total of 317 system calls defined in

the Linux kernel. The overhead depends on information that needs to be saved for each specific system call. However, the relative frequency of system calls differs significantly. We ran the MySQL benchmark, the Apache benchmark, and the Volano benchmark described in the previous section and found that while executing these benchmarks more than 95% of system call invocations were limited to 26 distinct system calls. Out of these 26 system calls, only 5 system calls, namely *read()*, *write()*, *recv()*, *send()*, and *rt_sigprocmask()*, are non-idempotent.

The information that needs to be saved for these system calls in order to to safely reissue them in case they are interrupted by a microreboot is minimal. For the *read()* and *write()* system calls, it is the current file pointer. For the *recv()* and *send()* system calls, when called for a TCP socket, it is the sequence number of the last byte read and written by the application. Also, for the *recv()* system call, the contents of the kernel buffers copied to the application space needs to be preserved as described in Section 3.4.4. For the *rt_sigprocmask()* system call, the old value of the signal mask has to be preserved before proceeding with this system call.

In order to estimate the overhead due to saving the rollback information, we added the code to the main kernel to save the rollback information for the five system calls mentioned above. Then, we run the MySQL, Apache, and Volano benchmarks, measured their execution time, and compared it with the corresponding execution time when running on top of the unmodified Linux kernel. We did not find any measurable overhead caused by the added code.

7.3.3 Service Interruption Time

Finally, we measured the time it takes for the system to recover from a failure while running different workloads. The results are presented in the Table 7.5. The second column of the table contains the time it takes for a system cold start, measured from the time of pressing the power button to the time the workload is operational. The

Application	**Boot time**	**Service interruption time**
shell	64	53
MySQL	71	64
Apache	70	68

Table 7.5: Service interruption time (seconds).

third column contains the time from when the workload is interrupted by a failure to the time the workload is resurrected and operational again. The first row estimates service interruption time for an interactive user and contains the time until the interactive user is presented with the text mode shell without any additional application starting. The second and the third rows show the time until MySQL and Apache are operational, respectively. Since the crash kernel initializes itself after the failure from scratch, the time it takes to boot the crash kernel is comparable to the time of a cold system start excluding the time consumed by the BIOS and boot loader initialization. Both Apache and MySQL resurrection involves calling the crash procedure to save the application data and the restart of the application. Because of this, the resurrection process is longer than a clean application start. The combination of both factors makes the time during which the workload is not operational comparable to that of a full system reboot. Nonetheless, in all test scenarios, Otherworld has a smaller service interruption time than the full system reboot.

The time of microreboot can be reduced even further by using faster hard drives and newer versions of Linux kernels. Replacing the regular 7,200 rpm HDD on which we ran the above experiments with a fast SSD drive that has a sustainable read speed of 210 MB/s reduces the service interruption time by 40%, down to 32-41 seconds. Also, the boot time of Linux distributions has been significantly improved over the last few years. When using the above SSD drive with the latest Ubuntu 10.10 Linux distribution based on 2.6.32 Linux kernel, the time from pressing the power button to presenting a

command prompt to the user is only 15 seconds, compared to 34 seconds for the Linux 2.6.18 kernel, on top of which we have implemented Otherworld. Thus, we expect that using fast SSD drives and porting Otherworld to the latest Linux kernels will reduce the service interruption time to 10-20 seconds.

The service interruption time is a very important characteristic of a fault recovery mechanism. In the next chapter, we will discuss methods of reducing it even further.

7.4 Summary

In this chapter, we presented the results of our experimental evaluation of Otherworld. We measured the reliability of Otherworld and found that it allows applications to successfully survive and preserve data in more than 97% of the cases, which is a substantial reliability improvement. Without application state protection, Otherworld introduces negligible run-time overhead. The additional cost of the optional application state protection mechanism depends on how system call intensive the workload is and ranges from less than 4% for the MySQL server to less than 12% for the Volano benchmark. Also, Otherworld insignificantly reduces service interruption time compared to a full system reboot. We are going to work further on reducing service interruption time.

Overall, Otherworld shows significant potential for improving the reliability of commodity operating systems.

Chapter 8

Conclusion and Future Work

In this work, we have introduced Otherworld, a mechanism that on an operating system kernel failure:

1. microreboots the operating system without clobbering the state of the applications;
2. restores the application processes along with their memory, open and mapped files, signal handler descriptors, physical terminals, and network connections;
3. continues the execution of these processes from the point at which they were interrupted if automatic restoration of all resources used by the application process was successful, or calls an application-defined procedure, giving the application a chance to save application data if not all resources were restored automatically.

Otherworld significantly increases the level of fault tolerance. In the vast majority of cases, the resurrected applications can at minimum preserve their data to disk and restart if they cannot continue their execution across the kernel failure. This is in stark contrast to the current state of affairs, where a kernel failure results in a full system reboot with the loss of all volatile application state.

We implemented Otherworld in Linux with only minor changes to the kernel and existing applications. We tested Otherworld using a variety of applications from different

application types, such as text editors, the database, the web application server, the in-memory checkpointing solutions, and showed that, even with some kernel resources used by applications not being restored, all of the above applications were able to restore their data in more than 97% of kernel faults. Either no changes to the applications were required or the changes were minimal and straightforward. The key benefits of our technique include negligible overhead (or small runtime overhead when application memory state is protected) and a small, fixed-size memory overhead that is independent of the amount of data used by the applications.

Another key element of Otherworld is that it does not depend on a specific operating system architecture. It can be used with existing commercial operating systems with monolithic kernels, such as Windows or BSD Unix, as well as with microkernel operating systems. This fact is crucial for an industry where billions of dollars have been invested in legacy operating systems.

