BUILDING A RELIABLE OPERATING SYSTEM

BY

FRANCIS MANOJ DAVID

B.Tech., Indian Institute of Technology Madras, 2001

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2008

Urbana, Illinois

Doctoral Committee:

Professor Roy H. Campbell, Chair
Professor Ravishankar K. Iyer
Associate Professor Ralph E. Johnson
Assistant Professor Samuel T. King
Ray Essick, Motorola

Abstract

Despite many decades of research, the management of errors in a live operating system remains a challenging problem. This thesis presents CuriOS, an operating system that incorporates several new error management techniques that significantly improve reliability. Errors detected by both hardware and software are signaled using language exception handling mechanisms. Unhandled exceptions do not crash the operating system and are dispatched to recovery routines.

The architecture of CuriOS is influenced by microkernel design principles. Individual operating system services are assigned separate protection domains. This componentization provided by traditional microkernel designs helps confine errors. However, an error that occurs in a microkernel operating system service can potentially result in state corruption and service failure. A simple restart of the failed service is not always the best solution for reliability. Blindly restarting a service which maintains client-related state such as session information results in the loss of this state and affects all clients that were using the service. CuriOS adopts a novel design that uses lightweight distribution, isolation and persistence of client-related state information maintained by operating system services. This helps mitigate the problem of state loss during a restart. This design also achieves inter-client isolation by curtailing error propagation within services.

Fault injection experiments show that it is possible to recover from 87% or more manifested errors in operating system services such as the file system, timer, scheduler and network while maintaining low performance overheads.

To Hina

Acknowledgments

I thank my advisor, Prof. Roy H. Campbell, for his enduring optimism and encouragement during my years at graduate school. It was a wonderful experience learning and working with him. Credit is also due to him for reading and helping revise several other publications in addition to this thesis. The other members of my PhD committee: Prof. Ravishankar Iyer, Prof. Ralph Johnson, Prof. Sam King and Dr. Ray Essick were also extremely helpful with their feedback and guidance.

I owe Jeffrey Carlyle and Ellick Chan a lot of gratitude for working with me through the years in an effective research team. We spent many late nights in the Siebel Center fixing our code, running experiments and writing papers. They are awesome people to work with and are very good friends. Thanks are also due to everyone in the Systems Software Research Group for their support, feedback and encouragement during my research.

My research team was supported by many partners in industry. I am very grateful to DoCoMo Labs USA, Motorola and Texas Instruments for providing us with equipment and funding for our research.

I thank the many staff members in the Department of Computer Science who ensured my smooth progress through the PhD program. Anda Ohlsson and Barb Cicone helped me solve many non-technical issues over the years. Mary Beth Kelly and Lynette Lubben helped sort through all the paperwork required to submit this thesis on time.

My numerous friends in Champaign-Urbana made day-to-day life colorful and fun. And finally, I will always be indebted to my family for their continuing support and love.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

DMA	Direct Memory Access
ECC	Error Correcting Code
FIFO	First In, First Out
HTTP	Hyper Text Transfer Protocol
I/O	Input/Output
IP	Internet Protocol
KB	Kilobyte
MAC	Media Access Control
MB	Megabyte
MMU	Memory Management Unit
NFS	Network File System
NMI	Non Maskable Interrupt
OS	Operating System
QoS	Quality of Service
SSR	Server State Region
TCP	Transmission Control Protocol
TLB	Translation Lookaside Buffer
UDP	User Datagram Protocol

Chapter 1

Introduction

Operating system reliability remains a challenging problem today [1] despite several decades of research [2, 3, 4, 5]. Errors caused by hardware and software faults are a major factor affecting operating system reliability. Hardware faults can arise due to various factors, some of which are aging, temperature, firmware faults, and radiation-induced bit-flips in memory and registers (Single Event Upsets [6]). Software faults (bugs) due to incorrect code are also very common in large and complex operating systems [7].

Errors in a monolithic operating system can easily propagate and corrupt other parts of the system [8, 9], making recovery extremely difficult. Microkernel designs componentize the operating system into servers managed by a minimal kernel. These servers provide services such as the file system, networking and timers. User applications and other operating system components are modeled as clients of these servers. Inter-component error propagation is significantly reduced because, in many microkernel designs, servers usually execute in their own restricted address space similar to user processes [10, 11].

This thesis introduces CuriOS: a microkernel operating system that incorporates several effective error detection and recovery techniques in order to attain high reliability. CuriOS is written in C++ and is based on the Choices object-oriented operating system [12]. It is constructed as a collection of objects representing various operating system components. Several operating system services are encapsulated in special objects, each of which executes in its own restricted address space.

Errors are detected using a combination of techniques. The hardware virtual memory protection system detects invalid memory accesses. Consistency and integrity checks help

detect memory corruption and hardware watchdog timers are used to detect lockup errors. When errors are detected, CuriOS uses the C++ exception handling support as the signaling framework to notify recovery routines written as exception handlers. Language exceptions are automatically created for processor or other hardware exceptions and are dispatched in a manner similar to programatically raised software exceptions. This allows for a unified approach to error management.

Recovery from a microkernel server failure is typically attempted by restarting it. The intuition behind this approach is that reinitializing data structures from scratch by restarting a server usually fixes a transient fault. This is similar to microrebooting [13]. In Minix3 [10], for example, server restarts are performed by the *Reincarnation Server* [1]. If the server managing a printer crashes, it causes a temporary unavailability of the printer until it is restarted. Unfortunately, this approach to recovery does not always work. Many operating system services maintain state related to clients. In such cases, a server restart results in the loss of this state information and affects all clients that depend on the server. For example, a failure of the file system server in Minix3 impacts all clients that were using the file system. Simply restarting the file system server does not prevent errors from occurring in these existing clients. Reads and writes to existing open files cannot be completed because the restarted server cannot recognize the file handles that are presented to it. A similar problem exists with the network server. Thus, while stateless servers (which usually encapsulate device drivers) can be restarted to recover the system, this technique is not applicable for many important operating system services that manage client-related state.

The state loss problem may be addressed by making clients aware of operating system service restarts and incorporating recovery actions in the clients. This can result in increased code complexity. Another possible solution is to provide some form of persistence to the server's client-related state information. This allows a restarted server to continue processing requests from existing clients. Some microkernel operating systems like Cho-

rus and Minix3 support the ability to persist state in memory through restarts; but they do not use this functionality for operating system servers and, currently, only provide it as a service for user applications or device drivers.

Attempts to solve the state loss problem by simply persisting server state across a restart do not address the possible corruption of this state due to error propagation. An error that occurs in an operating system server, like a typical software error, can potentially corrupt any part of its state [14] before being detected. This highlights yet another significant limitation of traditional microkernel systems. While such systems minimize inter-component error propagation, nothing prevents intra-component error propagation.

Checkpointing operating system service state in order to mitigate the effects of error propagation is not a viable solution because rolling back to a consistent system state requires checkpointing of client state as well. Additionally, multiple checkpoints may have to be maintained in order to avoid rolling back to an incorrect state. This may be expensive in terms of memory and performance.

CuriOS significantly minimizes error propagation between as well as within operating system services and recovers failed services transparently to clients. We accomplish this by lightweight distribution, isolation and persistence of client-related state information used by operating system servers. client-related state is stored in client-associated, but client-inaccessible memory and servers are only granted access to this information when servicing a request. Because this state is not associated with the server, it persists after a server restart. This distribution of state is illustrated in figure 1.1. A server failure that occurs when servicing a client can only affect that client and the restarted server can continue to process other requests normally.

Distribution of state information from servers to clients for fault tolerance is not new. Researchers have exploited this technique to improve the reliability of file system services in distributed operating systems such as Sprite [15] and Chorus/MiX [16]. A more widely known example is Sun's stateless Network File System (NFS) [17]. But these designs do

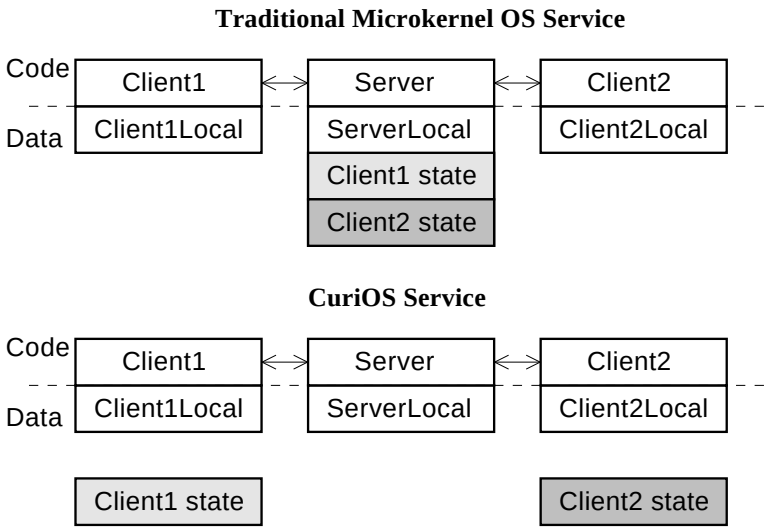

Figure 1.1: State distribution

not protect the state information from being manipulated by clients and leads to various security problems such as those with NFS [18, 19]. Our design supports safe distribution of state by protecting the state from modification by clients. Our implementation is also lightweight because we use virtual memory remapping instead of memory copying to grant access to state. Additionally, we provide a generic framework for implementing distributed state and recovery for any operating system service, not just the file system.

Our work is complementary to other research in operating system error detection such as the language-based type-safety techniques used in SafeDrive [20] and software guards used in the XFI system [21]. Employing such techniques in CuriOS can improve error detection latency and further reduce error propagation.

The primary contributions of this thesis are:

1. Exploration of an exceptions-based framework for unified error management in an object-oriented operating system.

2. Integration of techniques for detecting memory access violation errors and lockup errors with exceptions-based error signaling.

3. Design, implementation and evaluation of a novel restart-based error recovery scheme for stateful operating system components.

CuriOS is being developed to provide a highly reliable operating system environment for mobile devices such as cellular phones powered by an ARM processor. CuriOS currently runs on hardware development platforms based on the Texas Instruments OMAP [22] architecture. The OMAP architecture is a system-on-chip design with an ARM processor core and is marketed for mobile devices. The platform revision that was used for developing CuriOS is the OMAP1610 H2 board which is based on an ARM926EJ-S processor core. CuriOS also runs under the QEMU [23] system emulator which emulates the hardware found in the ARM Integrator family [24] of development boards.