To conclude the thesis, we present a summary of our contributions, lessons that we have learned from our Otherworld implementation, and directions for future research and improving the capabilities of Otherworld.

8.1 Summary of Contributions

Our thesis is that it is not necessary to perform a full system reboot as a response to an operating system kernel crash. Instead, rebooting only the operating system kernel and continuing the execution of the processes that were running at the time of the failure is sufficient in most of the cases. Our contributions include the following:

- We designed and implemented Otherworld, the mechanism that allows an operating system kernel microreboot, without termination of user processes running on top of the kernel.

- We implemented an application state protection mechanism that reduces probabil-

ity of a kernel bug corrupting application data.

- We conducted 4,000 fault-injection experiments that injected 120,000 faults into the operating system kernel. Otherworld was able to prevent application data loss in more than 97% of operating system kernel failures.

- We showed, using examples of different application types, that simple application-defined functions, which we call crash procedures and are invoked after a kernel microreboot, can save application state even if kernel data structures except those that are responsible for memory and process management are corrupted as a result of the kernel fault. Implementing these functions inside applications combined with the Otherworld mechanism significantly increases application fault tolerance with respect to kernel failures.

8.2 Limitations

As described in the Introduction, Otherworld is the best effort approach. Therefore, it is not suitable for mission-critical applications, e.g., nuclear reactor or airplane control software. Also, it is not suitable for systems that cannot tolerate any possibility of data corruption or inconsistency; e.g., systems that process financial transactions.

While, Otherworld, as a best effort approach, does not guarantee recovery, it is able to recover application data in more than 97% of operating system crashes. There are several reasons why Otherworld can fail. First, without having any software layer between the operating system and the hardware, the microreboot triggering code has to be implemented as a part of the kernel itself and, therefore, can be corrupted by a fault in the kernel; if the microreboot code was implemented within a hypervisor, it could be guaranteed to be successful by the hypervisor. We discuss the ways of combining Otherworld with a hypervisor in the Future Work section. Second, the resurrection of a process may fail either because of a corruption or inconsistency of the kernel data required for

the resurrection. Because Otherworld uses only a small portion of the kernel state for resurrection, as we have shown, the probability of encountering corrupted or inconsistent data is low.

Other limitations of Otherworld come from the generic properties of the microreboot and from our specific implementation of the microreboot mechanism. The generic limitations are:

- Otherworld does not protect a system from power failures, hardware failures, or application failures. Other mechanisms can be used to protect from such kind of failures. Uninterruptible power supply can keep the system running long enough to prevent data loss and shutdown it gracefully in case of a power failure. Redundant hardware or hypervisor-based replication techniques, such as VMWare Fault Tolerance [130], can protect the system from hardware failures. Application checkpointing can prevent a loss of data when an application experiences a failure [67].

- Otherworld does not protect from undetected errors. Microreboot is triggered by the operating system itself and, therefore, depends on existing error detection mechanisms. However, in the Future Work section, we discuss the ways to use microreboot proactively, for system rejuvenation.

- Otherworld cannot prevent the kernel or drivers from intentionally executing instructions that corrupt system state.

- Otherworld may not protect from some deterministic faults. For example, if a fault is deterministically triggered by a particular combination of system calls, then the reissuing of the same system call sequence by resurrected applications will trigger the same fault. However, such faults are easy to reproduce and fix; therefore, they are not very common in production systems.

- If a corrupted kernel or driver code stores persistent state (e.g., prints corrupted

document or writes a corrupted data to disk), Otherworld will not be able to correct it.

- The microreboot process does not involve execution of system or device-specific BIOS initialization code. As a result, drivers that upon startup expect devices to be initialized to a specific state by the device-specific BIOS code may fail to start. However, such devices are rare, and the problem may be solved by fixing the corresponding device drivers.

- Otherworld may not continue running correctly some applications that rely on devices to be in a specific state, e.g., sound cards. This problem can be solved by modifying the corresponding device drivers. For example, during its initialization, the sound card device driver might determine that a crash has occurred and read the latest device state from the memory and device registers instead of reinitializing the device.

- Otherworld is not suitable for continuing running applications with real-time requirements. A microreboot takes some time, during which applications are not executing.

- Otherworld leaves a system unprotected from the start of the microreboot process to after all applications have been resurrected, when the crash kernel morphs itself into the main kernel and protects itself with another crash kernel.

In addition to generic limitations of the microreboot approach, our specific kernel microreboot implementation has the following limitations:

- We did not implement resurrection of Unix domain sockets, pipes, pseudo terminals, System V semaphores, and futexes shared by two or more processes. Thus, if an application makes use of any of these resources, a crash procedure is required for

successful resurrection. This was not implemented not because of any limitations of our approach, but only because of time limitations.

- The current implementation of Otherworld lacks a mechanism for automatic reissue of system calls that failed because of the microreboot. Currently, we rely on application code to handle errors returned by system calls or to provide a crash procedure that saves application data and restarts the application.

These limitations of our implementation and methods of overcoming them were discussed in details in Section 4.4.

In this work, we have reused a fault model that models common software errors. This model has been used to evaluate reliability of other fault-recovery techniques, such as Rio file cache and Nooks [33, 125], but there is always a concern on how correctly the artificially injected faults represent the real software faults that cause operating system failures. Our belief that the above fault model covers a wide range of different real-world failures is based on the long history of previous use, a diversity of injected fault types, the random nature of fault injection, and the large number of operating system kernel failures that we observed. A more direct approach to address this concern would be to use real operating system bugs while testing Otherworld, which we did not do.