This thesis is the culmination of several years of research, which has also resulted in several publications at various venues [25, 26, 27, 28, 29, 30]. Revised sections of text from these publications are used in this thesis. The dependability related terms used in this thesis conform to the taxonomy suggested by Avizienis et al. [31]. Words that use the font `Class`, represent C++ classes.

The remainder of this thesis is organized as follows. Chapter 2 presents a detailed study of several related operating systems that also have reliability as their goal. A summary at the end of the chapter highlights the properties necessary for transparent operating system recovery. The architecture of CuriOS is presented in Chapter 3. Chapter 4 discusses the exceptions-based error signaling support in CuriOS. The various error detection techniques deployed in CuriOS are described in Chapter 5. Chapter 6 presents the recovery techniques adopted by CuriOS and presents an analysis of the types of errors that are recoverable. Chapter 7 includes a detailed description of several operating system services that were re-written to be restartable. Chapter 8 evaluates CuriOS in terms of its resilience to errors as well as performance and memory overheads. Many design decisions in CuriOS can be categorized into software patterns for reliability. A set of such patterns are highlighted in Chapter 9. While the primary focus of this thesis was on reliability, some of the de-

sign decisions result in improvements to other dependability aspects. Chapter 10 describes the security and maintainability benefits of the CuriOS architecture. Chapter 11 catalogs other related work and Chapter 12 sketches some future directions of this research. Finally, Chapter 13 summarizes and concludes this thesis. Appendix A provides some background information about the Choices operating system and Appendix B presents a brief introduction to the ARM processor architecture.

Chapter 2

Related Operating Systems

Some microkernel operating systems that are closely related to our work are Minix3 [10], L4 [11], Chorus [32] and EROS [33]. We evaluate each of these microkernel-based operating systems by manually injecting memory access errors into different services. This allows us to explore the effect of an operating system error on its reliability. A memory access error is the typical manifestation of a hardware or software fault in an operating system [9]. In all our experiments, a memory access error results in the termination of the operating system server.

Table 2.1 shows the results of our experiments. The immediate effect of encountering the memory access fault is shown in the third column. The last column shows our analysis of whether a restarted server will continue serving existing clients correctly (if restartability support were included in the corresponding operating system). Except for Minix3, which already implements restartable services, this observation is based on purely source code analysis. A brief explanation for our conclusion is provided in each row. The entries in the last column for Minix3 are actual experimental results.

The rest of this chapter discusses reliability aspects of the previously mentioned systems and several other related operating systems in more detail.

2.1 Minix3

Reliability support in Minix3 is provided by the *Reincarnation Server* which is able to restart both failed services and device drivers. Server restarts work well only for device

Table 2.1: Recoverability of microkernel operating systems from memory access errors

μkernel	Failed Server	Immediate Effect	After Restart
Minix3	File System (fs)	System unusable.	× Server is not restarted because the *Reincarnation Server* depends on the file system. Also, all current file system state information is lost.
	Network (inet)	All existing network connections fail.	× Restart does not help re-establish connections because state information is lost.
	Random Numbers (random)	Temporary read failure.	✓ Once the server is restarted, client reads begin working again.
	Printer Driver (printer)	Temporary printer access failure.	✓ Print job completes successfully after spooler retries request to the restarted printer server.
L4	Timer (ig_timer)	System unusable.	× All clients stop receiving timer interrupts. Restart does not help because clients waiting on interrupts can't re-register.
	Name Server (ig_naming)	No immediate effect.	× But many critical services inaccessible because lookup of registered names fail. Restart does not help because all registered clients need to re-register.
	Serial (ig_serial)	Serial port inaccessible.	✓ Request retries will eventually work.
Chorus	File System (vfs)	System unusable.	× Restart does not help because file system state information is lost.
	Network (netinet)	System unusable.	× Restart does not help recover existing network connections.
	Timer (kern)	System unusable.	× Restart does not address clients waiting on timeout.
EROS	Memory allocator (spacebank)	System unusable.	✓ Restore from a previous checkpoint may fix this error.
	Process Creator	System cannot create new processes.	✓ Restore from a previous checkpoint may fix this error.

drivers [1, 34]. This is substantiated by our experiments (Table 2.1). In our experiments, the file system server crashes on all invalid memory accesses and results in an unusable system. Even if the file system server were restarted correctly, existing open files would be inaccessible because of the lost server state. Based on this information, we believe that the Minix3 file system server could be less reliable than the file system in a monolithic operating system such as Linux. This is because a virtual memory error that occurs in Linux file system code only results in the termination of the thread that encountered the error and does not always result in an unusable system.

Minix3 includes a data store server that can be used to store state that persists after a failure induced restart. The Minix3 data store provides some protection from errors in a server because it resides in a separate address space from the server. It has been used to implement failure resilience for device drivers [35]. A drawback of the data store approach is the additional communication and data copying overhead involved. This approach also does not restrict intra-component error propagation.

2.2 L4/Iguana

Iguana [36] is a suite of operating system services that are implemented for the L4 microkernel [11]. This comprises basic operating system services such as naming, memory management, timer and some device drivers. Our experiments study the behavior of some Iguana services when they encounter memory errors. Unlike Minix3, there isn't any support for restartable services. An analysis of the source code shows that server restartability, if implemented, still does not solve the problem of preventing the corruption of state and recovering it. As an example, the Iguana timer service maintains information about clients to which it periodically sends messages. This information will be irrecoverably lost upon a restart. A stateless server like the serial driver, on the other hand, can be restarted and may continue to work for existing clients. More complex functionality such as a file sys-

tem is part of the L4Linux [37] suite, which implements a complete Linux system as a user-mode server. Since most of the functionality required by Linux applications is implemented in this server, the reliability of all L4Linux applications depends on the reliability of this server, which in turn is not any more reliable than the normal monolithic Linux operating system. This has been improved to some extent by isolating device drivers in separate virtual L4Linux servers [38].

2.3 Chorus

The Chorus operating system [32] is designed for high reliability and is used in several telecommunication systems. In contrast to Minix3 and L4, services are executed in privileged mode and share the same address space as the microkernel. Chorus includes "Hot Restart" technology [39] that allows servers to maintain state in persistent memory and resume execution quickly after a failure. Unlike both the design we use in CuriOS and the Minix3 data store, all allocated persistent memory in Chorus is permanently mapped into the server domain. There is no mechanism in place that prevents state information saved in the allocated persistent memory from being potentially corrupted by an error that occurs in a server. Unfortunately, Chorus' operating system services do not take advantage of the "Hot Restart" functionality.

2.4 EROS

EROS [33] is a capability-based system which saves periodic snapshots of the entire machine state to disk. When the system recovers after a crash, the last written snapshot is reloaded. This approach only works when the error is not present in the snapshot. Though the system performs some consistency checks on snapshots, correctness cannot be assured and several previous snapshots may have to be reloaded before a working version is ob-

tained. Minix3's approach of restarting an erroneous server results in a re-creation of all internal state and has better chances of eliminating errors. Another drawback is that snapshots of large systems and device state (not currently performed by EROS) can be expensive in terms of memory and performance. Reverting to a previous system snapshot on a failure also results in a loss of all work done since the snapshot. This may be undesirable in some situations. For example all user input since the last snapshot is lost.

2.5 Others

The Exokernel operating system architecture [40] places most operating system abstractions in an application library and securely multiplexes machine resources. Similar to a monolithic kernel, error propagation is possible throughout the library operating system and the application. There is no mechanism that provides transparent recovery for an application when errors occur in the associated library operating system. An important advantage of the exokernel approach is that errors only affect the process in which they occur. This benefit is at the cost of a complex design for multiplexing shared resources like the storage subsystem. Four design iterations were required to build the XN storage system [41]. The Nemesis OS [42] also adopted a vertical structure similar to the exokernel architecture while providing explicit low-level guarantees for reserved resources. Error propagation was limited by enforcing isolation between device driver, system and application domains. The design of Nemesis was driven by QoS considerations and not surprisingly, does not include recovery support for arbitrary errors in components. However, Nemesis provides QoS isolation between the clients of a system service. Services are designed to prevent one client from adversely affecting the QoS observed by others.

The Singularity system [43] adopts a radically different approach to security and reliability by using software enforcement of address spaces. CuriOS relies on hardware support to enforce memory protection.

2.6 Summary

From our study of the operating systems presented in this chapter, we are able to make several observations about how the design of an operating system can impact its ability to transparently recover in the event of the failure and restart of an operating system service.

Transparency of addressing: Clients should be able to use the same address to access the operating system service after it is restarted. In EROS, since the whole system is restored to a previous checkpoint, this property is true. This is not supported by Chorus, whose hot restart algorithm restarts servers with a new address. Nor is this supported by L4 or Minix3 since a restarted server would be assigned a different address. A name server can be used to ameliorate this problem by maintaining a consistent name for the server across a restart. The restarted server would register its new address with the name server to provide continued availability.

Minix3 achieves transparency of addressing to some degree by using the file system server as a name server. A server can register itself as the handler for a device entry on the file system. For instance the Minix3 *random* server mentioned in table 2.1 handles requests for the `/dev/random` file system node. In our experiment, we opened `/dev/random` using the `open` system call and then proceeded to read a stream of data from the server. When the server crashed, reads using the file handle we obtained when the device was originally opened failed; however, once the server was restarted, reads using the same handle began to work once again.

Suspension of clients for duration of recovery: Clients should not time out or initiate new requests during the recovery phase. This property is supported by Chorus. In Minix3, clients are allowed to run when the server is restarting, and this results in errors when a client attempts to communicate with it. This is also the case in L4; the client will receive

an error when it tries to communicate with a server that may be restarting. The whole system is restored to a previous checkpoint in EROS and therefore, this property is not applicable.

Persistence of client-related state: When a service is restarted, requests from clients must not fail because the server lost client-related state. Client-related state must be preserved and made available to the restarted server. Chorus and Minix3 have some support for in-memory state preservation, but this is not exploited by any of the operating system services they support. An alternative is to save this information to stable storage. In EROS, all computation since the last saved checkpoint is lost.