8.3 Lessons Learned

In this section, we present lessons that we learned from the process of designing and implementing Otherworld and using Otherworld with real-world applications.

8.3.1 Kernel Microreboot

We have found that, although applications and the operating system kernel share the same process memory space, the kernel is logically well isolated from applications to be

rebootable as a separate entity. We have also found that the amount of kernel data required to recreate the kernel state of the processes after a microreboot is significantly smaller than the total amount of memory used by the kernel. This is due to data duplication inside the kernel and because much of the kernel state is not visible to applications and can be recreated from scratch, e.g., statistical information or cross-references between different data structures used to improve access speed to the kernel data in different contexts.

We have also found that an application can be protected from most non-malicious bugs in the kernel by means of memory hardware with a relatively small amount of overhead. Even without protection, kernel bugs have a relatively low probability of corrupting application data, but running a kernel in an isolated memory address space reduces this probability several times without changing the kernel-application interface.

8.3.2 Data Layout

The layout of kernel data affects the ease of application resurrection and probability of data corruption detection. The tree-like structure of the Linux kernel data layout with roots stored in global variables allows Otherworld to parse kernel data after a microreboot starting from few global root points and to retrieve all necessary data from the kernel memory image.

Data redundancy within the operating system kernel allows Otherworld to detect corruption without additional run-time overhead beyond the check at microreboot time. Data redundancy is maintained in order to speed-up or simplify some kernel operations. In addition, there often exist data-specific rules, which data structures must comply with. When data does not comply to these rules or duplicated data is inconsistent, then we have an an overhead-free indicator that signals some form of data corruption.

8.3.3 Data Consistency

The way how access to shared data is synchronized within the operating system kernel directly affects the chances of successful resurrection. Global kernel data has to be accessed from different independent contexts that need to share data between each other, e.g., from interrupts, from the context of user processes or kernel threads. Lock is a common mechanism to ensure that these different contexts cannot access data when it is inconsistent. Whether a particular data structure is consistent or not can be determined by checking the state of the lock that protects this data structure.

The crash kernel can be viewed as another context accessing a main kernel data structure. The crash kernel can expect that the data structure is consistent if the lock that guards this data structure is not held. On the other hand, if the crash kernel detects that the lock is held, there is no guarantee that the corresponding data structure is consistent. Therefore, the fine-grained locking kernel design is crucial for effectiveness of Otherworld. For example, early versions of the Linux kernel had only one lock (called the *Big Kernel Lock*) that allowed only a single thread to execute in kernel space. In this extreme case, it is impossible for the crash kernel to ensure the consistency of the main kernel data structures.

Fine-grained locking not only increases the time when a particular data structure is guaranteed to be consistent, but also allows more precise localization of data inconsistency. Techniques, such as read-copy-update, that never leave data structures inconsistent, reduce the possibility of data being inconsistent even further.

8.3.4 Crash procedures

Writing a simple crash procedure that saves application state to disk and restarts an application not necessarily requires deep understanding of the application. Although application in-memory state can be complex, applications often already have dedicated

functions that save and restore application state, for example, to a byte stream or a file. In this case, the author of a crash procedure only needs to identify such functions and call them from the crash procedure without requiring the exact knowledge of a data layout or a logic behind these functions. These functions existed in all five applications that we have used for testing Otherworld in Chapter 7.

For cases when a crash procedure saves the application state and restarts the application, it may be time and labor consuming to write a crash procedure that parses and saves the application state on its own.

8.4 Future Work

Even our limited prototype makes it feasible for applications to tolerate kernel faults with negligible run-time overhead and indirectly offers new ways to significantly improve performance of some applications, such as databases or checkpointing libraries, by allowing them to keep all their data in memory with significantly reduced risk of losing the data due to an operating system fault. The main directions of our future work are to research different ways of improving the capabilities of the existing prototype and find new applications of our mechanism.

8.4.1 Resurrecting More Resources Types

If not all resources types used by an application can be resurrected by the crash kernel, application resurrection can be performed only if the application defines a crash procedure. This requires additional efforts on behalf of application developers. In addition, crash procedures cannot be added to existing already compiled binary modules. This narrows the applicability of Otherworld and complicates its adoption by the industry. On the other hand, Otherworld allows application execution to continue without calling a crash procedure if all kernel resources used by an application are automatically res-

urrected. Hence, in the future, we intend to add support for automatic resurrection of remaining resource types, such as pipes, System V semaphores, pseudo-terminals, Unix domain sockets, and full support of futexes.

We believe that the resources for local IPC channels, such as pipes, pseudo-terminals, and Unix domain sockets, are resurrectable. For example, pipes are implemented as a circular buffer of memory shared between two or more processes. All accesses to pipe data structures are serialized using a semaphore, and when the semaphore of a pipe structure is not locked, then the structure should be in a consistent state. If the semaphore of a pipe structure is locked, then the structure was being accessed when the kernel failed and one must assume that the structure is in an inconsistent state, preventing resurrection. The amount of time shared IPC data structures are locked depends on application usage; e.g., if two processes start exchanging a lot of data through the pipe, the pipe may be in an inconsistent state for a noticeable amount of time. As a result, we expect that the resurrection of IPC resources resurrection may fail at a higher rate compared to that of private process resources, such as process or memory descriptors.

8.4.2 Detecting Process's Interdependencies

Often, processes running on the same computer depend on and communicate with each other, e.g., one process may wait for another to finish, or two processes can exchange data through a pipe. The current implementation of Otherworld is able to resurrect only one process at a time. This means that resurrection of group of interacting processes will fail because such processes expect other process to be running.

Since inter-process communication is implemented through kernel resources, such as pipes or process identifiers, it should be possible to detect such interconnections by analyzing the main kernel data structures. For example, when Otherworld, while resurrecting a process, finds that an open file descriptor is a pipe, it can check all other processes that were running on the system at the time of the crash for file descriptors that point to the

same pipe. If such processes are found, they should be queued for resurrection as well. Once the first process is resurrected, Otherworld puts it into a sleeping state until all dependent processes are resurrected. After all dependencies are resurrected, all processes are transitioned into a running state. We intend to modify Otherworld to automatically detect process dependencies and collectively resurrect group of dependent processes.

8.4.3 Performance Optimizations

Various performance optimizations can be done to Otherworlds current implementation. For example, when Otherworld resurrects a process, it allocates a new page from the crash kernel's memory for every application page of the process it is trying to resurrect. Then, Otherworld copies the page contents from the main to the crash kernel's memory. Because initially the crash kernel uses only the region of memory reserved for it by the main kernel, the crash kernel is very memory-constrained before it finishes the resurrection process and reclaims the remaining system memory back. Thus, resurrection of a process with a large memory footprint causes a lot of swapping and may be slow. Instead, we plan to change the resurrection code so that the crash kernel reclaims a memory page as soon as it detects that it belongs to the process being resurrected and immediately maps it to the newly created process address space preserving its contents. A similar approach can be used for pages that have been swapped to disk. With these optimizations, the time to resurrect a process's address space will be equal to the time to parse and recreate the memory management structures, which is significantly faster than to have to copy the entire address space.

8.4.4 Reducing the Time of Microreboot

Reducing the time of microreboot is important for minimizing time during which computer system's services are unavailable and for minimizing effects a system crash has on user experience. From our experiments, we found that the boot time of the system

is divided approximately evenly between the kernel and drivers initialization and starting user processes, such as the *init* process, daemons, and applications. Currently, we are considering different methods to reduce the service interruption time after a kernel failure.

First, the exact hardware configuration of the system is available in the main kernel at the time of a crash. If the main kernel and its drivers save this information at a predefined location during their initialization, the crash kernel and its drivers can retrieve it and thus avoid the unnecessary and lengthy process of hardware discovery, reducing the time of the crash kernel initialization.

The second approach we intend to consider is applicable only to systems running in virtual machines. This approach significantly reduces service interruption times by preinitializing the crash kernel and some applications before a failure. We suggest booting the crash kernel in its own virtual machine, which is identical from a configuration perspective to the machine on which the main kernel is running. Machine running the main kernel shares disk storage with the machine running the crash kernel. After booting itself up, the virtual machine with the crash kernel goes to sleep. Upon a crash, the main kernel signals to the hypervisor that it failed, and the hypervisor suspends the failed virtual machine. Next, the hypervisor resumes execution of the crash kernel virtual machine and provides it with access to the main kernel virtual machine's memory. Then, the crash kernel can proceed with application resurrection as we described before. Because the crash kernel was initialized before the failure, it does not need to spend time on its initialization after the failure, reducing the time during which system's services are unavailable.

Although this approach is applicable only to virtual machines, server and desktop virtualization deployments continue to grow. Roughly 30% of all workloads ran on virtual machines in 2009, and predictions show that this number may grow up to 70% in 2011 [39]. Also, this approach increases reliability since the transfer from the main kernel to

the crash kernel is managed not by the already failed main kernel but by the hypervisor that is not affected by the failure (except for the main kernel signaling to the hypervisor that it failed).

8.4.5 Hot Updates

We expect that the reliability of operating systems may well improve in the future. Even then, we believe that Otherworld will still be useful by allowing the kernel to microreboot without terminating running applications. For example, an operating system often becomes sluggish without failing (say, due to a minor memory leak). In this case a microreboot may improve performance. Provided that the kernel microreboot time can be improved, it can be used for fast system rejuvenation, i.e., regular proactive kernel microreboots in order to clean up the system internal state and prevent the occurrence of more severe crash failures and performance degradation.

Otherworld may also be used for hot updates of an operating system running mission critical software that cannot afford restarts, for example, electrical or telecommunication utility applications.

8.4.6 Corruption Detection and Prevention

Finally, an interesting research task is to further investigate ways of efficiently detecting and preventing kernel and application data corruption that might be caused by kernel faults. Creating a list of rules mentioned in the Section 5.3 may significantly increase the probability of main kernel corruption being detected after the microreboot. No additional run-time overhead is introduced during the normal functioning of the main kernel because those rules will be checked by the crash kernel only in case of a main kernel failure. The ultimate goal would be to create at least one rule for every kernel structures data field used during the resurrection process.

In addition to kernel data structures that describe kernel resources used by applica-

tions, the kernel also contains application data, such as as dirty file, network, and pipe buffers. As shown by Chen et al. on the example of file buffers, this data can be protected by memory protection hardware without any overhead [33]. Implementation of the protection scheme proposed by Chen et al. in the context of Otherworld composes an interesting engineering task.

Finally, the most important kernel data structures, where rules are insufficient to detect corruption with high probability, such as page tables, can be protected with signatures or memory hardware protection. This approach will introduce some overhead but will guarantee application data protection against non-malicious bugs inside an operating system kernel. It would protect from the only application state corruption case that we have detected during our reliability evaluation when the application state protection was enabled.

Bibliography

[1] M.J. Accetta, R.V. Baron, W.J. Bolosky, D. B. Golub, R.F. Rashid, A. Tevanian, and Young M. Mach: A new kernel foundation for UNIX development. *Proceedings of the 1986 USENIX Summer Conference*, pages 93–113, 1986.