Isolation of client-related state: Designs of existing microkernel operating systems provide unrestricted access to client-related state within a server. An error that occurs in the server can potentially corrupt state related to all clients. This intra-component error propagation problem exists in a large number of important microkernel operating system services. In EROS, error propagation may lead to inconsistent data being checkpointed.

The design of CuriOS achieves all of the above goals in order to provide transparent recovery of the operating system to applications. Address transparency is achieved by in-place restarts. All client requests are blocked until recovery is completed. Persistence and isolation of client-related state is achieved through distribution and memory protection. This also reduces intra-component error propagation.

Chapter 3

CuriOS Architecture

CuriOS is written in C++ and uses object-oriented techniques throughout its design. It is based on the Choices operating system [12] and inherits many aspects of its software architecture. Unlike Choices which is built as a modular monolithic kernel, CuriOS adopts microkernel design principles for reliability.

This chapter only presents the features of CuriOS that are relevant to this thesis. A comprehensive list of additional features is available as part of the description of Choices in Appendix A.

3.1 Organization

CuriOS is structured as a collection of interacting objects that represent various components and services. An object can be confined to an isolated memory protection domain in order to limit error propagation. We refer to such an object as a *protected object*. Protected objects work together with a small kernel, known as *CuiK*, in order to provide standard operating system services as shown in figure 3.1.

CuiK is a thin layer of the operating system that mediates the interaction between the rest of the system and the hardware. It runs with the highest privileges and works as the "microkernel" for CuriOS. CuiK is composed of a small set of objects that manage low-level architecture specific functionality such as processor configuration, interrupt dispatching and context switching. All communication between applications and protected objects is managed by CuiK.

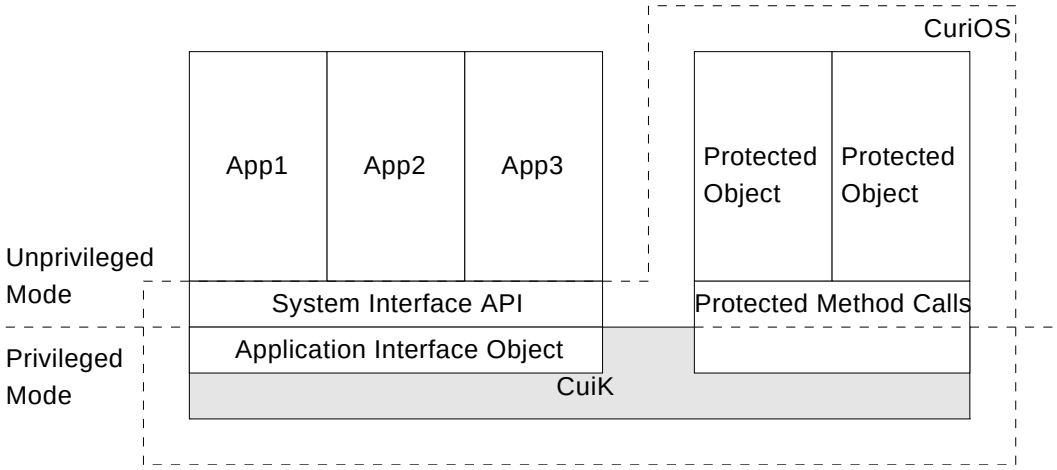

Figure 3.1: CuriOS organization

3.2 Protected Objects

A protected object is the mechanism that provides isolation for operating system components in CuriOS. All methods on a protected object are executed with reduced privileges and run with hardware enforced memory protection. CuriOS applies the principle of least privilege to protected objects and only grants them access to memory regions that are required for correct operation. This prevents an error that occurs while running code in a protected object from corrupting other parts of the system by overwriting memory outside of the protected object.

A protected object in CuriOS is analogous to a “server” in a traditional microkernel system. Thus, operating system servers in CuriOS are implemented using protected objects. Clients are either user applications or other protected objects.

Our implementation of protected objects on the ARM platform enforces restrictions on memory access and completely disallows execution of privileged processor instructions. Nevertheless, because hardware devices on ARM platforms are controlled through memory-mapped registers, they can be made accessible from within protected objects in order to encapsulate device drivers. Implementations of protected objects on other platforms such as the x86 can additionally exploit architectural features to provide fine-grained access

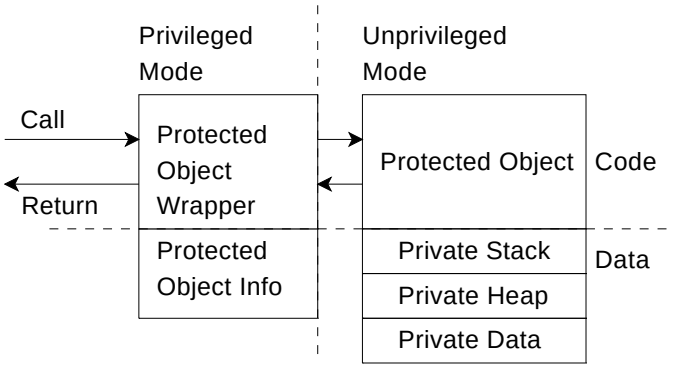

Figure 3.2: Protected method call

control for other resources such as I/O ports.

Our usage of the term protected object should not be confused with the usage of this term in Ada. In Ada, a protected object refers to data objects which are protected by mutual exclusion [44].

Protected method calls are used to invoke operations on protected objects. This is illustrated in figure 3.2. Each protected object is assigned a private heap. A private stack is reserved for every thread that accesses the protected object. This stack is allocated at the first invocation of a protected method, contributing to a small delay in processing the first call to a protected object. Subsequent invocations of protected methods on the same protected object by the same thread reuse this stack and are therefore much faster. A protected method call results in a switch to a reduced privilege execution mode and constrained access rights to memory. The private stack and the heap are mapped in with read-write privileges. The rest of CuriOS is mapped in with read-only privileges. Permissions to write to any additional memory has to be explicitly granted by CuiK.

Protected objects are implemented using a wrapper object that intercepts method calls to a protected object and manages memory access control, processor mode switching and recovery. Figure 3.3 shows pseudo code for this implementation. The `ExampleWrapper` class implements protected object functionality for the `Example` class. The C++ inheritance based construction allows the use of `ExampleWrapper` objects as drop-in re-

```
class Example {
public:
   void runExample() {
     // Code executed in protected object context
   }
};

class ExampleWrapper: public Example {
public:
   void runExample() {
     switchStack();
     switchHeap();
     dropPrivilegeLevel();
     try {
        Example::runExample();
     } catch (Exception e) {
        // Perform recovery actions
     }
     elevatePrivilegeLevel();
     restoreHeap();
     restoreStack();
   }
};
```

Figure 3.3: Pseudo code for the protected object implementation

placements for `Example` objects. The example shown illustrates a simple case where the protected method call has no arguments or return values. If arguments are present, they are passed through to the base class method. If the protected object's method returns a value, it is saved in the wrapper and returned to the calling method. The invocation of the method inside the protected object is enclosed in an exception handling block. If an exception is raised while executing the protected method call, it is intercepted at this point and recovery code is invoked. The recovery process is described in Chapter 6.

Wrapper classes also inherit from a common base class that provides the support functions used to switch protection domains and manage private heaps and stacks. While it is used in the actual CuriOS code, this multiple inheritance is not depicted in figure 3.3.

Every CuriOS thread has an associated memory allocator and uses it for any memory

allocation requests. Normally, all kernel threads would share the same memory allocator for the global kernel heap. The private protected object heap is implemented by temporarily changing the allocator for the thread executing the protected method call. The old allocator for the thread is restored in the wrapper when the protected method call completes.

The protected object wrapper tracks all protected method calls made by a thread by maintaining a stack of activation records for that thread. It uses this information to identify recursive or multiple calls into the same protected object. This is necessary in order to identify the correct stack pointer that should be assigned at any particular protected method call invocation. This is important because only one stack in the protected object is allocated per-thread and all recursive invocations need to reuse the same stack.

The combination of protected objects and CuiK results in a single address space operating system [45, 46], where virtual addresses are identical across various components, but access permissions differ. This design is beneficial in several ways. Pointers are always valid across operating system components and can be easily transferred between them. This minimizes data copying overheads. When switching between components with different access permissions, only the TLB (which maintains access permissions) needs to be flushed; virtually tagged caches do not have to be cleaned and flushed because contents remain valid across the protection boundaries.

3.3 Thread Model

Threads in CuriOS are managed by CuiK. Using defined interfaces, a thread executing in CuriOS can cross application, kernel, and protected object boundaries. For example, a system call from an application causes the thread to cross from user space into the CuiK kernel. This same thread can cross from CuiK into a protected object using a protected method call. Some example threads are illustrated in figure 3.4. This is reminiscent of the design of Brevix [47] and that of Clouds [48].

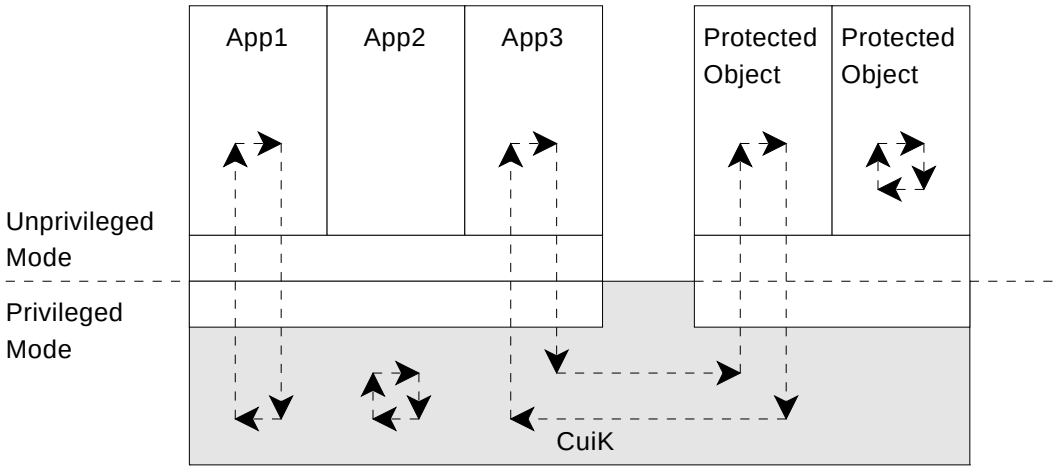

Figure 3.4: CuriOS threads

CuriOS servers can be multi-threaded and our current implementation shares the same virtual memory mappings for all threads that execute within a protected object. An extension of this model to provide thread-private memory could provide additional protection against errors. This direction requires further study and investigation.