[2] J. Amor. SLOCCount Web for Debian Etch. http://libresoft.dat.escet.urjc.es/debian-counting/etch/index.php, 2005.

[3] Apache Foundation. Apache HTTP server benchmarking tool. http://httpd.apache.org/docs/2.2/programs/ab.html.

[4] J. Appavoo, M. Auslander, M. Butrico, D. M. da Silva, O. Krieger, M. F. Mergen, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. Experience with K42, an open-source, Linux-compatible, scalable operating system kernel. *IBM Systems Journal*, 44(2):427–440, 2005.

[5] F. Armand. Give a process to your drivers. *Proceedings of the EurOpen Autumn 1991 Conference*, 1991.

[6] M. A. Auslander, D. C. Larkin, and A. L. Scherr. The evolution of the MVS operating system. *IBM Journal of Research and Development*, 25(5):471–482, 1981.

[7] A. Avizienis. *The Methodology of N-Version Programming*. John Wiley and Son, 1995.

[8] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault-tolerance during program execution. *Proceedings 1st IEEE International Computer Science Applications Conference*, 77:149–155, 1977.

[9] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, 2004.

[10] O. Babaoglu. Fault-tolerant computing based on mach. *Operating Systems Review*, 24(1):27–39, 1990.

[11] M. Baker, S. Asami, E. Deprit, J. Ouseterhout, and M. Seltzer. Non-volatile memory for fast, reliable file systems. *Proceedings of the 5th International Confonference on Architectural Support for Programming Languages and Operating Systems*, pages 10–22, 1992.

[12] M. Baker, M. Shah, D. S. H. Rosenthal, M. Roussopoulos, P. Maniatis, T. Giuli, and P. Bungale. A fresh look at the reliability of long-term digital storage. *Proceedings of the 2006 EuroSys Conference*, pages 221–234, 2006.

[13] M. Baker and M. Sullivan. The recovery box: Using fast recovery to provide high availability in the UNIX environment. *Proceedings of the 1992 USENIX Summer Conference*, pages 31–43, 1992.

[14] M. Banatre, Ph. Joubert, Ch. Morin, G. Muller, B. Rochat, and P. Sanchez. Stable transactional memories and fault tolerant architectures. *Proceedings of the 4th Workshop on ACM SIGOPS European Workshop*, pages 1–5, 1990.

[15] J.F. Bartlett. A nonstop kernel. *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 22–29, 1981.

[16] R. Baumann. Soft errors in commercial semiconductor technology: Overview and scaling trends. *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, page 121, 2002.

[17] E. D. Berger and B. G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 158–168, 2006.

[18] B.N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. *ACM SIGOPS Operating Systems Review*, 29(5):267–283, 1995.

[19] E. Biederman. Kexec. http://lwn.net/Articles/15468/, 2002.

[20] A. Bohra, I. Neamtiu, P. Gallard, F. Sultan, and L. Iftode. Remote repair of operating system state using backdoors. *Proceedings of the International Conference on Autonomic Computing*, pages 256–263, 2004.

[21] A. Borg, J. Baumbach, and S. Glazer. A message system supporting fault tolerance. *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, pages 90–99, 1983.

[22] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, 1989.

[23] D. Bovet and M. Cesati. *Understanding the Linux kernel*. O'Reilly & Associates, Inc. Sebastopol, CA, USA, 2006.

[24] T.C. Bressoud. TFT: A software system for application-transparent fault tolerance. *Proceedings of the 28th Symposium on Fault-Tolerant Computing*, pages 128–137, 1998.

[25] T.C. Bressoud and F.B. Schneider. Hypervisor-based fault-tolerance. *ACM Transactions on Computer Systems*, 14(1):80–107, 1996.

[26] P. Brinch-Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241, 1970.

[27] G. Candea and A. Fox. Crash-only software. *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, pages 67–72, 2003.

[28] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot–a technique for cheap recovery. *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 31–44, 2004.

[29] S. Chandra and P. M. Chen. The impact of recovery mechanisms on the likelihood of saving corrupted state. *Proceedings of the 13th International Symposium on Software Reliability Engineering*, pages 91–101, 2002.

[30] K. M. Chandy and C. V. Ramamoorthy. Rollback and recovery strategies for computer programs. *IEEE Transactions on Computers*, 21(6):546–556, 1972.

[31] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault containment for shared-memory multiprocessors. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 12–25, 1995.

[32] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, pages 145–185, 1994.

[33] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio file cache: Surviving operating system crashes. *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 74–83, 1996.

[34] D. Cheriton, M. Malcolm, L. Melen, and G. Sager. Thoth, a portable real-time operating system. *Communications of the ACM*, 22(2):105–115, 1979.

[35] D. R. Cheriton and K .J. Duda. A caching model of operating system kernel functionality. *Proceedings of the 1st Symposium on Operating Systems*, pages 179–193, 1994.

[36] T. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 140–153, 1999.

[37] A. Chou, J. Yang, B. Chelf, S. Hallem, and D.R. Engler. An empirical study of operating system errors. *Proceedings of 18th Symposium on Operating Systems Principles*, pages 73–88, 2001.

[38] J. Christmansson and R. Chillarege. Generation of an error set that emulates software faults based on field data. *Proceedings of the 26th Annual International Symposium on Fault-Tolerant Computing*, pages 304–313, 1996.

[39] Forester Consulting. Virtualization management and trends. http://www.ca.com/files/IndustryAnalystReports/virtual_mgmt_trends_jan2010_-227748.pdf, 2010.

[40] A. L. Cox, K. Mohanram, and S. Rixner. Dependable $\neq$ unaffordable. *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, pages 58–62, 2006.

[41] F. Cristian, B. Dancey, and J. Dehn. Fault-tolerance in air traffic control systems. *ACM Transactions on Computer Systems*, 14(3):265–286, 1996.

[42] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. CuriOS: Improving reliability through operating system structure. *Proceedings of the 8th Symposium on Operating Systems Design and Implementation*, pages 59–72, 2008.

[43] R. Y. de Camargo, A. Goldchleger, F. Kon, and A. Goldman. Checkpointing-based rollback recovery for parallel applications on the integrade grid middleware. *Proceedings of the 2nd Workshop on Middleware for Grid Computing*, pages 35–40, 2004.

[44] David J. DeWitt. *The Wisconsin benchmark: Past, present, and future. In The Jim Gray Benchmark Handbook for Database and Transaction Systems, Second Edition.* Morgan Kaufmann, 1993.

[45] E. N. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

[46] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 39–47, 1992.

[47] D. R. Engler, Kaashoek M. F., and J. O'Toole Jr. Exokernel: an operating system architecture for application-level resource management. *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, 1995.

[48] W. Feng. Making a case for efficient supercomputing. *ACM Queue*, 7:54–64, 2003.

[49] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. *Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure*, 2004.

[50] A. Ganapathi. *Why does Windows crash?* UC Berkeley Technical Report UCB//CSD-05-1393, 2005.

[51] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP kernel crash analysis. *Proceedings of the Large Installation System Administration Conference*, pages 149–159, 2006.

[52] R. Gioiosa, J. C. Sancho, S. Jiang, F. Petrini, and K. Davis. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 9–23, 2005.

[53] V. Goyal, E. W. Biederman, and Nellitheertha H. Kdump, a kexec-based kernel crash dumping mechanism. *Proceedings of the Linux Symposium*, pages 169–181, 2005.

[54] J. Gray. Why do computers stop and what can be done about it? *Proceedings of the Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[55] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang. Characterization of Linux kernel behavior under errors. *Proceedings of the International Conference on Dependable Systems and Networks*, pages 459–468, 2003.

[56] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. *Proceedings of the 11th ACM Symposium on Operating systems principles*, pages 155–162, 1987.

[57] P. Hargrove and J. Duell. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. *Journal of Physics: Conference Series*, 46:494–499, 2006.

[58] G. Hunt, J. R. Larus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, et al. An overview of the Singularity project, 2005.

[59] Intel. Using the Intel ICH family watchdog timer (WDT) application note: Ap-725. http://www.intel.com/design/chipsets/applnots/292273.htm, 2002.

[60] D. Jewett. Integrity s2: a fault-tolerant UNIX platform. *Proceedings of the 21st International Symposium Fault-Tolerant Computing*, pages 512–519, 1991.

[61] D.B. Johnson and W. Zwaenepoel. Sender-based message logging. *The 7th Annual International Symposium on Fault-Tolerant Computing*, 1987.

[62] M.F. Kaashoek, D.R. Engler, G.R. Ganger, H.M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 52–65, 1997.

[63] W. I. Kao, R. K. Iyer, and D. Tang. FINE: a fault injection and monitoring environment for tracing the Unix system behavior under faults. *Software Engineering, IEEE Transactions on*, 19(11):1105–1118, 1993.

[64] J. Kephart and D. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[65] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *Proceedings of 1986 ACM Fall Joint Computer Conference*, pages 1150–1158, 1986.

[66] G. Kroah-Hartman. Driving me nuts - things you never should do in the kernel. *Linux Journal*, 133(5), 2005.

[67] O. Laadan, R. A. Baratto, D. B. Phung, S. Potter, and J. Nieh. DejaView: a Personal Virtual Computer Recorder. *ACM SIGOPS Operating Systems Review*, 41(6):279–292, 2007.

[68] I. Lee and R. K. Iyer. Software dependability in the tandem GUARDIAN system. *Software Engineering IEEE Transactions on*, 21(5):455–467, 1995.

[69] T. Lehman, E. Shekita, and L. Cabrera. An evaluation of the starburst memory-resident storage component. *IEEE Transactions on Knowledge and Data Engineering*, 1992.

[70] A. Lenharth, V. S. Adve, and S. T. King. Recovery domains: an organizing principle for recoverable operating systems. *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, 44(3):49–60, 2009.

[71] J. LeVasseur, V. Uhlig, J. Stoess, and S. Gotz. Unmodified device driver reuse and improved system dependability via virtual machines. *Proceedings of the 2004 Symposium on Operating Systems Design and Implementation*, pages 17–30, 2004.

[72] C. C. J. Li and W. K. Fuchs. CATCH - compiler-assisted techniques for checkpointing. *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, page 213, 1995.

[73] K. Li, J. F. Naughton, and J. S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):874–879, 1994.

[74] J. Liedtke. On microkernel construction. *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 237–250, 1995.

[75] D. E. Lowell, S. Chandra, and P. M. Chen. Exploring failure transparency and the limits of generic recovery. *Proceedings of the 4th Symposium on Operating System Design and Implementation*, pages 289–304, 2000.

[76] D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 92–101, 1997.

[77] D. E. Lowell and P. M. Chen. Discount Checking: Transparent, low-overhead recovery for general applications. *University of Michigan Technical Report CSE-TR-410*, 1998.

[78] Linux man pages. read(2) system call. http://linux.die.net/man/2/read.

[79] P. B. Mark. The sequoia computer: A fault-tolerant tightly-coupled multiprocessor architecture. *Proceedings of the 12th Annual International Symposium on Computer Architecture*, page 232, 1985.