3.4 Error Management

The management of errors in an operating system comprises error detection, error signaling and error recovery. CuriOS sports innovations in all three areas.

Error detection is the first step towards ensuring correct system operation in the presence of faults. In software systems, it is extremely important to detect errors quickly and respond with appropriate recovery actions before they cause cascading data corruption and make recovery more difficult. The time between the occurrence of an error and its detection is referred to as the *error latency*. Reducing error latency helps reduce error propagation. The errors detected by CuriOS are described in detail in Chapter 5.

C++ exception handling is used as the error signaling mechanism in CuriOS [25] and is discussed in Chapter 4. Exceptions are raised for both processor signaled errors such

as invalid memory accesses and for externally signaled errors such as operating system infinite loop lockups (signaled by a watchdog timer) [28].

CuriOS restarts components that encounter errors in order to avoid complete system failure. This error recovery mechanism and the associated state management support for restarting stateful components are covered in Chapter 6.

Chapter 4

Error Signaling

Exceptions are events that disrupt the normal execution flow of a program. Languages like Java and C++ provide constructs for programmers to write code to both generate exceptions and handle them. In C++, an exception is generated using the `throw` keyword. The `catch` keyword is used to define a code block that handles exceptions.

Exceptions have several advantages [49] over traditional mechanisms for conveying errors. The use of exception handling allows software developers to avoid return value overloading and clearly separate error handling code from regular code. Using error codes to signal error conditions requires either ugly global variables or propagation of the codes through the call stack, sometimes through methods that do not care about them. Exceptions, on the other hand, are directly dispatched to methods that have handler code. When exceptions are expressed using objects, class hierarchies can be used to classify and group error conditions.

In CuriOS, exceptions are generated for errors detected by both hardware and software. This use of language exceptions to represent arbitrary error conditions results in a uniform framework for managing operating system errors. This empowers developers to use C++ `catch` statements to handle both normally thrown exceptions as well as special error conditions such as bad memory accesses.

While exceptions can be raised for many unexpected conditions in order to attempt recovery, it may be necessary to halt the system in some rare cases where the programmer has determined that recovery may be impossible. For such cases, CuriOS provides a method that stops all computation.

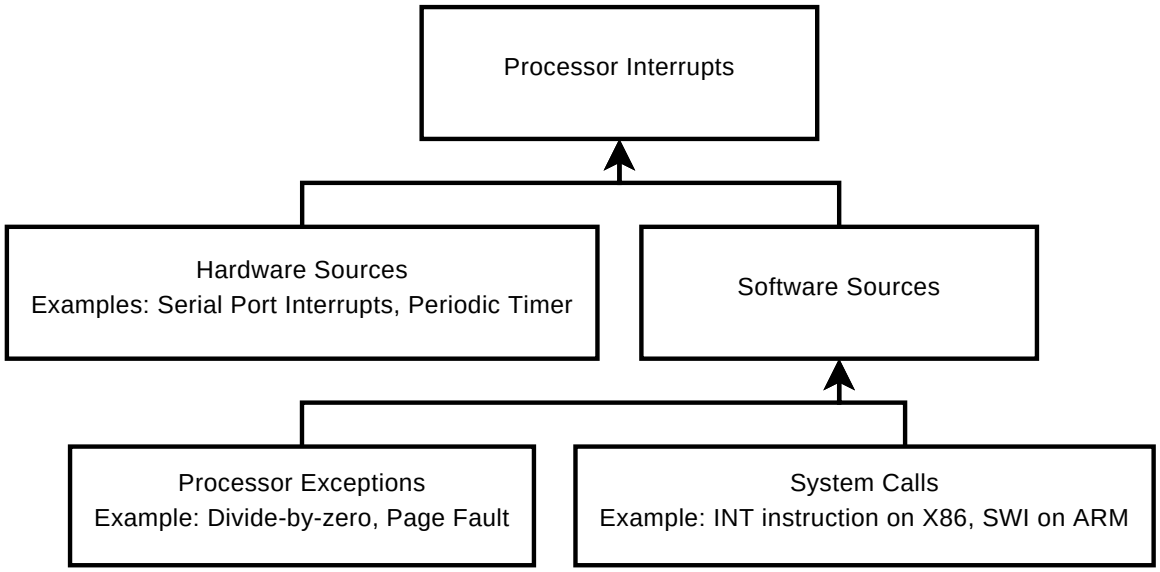

Figure 4.1: Terminology

4.1 Creating C++ Exceptions from Processor Exceptions

In this section, we describe the process by which errors signaled by the processor are converted to C++ language exceptions. Before we delve into details, it is important to understand the various sources of processor interrupts. As shown in figure 4.1, processor interrupts can arise from both hardware and software sources. Peripheral devices can use hardware interrupts to communicate asynchronous events to the processor. For instance, a serial port may interrupt the processor to indicate it has received data.

In addition to hardware sources of interrupts, the software running on a processor may also directly cause an interrupt. Software sources of interrupts can be classified into two categories. Processors generally provide a means for invoking an interrupt via the instruction set. The x86 architecture provides the INT instruction, and the ARM architecture provides the SWI instruction. This mechanism is used to implement system calls. The other category is processor exceptions. A processor exception indicates that some sort of "exceptional" event has occurred during execution. There are many causes of processor exceptions, and possible processor exceptions vary between architectures. Example pro-

```
class Interrupt {
public:
 virtual void raise() = 0;
};

class InterruptManager {
public:
 Interrupt *setInterrupt(int number, Interrupt *);
 Interrupt *getInterrupt(int number);
 void dispatchInterrupt(int number, Context *context);
 void interruptReceived(Context *context);
 void interruptReceived(int number, Context *context);

 virtual void enableInterrupts() = 0;
 virtual void disableInterrupts() = 0;
 virtual void enableInterrupt(int number) = 0;
 virtual void disableInterrupt(int number) = 0;

protected:
 virtual int getInterruptNumber() = 0;
 virtual void ackInterrupt(int number) = 0;

 Interrupt **interrupts;
};
```

Figure 4.2: Interrupt management interfaces in CuriOS

cessor exceptions include division by zero, execution of an undefined instruction, and page faults. It is important to note that not all processor exceptions indicate errors. For instance, page faults are usually handled by the operating system and only result in an error if the faulting process has no valid mapping for the requested page.

Interrupts in CuriOS are supervised by an `InterruptManager` object. This object manages subsystem initialization, interrupt handler registration and interrupt dispatch. Hardware interrupts, software interrupts and processor exceptions are all dispatched using the same interface. Processor interrupts are first delivered to the `InterruptManager`. The `InterruptManager` then dispatches the interrupt to a pre-registered handler. The `InterruptManager` is a machine independent abstract C++ class and its interface is

```
void throwException() {
  // Throw saved exception object
  throw thisProcess()->getException();
}
```

Figure 4.3: Code for the function that throws the exception

shown in figure 4.2. It only includes code for handler registration and dispatch, and it delegates hardware initialization and interrupt number lookup to machine specific subclasses. Interrupt handlers are created as instances of classes derived from the `Interrupt` base class which exports an abstract `raise()` method. The `dispatchInterrupt()` method in `InterruptManager` looks up the handler for the current interrupt and invokes the `raise()` method.

CuriOS registers special handlers with the `InterruptManager` for all processor exceptions like invalid instructions or memory access errors that require the generation of language exceptions. When the processor is handling an interrupt, the context of the interrupted process is available in a `Context` object associated with the `Process`. If the interrupt is due to a processor exception that signals an error, the registered special handler is invoked. The handler creates an appropriate `Exception` object and associates it with the `Process` object representing the currently running process. The `Exception` object includes a saved copy of the current context and a stack backtrace. This information is useful for debugging. The handler then updates the program counter (PC) in the `Context` of the interrupted process to point to a function that throws the exception (`throwException()` in figure 4.3). The special processor exception handler now returns, and the context of the interrupted process (with the modified PC) is restored to the processor. This causes the process to immediately start executing `throwException()`.

The code in `throwException()` extracts the saved exception from the `Process` object and raises the exception using the normal C++ `throw` syntax. Thus, it appears as if the process called a function throwing a C++ exception at the exact instruction address

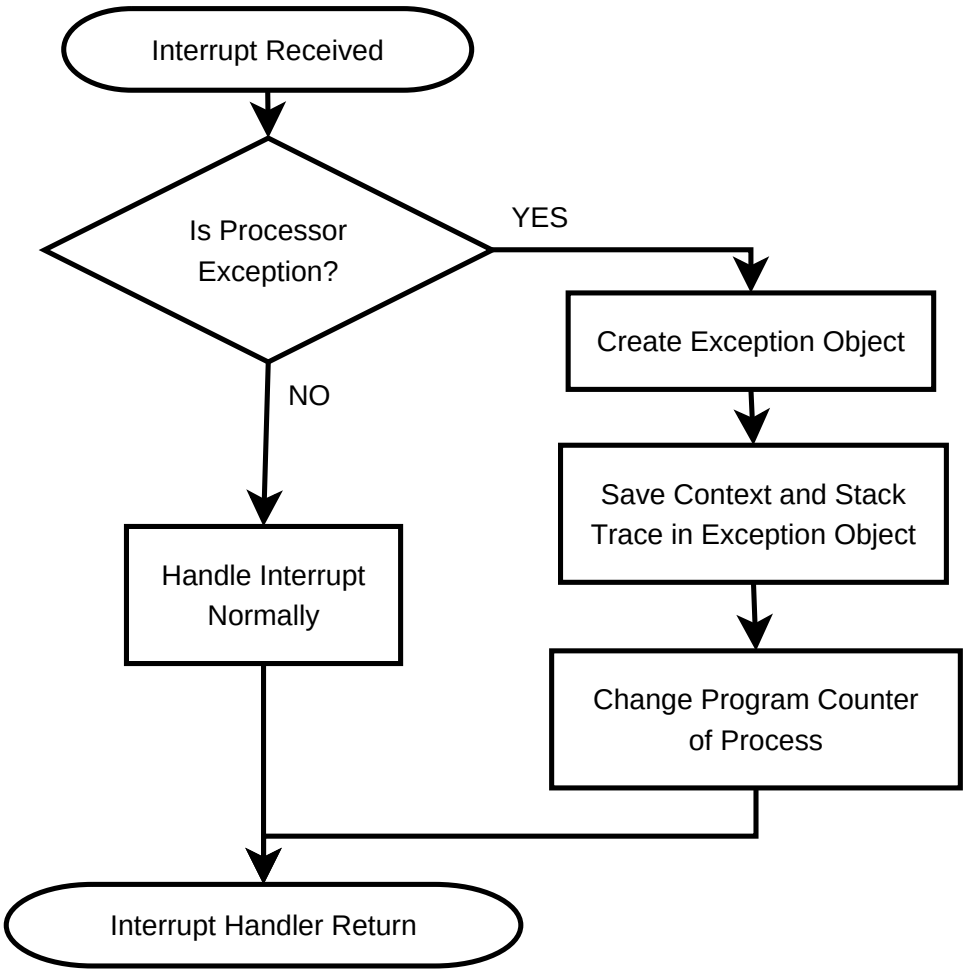

Figure 4.4: Processor exception control flow

where it encountered a processor exception. Figure 4.4 illustrates the flow of control in the interrupt handler for a processor exception. This implementation enables the conversion of processor exceptions to C++ language exceptions.