[80] Microsoft. Trustworthy computing. http://www.microsoft.com/mscorp/execmail/2002/07-18twc.mspx, 2002.

[81] Microsoft. Underpinnings of the session state implementation in ASP.NET. http://msdn.microsoft.com/en-us/library/aa479041.aspx, 2003.

[82] Microsoft. How to modify the TCP/IP maximum retransmission timeout. http://support.microsoft.com/kb/170359, 2007.

[83] D. Milojicic, A. Messer, J. Shau, G. Fu, and A. Munoz. Increasing relevance of memory hardware errors: a case for recoverable programming models. *Proceedings of the 9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System*, pages 97–102, 2000.

[84] G. Muller, M. Banatre, N. Peyrouze, and B. Rochat. Lessons from FTM: an experiment in design and implementation of a low-cost fault tolerant system. *IEEE Transactions on Reliability*, 45(2):332–340, 1996.

[85] B. Murphy and N. Davies. System reliability and availability drivers of Tru64 UNIX. *Proceedings 29th International Symposium on Fault-Tolerant Computing*, 1999.

[86] B. Murphy and T. Gent. Measuring system and software reliability using an automated data collection process. *Quality and Reliability Engineering International*, 11(5):341–353, 1995.

[87] B. Murphy and B. Levidow. Windows 2000 dependability. *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, 2000.

[88] MySQL. MySQL benchmark suite. http://dev.mysql.com/doc/refman/5.0/en/mysql-benchmarks.html.

[89] G. Nelson, editor. *Systems rogramming with Modula-3*. Prentice-Hall, Inc., 1991.

[90] NetMarketShare. Operating system market share. http://marketshare.hitslink.com/operating-system-market-share.aspx?qprid=8, 2010.

[91] W. T. Ng. Design and implementation of reliable main memory. *Ph.D. thesis*, 1999.

[92] W. T. Ng, C. M. Aycock, G. Rajamani, and P. M. Chen. Comparing disk and memory's resistance to operating system crashes. *Proceedings of the 7th International Symposium on Software Reliability Engineering*, pages 185–194, 1996.

[93] W. T. Ng and P. M. Chen. The systematic improvement of fault tolerance in the Rio file cache. *Proceedings of the 1999 Symposium on Fault-Tolerant Computing*, pages 76–83, 1999.

[94] W.T. Ng and P. M. Chen. Integrating reliable memory in databases. *The International Journal on Very Large Data Bases*, 7(3):194–204, 1998.

[95] S. Nichols. McAfee antivirus update crashes Windows XP. http://www.securecomputing.net.au/News/172742,mcafee-antivirus-update-crashes-windows-xp.aspx, 2010.

[96] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in superscalar processors. *IEEE Transactions on Reliability*, 51(1):63–75, 2002.

[97] D. A. Patterson. Recovery oriented computing: A new research agenda for a new century. *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, page 223, 2002.

[98] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. *Proceedings of 1995 USENIX Winter Technical Conference*, pages 213–224, 1995.

[99] J. Pool, I. Sin, and Lie D. Relaxed determinism: Making redundant execution on multiprocessors practical. *Proceedings of the 11th Workshop on Hot Topics in Operating Systems*, pages 25–30, 2007.

[100] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating systems transactions. *Proceedings of the 22nd Symposium on Operating Systems Principles*, pages 161–176, 2009.

[101] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON file systems. *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 206–220, 2005.

[102] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies – a safe method to survive software failures. *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 235–248, 2005.

[103] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, pages 243–254, 2005.

[104] RFC 0791. Internet protocol. http://www.faqs.org/rfcs/rfc791.html.

[105] RFC 0793. Transmission control protocol. http://www.faqs.org/rfcs/rfc793.html.

[106] RFC 1122. Requirements for internet hosts - communication layers. http://www.faqs.org/rfcs/rfc1122.html.

[107] RFC 1323. TCP extensions for high performance. http://www.faqs.org/rfcs/rfc1323.html.

[108] RFC 2018. TCP selective acknowledgement option. http://www.faqs.org/rfcs/rfc2018.html.

[109] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee Jr. Enhancing server availability and security through failure-oblivious computing. *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 303–316, 2004.

[110] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The Lam/Mpi checkpoint/restart framework: System-initiated checkpointing. *International Journal of High Performance Computing Applications*, 19(4):479–491, 2005.

[111] M. Satyanarayanan, H.H. Mashburn, P. Kumar, D.C. Steere, and J.J. Kistler. Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems*, 12(1):33–57, 1994.

[112] F. B. Schneider. Byzantine generals in action: implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2):145–154, 1984.

[113] M. D. Schroeder and J. H. Saltzer. A hardware architecture for implementing protection rings. *Communications of the ACM*, 15(3):157–170, 1972.

[114] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 213–227, 1996.

[115] Top 500 Computer Sites. Operating system share for 11/2009. http://top500.org/stats/list/34/os, 2009.

[116] C. Small. A tool for constructing safe extensible C++ systems. *Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies*, pages 13–13, 1997.

[117] C. Small and M. Seltzer. VINO: An integrated platform for operating system and database research. *Harvard University Computer Science Technical Report TR-30*, 94, 1994.

[118] M. F. Spear, T. Roeder, O. Hodson, G. C. Hunt, and S. Levi. Solving the starting problem: Device drivers as self-describing artifacts. *Proceedings of the 2006 EuroSys Conference*, pages 45–57, 2006.

[119] R. E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, 1985.

[120] M. Sullivan and R. Chillarege. Software defects and their impact on system availability: A study of field failures in operating systems. *Proceedings of the 21st International Symposium on Fault-Tolerant Computing*, pages 2–9, 1991.