Another implementation option that was considered was directly modifying the processor exception causing instruction and replacing it with a call to `throwException()`. The idea was to execute the calling instruction on returning from the interrupt handler. On processors with separate instruction and data caches, this technique would require a cache flush in order to ensure that the modified instruction is correctly fetched when execution is resumed. But this design is not safe when there are multiple processes that share the

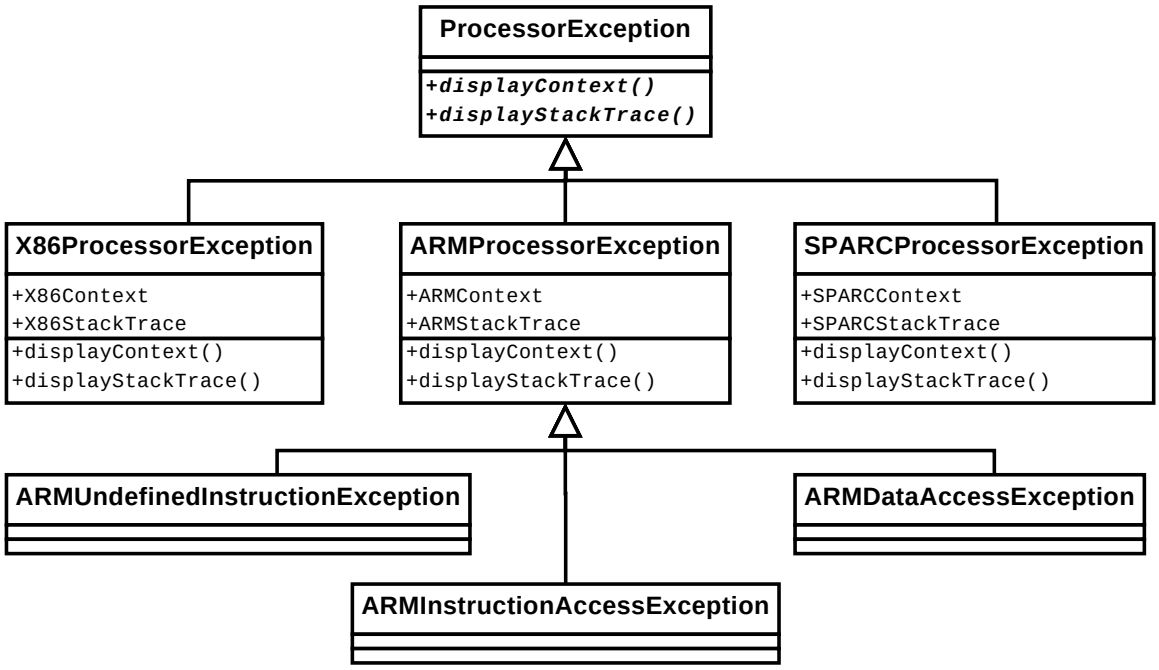

Figure 4.5: Processor exception classes

same code. This implementation was therefore discarded in favor of the PC modifying technique.

Since processor exceptions may occur anywhere in the code, the stack unwinding libraries must be prepared to handle exceptions within any context. This functionality is enabled by a special GNU g++ compiler flag, "-fnon-call-exceptions". Without the use of this flag, only explicitly thrown exceptions are handled. This option is typically used to compile user application code that attempts to throw exceptions from signal handlers. In the kernel, this allows an exception to be correctly dispatched even in the absence of an explicit throw call.

Unlike x86 processors, which support a large set of processor exceptions such as divide-by-zero, protection faults, device unavailable, invalid opcode, alignment checks and so on, ARM processors only classify processor exceptions into three basic types. The C++ exception class hierarchy in CuriOS for the ARM processor is shown in figure 4.5. An `ARMDataAccessException` is thrown when the processor encounters an error while

trying to fetch data. This is the result of a page fault when virtual memory is enabled. An `ARMInstructionAccessException` is thrown when the processor encounters an error while trying to fetch an instruction. This is also called a prefetch abort. An `ARMUndefinedInstructionException` is thrown when the instruction is invalid. This is equivalent to the Invalid OpCode exception for x86. All processor exceptions are derived from an abstract base class called `ProcessorException`. This base class declares abstract methods used to access the context and stack trace of processor-specific exceptions. These are implemented by processor-specific subclasses. This classification allows processor-specific code to respond differently to different types of exceptions. The use of the abstract base class allows machine and processor independent code to catch all processor exceptions independent of the architecture and handle them using generic mechanisms.

4.2 Cross Domain Exceptions

An exception can propagate normally within a single protection domain. However, extra support is required to allow an exception to traverse the multiple domains created by the protected objects in CuriOS. This is necessary because threads in CuriOS cross protected object boundaries which use private stacks and the C++ exception dispatch libraries only expect a single function call stack.

In order to support exceptions that need to propagate across domains, the wrappers that implement protected objects catch all exceptions in the callee domain and re-throw them in the calling domain. This cross-domain support is used for exceptions that don't require the restart-based recovery support. The exception re-throw mechanism is also used when multiple recovery attempts fail and escalation is required.

Because of the single address space design of CuriOS, objects allocated in the callee domain are also visible in the calling domain. Thus, an exception created within a protected

object domain is readable by all other operating system components. There is no need for copying of exception objects between domains.

Unlike other systems like Java that support cross domain exceptions, there is no need for serialization or deserialization of the exception objects. There exist no network or language representation issues that force the need for a serialized representation of objects.

4.3 Undispatchable Exceptions

The C++ language runtime exception handling support can only dispatch an exception correctly when the register context is not significantly corrupted. In particular, invalid values in the FP or SP registers that describe the stack cause the exception dispatching code to fail. In such cases, the language runtime is programmed to invoke an abort routine. Thus, errors that result in corruption of these registers cannot be immediately signaled using exceptions.

CuriOS implements a workaround to handle these undispatchable exceptions. The register context is saved by the protected object wrapper at every entry point into a protected object. The abort routine called by the exception dispatching code is programmed to restore the previously saved context and re-raise the exception in this context. Because the restored register context is valid, the exception can now be correctly dispatched down the remainder of the stack. This process is illustrated in figure 4.6. This approach to dealing with undispatchable exceptions may possibly result in skipping over several stack frames in the protected object by the exception handling system. This is clearly illustrated in the figure. This does not impact recovery significantly because the recovery approach adopted in CuriOS is a restart of the protected object. The restart is expected to reset most of the state information that would have normally been cleaned up by the exception dispatch through the stack frames in the protected object.

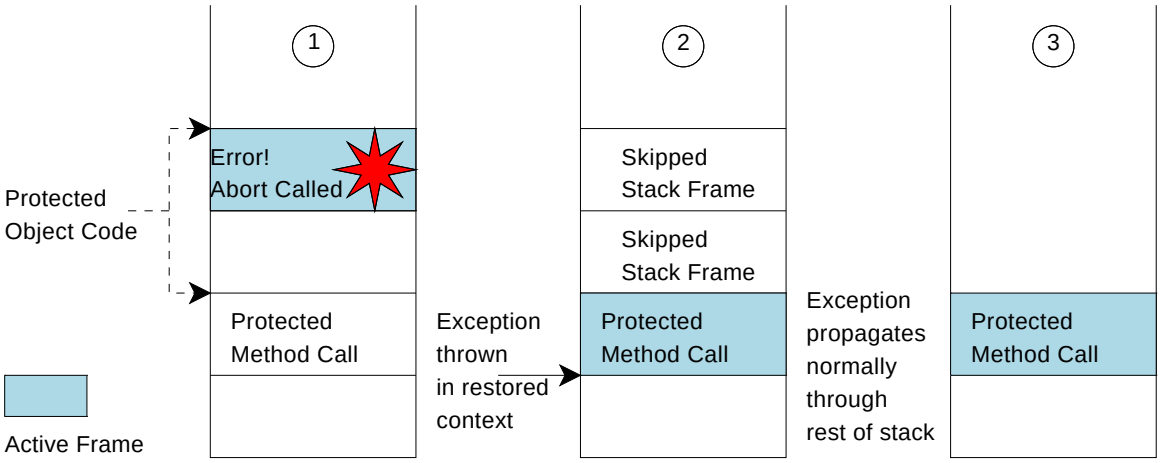

Figure 4.6: Handling of undispatchable exceptions

4.4 Size and Performance Impact

The size and performance impact of using exceptions depends on the implementation of exception handling in the C++ compiler. The GNU g++ compiler supports two different implementations for C++ exceptions. The first implementation is extremely portable [50] and is based on the `setjmp`/`longjmp` (SJLJ) pair of functions. This implements exceptions using a `setjmp` call to save context at C++ try blocks and in all functions with local objects that need to be destroyed when an exception unwinds stack frames. The `longjmp` call is used to restore saved contexts during stack unwinding. Since the context saves are performed irrespective of whether an exception is eventually raised or not, this implementation suffers a performance penalty. For example, this penalty is observed when the `new` operator is used to allocate memory from the heap. The standard library version of `new` is designed to throw a "bad_alloc" exception if the allocation fails. This design results in a context save at the beginning of every memory allocation request. Using SJLJ exceptions in an operating system also requires special precautions because the implementation uses state variables that need to be updated when switching contexts.

The DWARF2 [51] debugging information format specification allows for a table-

Table 4.1: Comparing SJLJ and table-driven implementations of exceptions

Characteristic	SJLJ	Table-driven
Portability	Portable	Not portable
Normal execution performance	Affected because of frequent context saves	Not affected because tables are computed at compile time
Exception handling performance	Fast because context restore is cheap	Slower because of table-based unwinding
Space overhead	Code to save contexts	Table entries for unwinding

driven alternative implementation of exceptions that only executes extra code when an exception is actually thrown. There is no performance overhead during normal execution. This approach uses a static compiler-generated table which defines actions that need to be performed when an exception is thrown. Table 4.1 compares these two implementations. The trade-off in this approach is size for performance. Modern compilers implement exceptions using the table-driven approach for better performance.

We have studied the exception handling framework under both SJLJ and table-driven implementations. To test SJLJ exceptions, we build an ARM compiler from the published GNU g++ source code and enable SJLJ exceptions as part of the build process. For table-driven exceptions, we use a version of the GNU g++ compiler published by Codesourcery [52] that implements table-driven exceptions and also conforms to the ARM Exception Handling ABI [53]. We use the same version (4.1.0) of both these compilers.

Adding exception handling support results in a larger operating system kernel. Researchers have reported kernel size increases of about 10% when adding exception handling to the Linux kernel [54]. Similar numbers for CuriOS are unavailable because the GNU g++ compiler does not allow disabling of exception handling support for C++ code.

We performed a comparison study of CuriOS kernels using SJLJ exceptions with kernels using table-driven exceptions. For each of these implementations, we build two versions of the kernel. One version only supports normal explicitly thrown exceptions. The

Table 4.2: Section sizes (in bytes) for different exception handling implementations

CuriOS	SJLJ		Table-driven	
ELF section	Normal	Processor exceptions	Normal	Processor exceptions
.text	1,252,980	1,296,176	1,063,600	1,066,484
.data	29,056	29,056	28,500	28,500
.bss	297,984	297,984	297,740	297,740
exception data	9,476	10,980	117,364	131,284
everything else	275,868	288,152	272,044	274,512
Total	1,865,364	1,922,348	1,779,248	1,798,520

other version includes support for mapping processor exceptions to language exceptions by using the "-fnon-call-exceptions" compiler flag. An early version of the CuriOS kernel source code was used for these experiments. All kernel executables were compiled to the ELF format. The g++ optimization level was set to 2. We measured the size of the .text section which holds program instructions, the .data section which holds program data, the .bss section which holds uninitialized data and the size of the sections holding exception handling data, such as tables and indices. Table 4.2 shows the results of our measurements. The sizes are also displayed in graphical form in figure 4.7.

The figure shows that the use of SJLJ exceptions increases the size of the program text area compared to table-driven exceptions. This is because of the extra instructions for saving and restoring context that are inserted into all functions that define local objects. A small portion of the kernel (0.5%) is reserved for data that is used during exception handling. Adding support for mapping processor exceptions results in just a 3% increase in the size of the kernel. This increase is due to some extra exception handling code and data.

The kernel compiled with table-driven exceptions is about 4% smaller in size. There is a significant reduction in the size of the program text compared to SJLJ exceptions due to the elimination of extra instructions for saving context. But this reduction in size is offset by the large number of exception table entries. It is possible to reduce this overhead using table compression [55]. When processor exceptions support is added, the size of the kernel

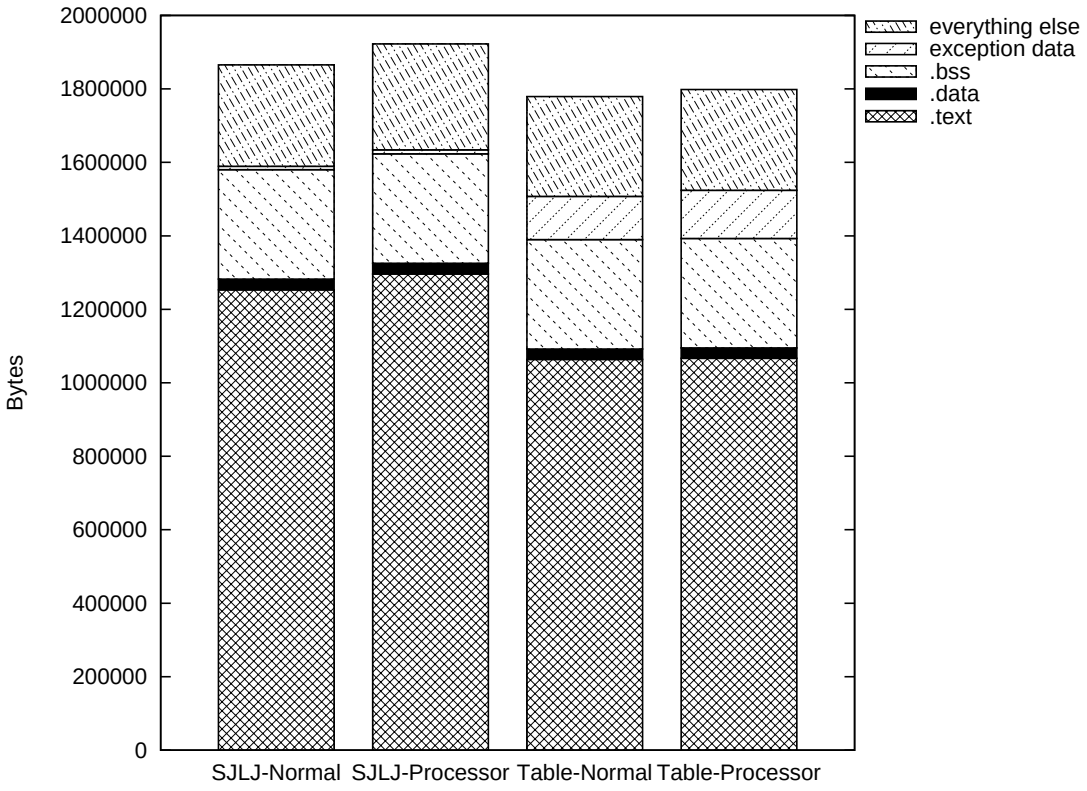

Figure 4.7: Size comparison of CuriOS using different exception handling mechanisms

only increases by 1%.

The performance impact of using exception handling was studied on the Texas Instruments OMAP1610 H2 hardware development kit with the processor clock frequency set to 96 MHz. The overhead of using a single try block in a function was measured to be 14 nanoseconds for table-driven exceptions. This is less than the time taken to run two hardware instructions on the processor. Thus, table-driven exceptions have no noticeable impact on performance. For SJLJ exceptions, this overhead was measured to be 833 nanoseconds. This reflects the significant number of extra instructions that are executed by this implementation to support exception handling. Because of its low performance overhead, we use table-driven exceptions by default for CuriOS builds. All experiments described in the remainder of this thesis use table-driven exceptions.

Chapter 5

Error Detection

CuriOS includes support for detecting three classes of errors: invalid memory accesses, memory corruption and lockup errors. This Chapter describes the techniques used to detect these errors.

The techniques presented in this Chapter are not comprehensive and do not detect all possible errors. However, we believe that the error detection mechanisms implemented in CuriOS are a good match for our primary error recovery approach - component restarts, and are powerful enough to detect a majority of errors. The results of our experiments presented later in Chapter 8 support our intuition. Nevertheless, the design of CuriOS doesn't preclude the addition of other error detection methods, if necessary. For example, the addition of stronger language-based checks as used in SafeDrive [20] or software guards as used in XFI [21] will significantly reduce error latency and may improve recoverability. Detected errors are usually translated into exceptions by CuriOS and are raised in the context of the thread that encountered the error.

5.1 Invalid Memory Access Errors

The memory protection enforced by the hardware on a protected object helps detect any erroneous computation that leads to incorrect memory accesses outside the protection domain. Invalid accesses such as disallowed writes or bad addresses manifest as processor exceptions and are automatically converted to software exceptions using the technique described in Chapter 4. Memory access control also helps constrain error propagation.

In addition to controlling read and write permissions, some processors also have the ability to control execute permissions. This support is available in newer ARM processors and could be used to increase the effectiveness of the protected object mechanism. The processors in our current platforms do not support this feature.

It is important to realize that the protected object mechanism does not detect cases where an incorrect memory access occurs within the protected object's domain. Such errors may be detected by approaches such as SafeDrive [20] or OKE [56].

5.2 Memory Corruption Errors

Operating system memory corruption may occur due to various factors. Some examples of faults that cause memory corruption are radiation-induced bit-flips, software bugs and misbehaving hardware. ECC memory is a widely used defense against flipped memory bits. Commonly available ECC memory modules are able to detect and correct single bit errors in memory words. ECC memory, however, does not address corruption caused by incorrect program execution. These errors may only be detected by checking the consistency or integrity of memory contents.

5.2.1 Data Consistency Checks

Data consistency checks are a standard approach to dealing with memory corruption errors. We use such checks in the CuriOS code whenever possible. Sanity checking code is also available in most existing systems and libraries. When adding third-party libraries to enhance the functionality of CuriOS, such code can be used for error detection. For example, when the lwIP networking library was ported to CuriOS, the "assert" code in lwIP was converted to generate exceptions and signal errors instead of halting execution. For such purposes, CuriOS provides an *AssertOrThrow* construct that throws an exception if the assertion fails.

An obvious limitation of this approach is that it assumes semantic knowledge about the data in memory. It, therefore, requires specialized checker routines written by operating system developers. Nevertheless, the EROS system [33] uses such an approach to verify the consistency of periodic system-wide checkpoints.

5.2.2 Corrupted Instructions

Transient memory faults such as flipped bits or memory corruption because of faulty code can cause errors such as invalid instructions in system code. Unlike other types of errors, corrupted instructions are easy to fix once detected. Recovery simply involves reloading the instruction from stable storage such as disk or other non-volatile memory such as flash.