[121] M. Sullivan and R. Chillarege. A comparison of software defects in database management systems and operating systems. *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 475–484, 1992.

[122] F. Sultan, A. Bohra, I. Neamtiu, and L. Iftode. Nonintrusive remote healing using backdoors. *Proceedings of the 1st Workshop on Algorithms and Architectures for Self-Managing Systems*, 2003.

[123] S. Sundararaman, S. Subramanian, A. Rajimwale, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. M. Swift. Membrane: Operating system support for restartable file systems. *Proceedings of the Usenix Technical Conference*, pages 281–294, 2010.

[124] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering device drivers. *ACM Transactions on Computer Systems*, 24(4):333–360, 2006.

[125] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 23(1):77–110, 2005.

[126] A.S. Tanenbaum, J.N. Herder, and H. Bos. Can we make operating systems reliable and secure? *Computer*, 39(5):44–51, 2006.

[127] D. Tang and R. K. Iyer. Analysis of the VAX/VMS error logs in multicomputer environments – a case study of software dependability. *Proceedings of the 3rd International Symposium on Software Reliability Engineering*, pages 216–226, 1992.

[128] A. Thakur, R.K. Iyer, L. Young, and I. Lee. Analysis of failures in the Tandem NonStop-UX operating system. *Proceedings of the 6th International Symposium on Software Reliability Engineering*, pages 40–50, 1995.

[129] T. Van Vleck. Unix and multics. http://www.multicians.org/unix.html, 1993.

[130] VMWare. Fault tolerance. http://www.vmware.com/files/pdf/VMware-Fault-Tolerance-FT-DS-EN.pdf.

[131] Volano. Volano benchmark. http://www.volano.com/benchmarks.html.

[132] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, 1994.

[133] Y. M. Wang, Y. Huang, K. P. Vo, P. Y. Chung, and C. M. R. Kintala. Checkpointing and its applications. *Symposium on Fault-Tolerant Computing*, pages 22–31, 1995.

[134] J. J. Wylie, M. W. Bigrigg, J. D. Strunk, G. R. Ganger, H. Kiliççöte, and P. K. Khosla. Survivable information storage systems. *Computer*, 33(8):61–68, 2000.

[135] S. Yi, J. Heo, Y. Cho, and J. Hong. Adaptive page-level incremental checkpointing based on expected recovery time. *Proceedings of the 2006 Symposium on Applied Computing*, pages 1472–1476, 2006.

[136] G. Zheng, L. Shi, and L.V. Kalé. FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 93–103, 2004.

[137] Y. Zhou, P. M. Chen, and K. Li. Fast cluster failover using virtual memory-mapped communication. *Proceedings of the 13th international Conference on Supercomputing*, pages 373–382, 1999.

Appendix A

Structures Used by the *otherworld()* System Call

```
1
2  /* Structure for specifying or getting parameters
3  of the process's crash procedure */
4  struct ow_crash_params
5  {
6  /* Address of the crash procedure */
7      int (*ow_crash_proc)(unsigned long failed_types);
8  /* Location of the stack */
9      unsigned long* stack;
10   /* Size of the stack */
11     size_t stack_size;
12 };
13
14 /* Structure for getting information about processes
15  that can be resurrected */
16 struct get_task_params
```

```
17 {
18 /* Number of processes running at the time of the microreboot */
19   int proc_num;
20 /* Array of structures with process information */
21   ow_task* tasks;
22 };
23
24 /* Structure with information about a process
25 that can be resurrected */
26 struct ow_task
27 {
28 /* id of the process */
29   int id;
30 /* name of the process */
31   char name[TASK_NAME_LEN];
32 /* address of the crash procedure */
33   unsigned long crash_proc_address;
34 /* id of the terminal session */
35   unsigned long terminal_id;
36 /* id of the user */
37   unsigned long user_id;
38 };
39
40 /* Structure with process resurrection statistics */
41 struct ow_statistics
42 {
43 /* Bytes read from the main kernel data structures for all resurrections */
44         unsigned long total_kernel_bytes_read;
45 /* Bytes read from the user process memory for all resurrections */
```

```
46          unsigned long total_user_bytes_read;
47 /* Bytes read from the main kernel data structures for the last resurrection *
48          unsigned long call_kernel_bytes_read;
49 /* Bytes read from the user process memory for the last resurrection */
50          unsigned long call_user_bytes_read;
51 } ow_statistics;
```

Listing A.1: Simplified crash procedure for MySql server

Appendix B

List of Kernel Structures Used for Resurrection

Structure	**Purpose**	**Size (bytes)**
Virtual memory		
task_struct	Process descriptor	1312
mm_struct	Memory space descriptor	408
vm_area_struct	Memory region object	84 per region
pgd_t[]	Page directory	4096
pte_t[]	Page tables	up to 4 MB
swap_info_struct	Swap area descriptor	2176
signal_struct	Signal descriptor	340
sighand_struct	Signal handler descriptor	1284

Table B.1: List of kernel structures used for resurrection and their sizes.

Structure	Purpose	Size (bytes)
User thread context		
thread_info	Thread data	56
pt_regs	User thread context	60
File information		
files_struct	Open files information	184
fdtable	Table of open files	48
file[]	Array of open files	4 per file
unsigned long[]	Open file bitmap	1 per 8 files
file	Open file descriptor	148 per file
char[]	File name	Name length
User screen		
tty_struct	State of tty port	1048
tty_driver	tty driver descriptor	46
vc_data	Console descriptor	332
unsigned short[]	Screen buffer	4800
termios	Terminal parameters	36

Table B.2: List of kernel structures used for resurrection and their sizes (Continued).