In some cases, because of hardware problems, the fault may be permanent and a bit in the memory word may be stuck at a particular value. These cases may be distinguished by testing the memory word's ability to store different bit patterns. It might still be possible to recover from such permanent faults by remapping the affected hardware page using virtual memory support.

In CuriOS, if the processor signals an undefined instruction exception, the low-level processor exception handler reloads the instruction from a copy of the code in memory-mapped persistent flash storage. When the handler returns, the newly loaded correct instruction is executed. This recovery strategy is simple to implement; but, it cannot detect memory corruption that results in a machine instruction changing to another valid instruction. Another limitation of this technique is that it cannot be transparently applied to code that is generated at run-time.

5.2.3 Checksums

Checksums are popularly used to verify data integrity. CuriOS can be configured to compute periodic checksums of memory that is not expected to change, such as kernel code. If

the checksum changes due to memory corruption, the affected memory block is reloaded from flash storage. The instruction cache is then flushed to ensure that any corrupted instructions in the cache are discarded. Unlike the undefined instruction reload technique, checksums can detect a valid instruction changing to another valid but incorrect instruction.

CuriOS does not use exceptions to signal a checksum failure. The checksum routine directly transfers control to a recovery routine to fix the corrupted memory. This is possible because unlike other types of error detectors, detection using checksums is asynchronous and recovery does not require specialized thread-local handlers.

This is a preemptive approach and can detect faults before they cause errors. A code checksum may also be performed immediately after other types of operating system errors are detected in order to ensure that system and recovery code is intact. While CuriOS currently only uses checksums to protect code, this technique may also be used to detect changes to static data.

The use of checksums to ensure integrity of code or data has several limitations. It is not feasible to use periodic checksums to protect dynamic data from corruption. Because these checksums are only performed at preset intervals, the error detection latency is high and it is possible for an error to propagate and worsen before being detected. Checksums also incur computational overhead.

Recent ARM based microprocessor designs [57] for mobile devices include Run Time Integrity Checker (RTIC) hardware which can be configured by the operating system to periodically compute and verify SHA-1 hashes of specified code sections. If a hash value changes, it is communicated to the processor through an interrupt. This design significantly reduces the performance cost of performing periodic checksums as the external hardware only utilizes the memory bus when it is idle. Our current development platforms do not include this latest hardware and we are therefore constrained to use checksums performed by software running on the primary processor.

5.3 Lockup Errors

Lockup errors are characterized by an unresponsive kernel that is unable to schedule and run useful code. Some examples of such errors are: infinite-loops, indefinite wait for non-responsive hardware and deadlock conditions. Lockup errors that occur in user space programs can be detected by other programs [58, 59], and can be usually handled without affecting other unrelated programs. On the other hand, lockup errors that occur inside the operating system can render the computer unusable by not allowing any other programs to execute. Lockup causing bugs occur often in operating system code. More than 30% of the bugs in Linux discovered by Chou et al. [7] were bugs that could potentially cause a lockup.

A lockup that occurs with interrupts enabled is called a soft-lockup. Some soft lockups can render the system unusable by permanently preempting all other threads. Such soft lockup errors can be detected by software. The Linux kernel, for example, has a built-in soft-lockup detector. A low system priority watchdog thread that wakes up every second and touches a timestamp is spawned when the system boots. The soft lockup detector is driven by the timer interrupt and checks to see if the timestamp is within 10 seconds of the current time. This check confirms that the watchdog thread is periodically scheduled. If it detects that the thread has not been scheduled for more than 10 seconds, it displays a message reporting the lockup error and records it in the system logs.

The soft lockup detector cannot detect lockups that occur when interrupts are disabled because the detector code is not executed. These errors are called hard lockups and can only be detected by an external observer. Hardware watchdog timers are normally used for this purpose. These timers are normally wired to the reset pin on the processor. They work by requiring the operating system to periodically restart the timer in order to signal the health of the software. If the operating system is in a locked up state, the timer expires, restarting the system.

Table 5.1: Effectiveness of lockup detectors

Kernel type	Lockup location	Soft lockup detector	Watchdog timer
Non-preemptable	Interruptible code	Yes	Yes
	Non-interruptible code	No	Yes
Preemptable	Interruptible code	Yes	No
	Non-interruptible code	No	Yes

Quoting Murphy et al. [60]: The process of restarting the watchdog timer's counter is sometimes called "kicking the dog." The appropriate visual metaphor is that of a man being attacked by a vicious dog. If he keeps kicking the dog, it can't ever bite him. But he must keep kicking the dog at regular intervals to avoid a bite. Similarly, the software must restart the watchdog timer at a regular rate, or risk being restarted.

Microsoft Windows, Linux and most other operating systems configure watchdog timers to reset the processor when they expire and thus trigger a reboot in order to recover the computer. While this improves overall system availability, it unfortunately results in the loss of all running user programs and data. CuriOS, on the other hand, possesses the surprising ability to recover the system even after a processor reset issued by a watchdog timer. The key observation that allows this approach to system recovery is the fact that the reset signal only resets the processor and leaves volatile memory intact. Information loss is limited to the contents of the processor at the time of the reset and the contents of volatile memory can be used to continue running the operating system.

Despite their popularity, hardware watchdog timers are not the complete solution to lockup detection. The preemptability of the kernel and the location of a lockup error influence whether or not the lockup will be detected by an external watchdog timer. Table 5.1 catalogs the detectability of lockups for a non-preemptable and a preemptable kernel. Software solutions are the only viable detection tool for lockups in interruptible and preemptable kernel code. These lockups don't impede progress of other threads, but nevertheless cause the processor to perform computation that is not useful when the locked up thread is

scheduled. Detection of such lockup errors is usually achieved by monitoring the progress of individual threads in the system [61]. CuriOS is a fully preemptable kernel and currently only supports hardware watchdog timer based lockup detection. Watchdog kicks are issued from the timer interrupt handler and non-interruptible hard lockups are always detected. The design of a soft lockup detector for CuriOS is a promising direction for extending this research.

On some platforms, it is possible to wire watchdog timers to a non-maskable interrupt. This can be used to signal the operating system directly instead of forcibly restarting it. Unfortunately, the ARMv5 processors in our target platforms do not include support for non-maskable interrupts.

5.3.1 Creating Lockup Exceptions

Instead of rebooting, CuriOS is designed to recover when a lockup condition results in a processor reset issued by the watchdog timer. CuriOS revives the computer system from the contents of volatile memory and raises a C++ exception in the context of the locked up thread. The following paragraphs describe the process of recovering from a processor reset and dispatching a lockup exception.

When the ARM processor is reset, it sets its program counter to address 0x00000000 and the reset reason (watchdog or power-on) is logged in a peripheral register. The CPSR is reset to place the processor in a privileged execution mode. The signal also resets the interrupt controller and all interrupts are turned off. The memory management unit (MMU) is turned off and thus only physical addressing is possible. Address 0x00000000 is normally the start address of the boot loader in flash memory. The boot loader's job is to initialize the memory hardware and load the operating system kernel into RAM from flash memory or secondary storage. It then relinquishes control to the operating system. The boot loader usually does not differentiate between resets attributed to watchdog timers and power-on resets. Thus, the operating system is always reloaded and rebooted, causing a loss of all

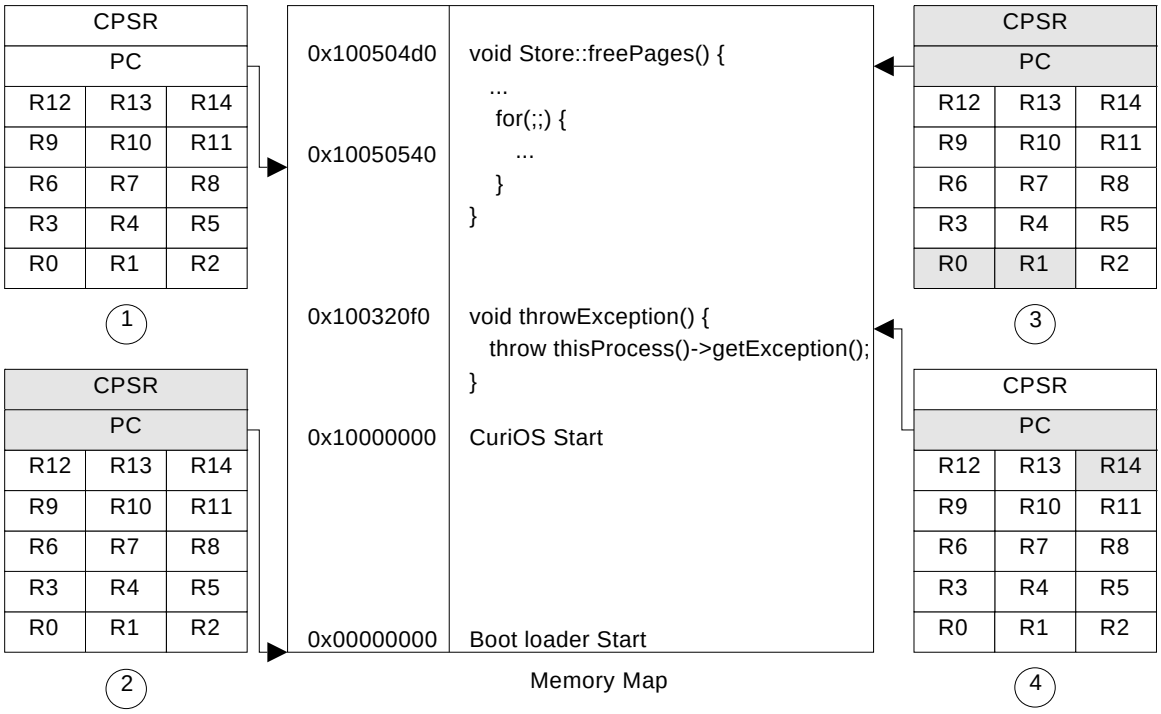

Figure 5.1: Creating an exception from a watchdog bite. Shaded registers identify changes from the previous stage.
Stage 1-Processor registers just before watchdog bites
Stage 2-Processor registers just after watchdog bites
Stage 3-Registers saved by boot loader and approximated PC and CPSR
Stage 4-Modified registers made to look like locked thread made a function call

running programs and data in memory.

In order to ensure that the contents of memory are preserved, the boot loader needs be modified to treat the watchdog bite differently. When the watchdog bites, the boot loader should not reload the kernel and should instead directly transfer control to the operating system start address in the memory. This is possible because once it is up and running, the operating system core is never paged out and resides in the same physical memory area into which it was first loaded.

Once the operating system regains control, a special restore routine takes over. The restore process involves switching the MMU back on, initializing the interrupt controller, re-enabling interrupts and creating a lockup exception in the context of the locked up thread.

An exception can only be properly dispatched by the C++ exception handling libraries if the context in which the exception is thrown is correct. The values of the PC, LR, SP and FP registers should be valid. Thus, simply writing a C++ throw statement in the recovery routine will not work. We needed a way to recover the context of the locked up thread at the time of the watchdog bite. After some experimentation, we discovered that the processor does not lose the contents of most of its registers when it is reset. The program counter is lost because it is reset to 0x0. The value of the CPSR is also lost. But the contents of all the other registers (including SP and FP) are preserved. This is illustrated in stage 2 of figure 5.1, which depicts an example infinite loop lockup in the memory page out service.

We modified the boot loader to respond to a watchdog bite by storing the contents of the reset preserved registers (R2-R14) into a predefined memory region before they are clobbered by running the recovery routine. The contents of R0 and R1 are lost because these registers are used in the store routine for the rest of the registers. But these two values are not required to dispatch an exception. Since the values of LR, SP and FP are recovered, the only remaining register that needs to be valid is PC.

We choose to approximate the value of the PC as the first instruction of the function in which the lockup occurred. In machine code generated by the GNU C++ compiler, the PC is saved on the stack frame in the preamble of every function. We can read the last saved PC from the stack using the recovered stack frame pointer register (FP) and use this value.

A valid CPSR is also required in order to continue executing the thread. The CPSR stores the current processor mode, condition and interrupt flags. Since the lockup happens in kernel mode, the CPSR value can be set to kernel mode with interrupts enabled (the other flags are not relevant for recovery and can be ignored). Once the recovery routine is ready to dispatch the exception, it populates a context object with these register values. This results in stage 3 in the figure.

The context of the locked up thread is now usable for dispatching an exception. A copy of this context is saved for debugging the error. This context object is now modified to

make it seem like the locked thread called a kernel function which uses C++ throw syntax to raise an exception. In order to do this, the LR is set to the value of PC and PC is set to the start address of the function. This emulates the effect of the Branch Link (BL) instruction on ARM that is used for function calls. This is illustrated by stage 4 in the figure. In order to complete the recovery process, this modified context is restored on the processor. The lockup condition is eliminated and a C++ exception is raised in the locked up thread.

The exception that is thrown is an instance of `ARMWatchdogTimeoutException`. This class inherits from the same base class as other exception classes in CuriOS. Standard C++ try-catch syntax can be used to watch for and handle these exceptions.

5.3.2 Limitations

There are a couple of issues that arise when attempting to recover from watchdog bites that reset the processor. Recovery from processor resets requires that the on-chip processor cache is configured as write-through instead of write-back in order to avoid loss of data in the cache after a reset. Thus, when using this technique, we gain increased reliability for some decreased performance. Also, part of the processor context at the time the watchdog bites is lost forever (PC, for example is instantaneously overwritten by 0x0). This makes it difficult to accurately pinpoint the location of the lockup and debug the error. Both these issues are non-existent if the watchdog timer is wired to a non-maskable interrupt on the processor. This would enable the operating system to respond to the lockup without any loss of information in the processor or the cache. All register values will be correct (and can be logged) and debugging the error is easier. The write-through cache requirement is eliminated and recovery is possible without paying a performance penalty for normal operation. Using an NMI also simplifies the recovery implementation because it does not disturb the MMU and interrupt controllers.

5.3.3 Lockup Exception Handling or Thread Termination

With support for lockup exceptions, it is extremely easy to implement simple recovery techniques such as method call retries using exception handlers. An alternate recovery strategy is to terminate the thread that encountered the error. The intuition behind this approach is that one bad thread may be sacrificed in order to allow the rest of the system to continue operation. The Linux kernel widely uses this approach to deal with many types of kernel errors. This motivated us to explore the effect of this technique on recoverability from kernel lockup errors.

We ran experiments to compare the exception handling and thread termination approaches to recovery. For the exception handling experiments, C++ "catch" statements are used to handle exceptions by simply retrying the request. For the thread termination experiments, CuriOS was programmed to point the locked up thread to a termination function instead of the function that raises the lockup exception. Protected object restarts were not used for this experiment.

A modified version of the QEMU emulator [23] was used to perform hundreds of automated lockup fault injection experiments in CuriOS. We randomly pick instruction addresses into which faults are to be injected. A fault is injected by changing the chosen instruction to a self-loop. Interrupts are also disabled in order to create a hard lockup error. The fault is transient and is not re-encountered if the instruction is executed again. We inject only one lockup in each experiment. In one set of experiments, faults are injected when running a gunzip decompression program. In another set, faults are injected when running a sort program. Our goal is to examine if lockups in random parts of the operating system affect the successful completion of these user tasks. The watchdog detects all the hard lockups errors that are encountered.

Figure 5.2 compares the recovery capabilities of the exception based approach with the thread termination based approach. Handling errors using exceptions results in the user

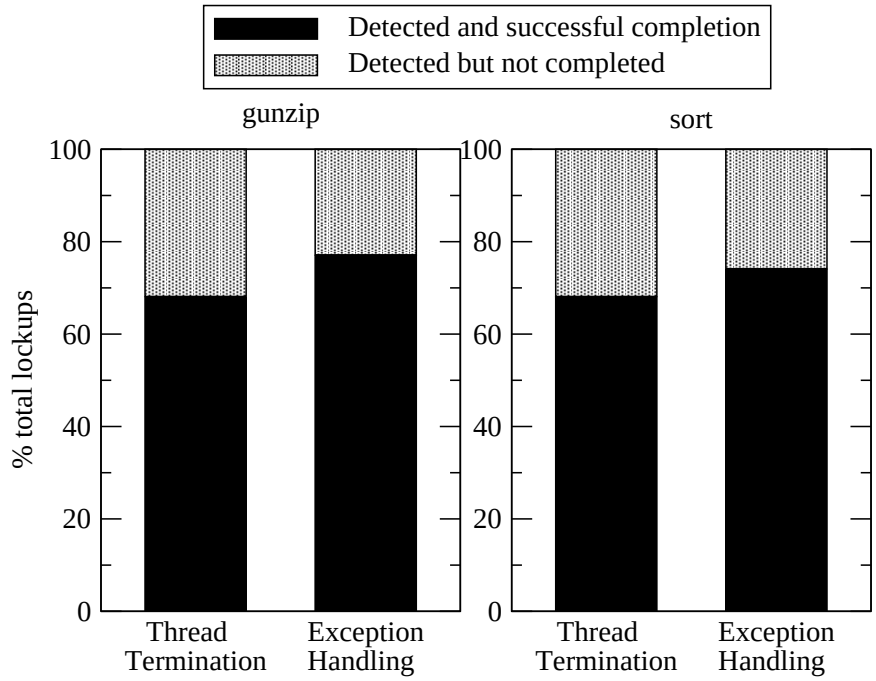

Figure 5.2: CuriOS hard lockup recovery comparison

task completing successfully from about 6-9% more lockups than when using thread termination. This is because exception handlers in various CuriOS objects attempt to recover the locked thread by retrying the method call that failed and some of these retries are successful. Non-recovered lockups are mostly due to data structures left in an inconsistent state. This may be improved significantly by the protected object restart mechanism.

5.3.4 Related Work

Lockup errors can also be detected by hardware enhancements to the processor such as the RSE [62], processor monitoring hardware [63] and logic in the southbridge chipset [64]. The RMK framework [9] detects an operating system lockup by using an APIC to count the number of instructions between two consecutive context switches. They reboot the system if a lockup is detected and do not recover running programs.

There is some directly related research in application recovery after operating system

crashes. The recovery box approach [65] uses non-volatile memory to store application state that is restored when the system is restarted after a crash. Researchers at Rutgers have used remote-DMA in order to access the memory of a crashed system and recover application state [66]. In the Rio file system [67], the buffer cache is recovered from volatile memory after a reset. We recover complete system state after a reset. OKE [56] can detect and recover lockup errors in operating system extensions compiled with a safety enforcing trusted C compiler. Our recovery approach does not require a special compiler and works with existing code.

Chapter 6

Restart-Based Error Recovery

6.1 Component Restarts

Exceptions are raised when errors are detected while executing code within a protected object. Exceptions that are not handled within the object are intercepted at the wrapper which attempts to destroy and re-create the protected object. This is similar to microrebooting or server restarts in Minix3 and can be used to fix transient hardware or software faults. The protected object is re-created in-place in memory ensuring that external references to it remain valid. The method call is immediately retried on the newly constructed protected object. Multiple retry failures cause an exception to be returned to the caller. All normal system activity is suspended until the recovery is completed.

The pseudo code listing in figure 6.1 shows the implementation of the restart mechanism using the wrapper. The changes from the example initially described in Chapter 3 figure 3.3 are underlined. If an exception is intercepted by the wrapper, the destructor is explicitly invoked on the object and this is followed by an in-place reconstruction using the placement form of the C++ new operator. The example shown is a simple case where the protected object does not have any constructor arguments. If required, the wrapper maintains a copy of the constructor arguments in order to re-create the protected object. The `retryCount` variable counts down from the `N` available attempts at invoking the method call. If none of the attempts succeed, the exception is escalated.

Component restarts and multi-threading present an interesting challenge to recovery. For example, the recovery subsystem should handle component restarts when multiple

```
class Example {
public:
   Example() {
     // Constructor
   }
   ~Example() {
     // Destructor
   }
   void runExample() {
     // Code executed in protected object context
   }
};

class ExampleWrapper: public Example {
public:
   void runExample() {
     switchStack();
     switchHeap();
     dropPrivilegeLevel();
     int retryCount = N;
     do {
       try {
          Example::runExample();
          break;
       } catch (Exception e) {
          // In-place destruction and re-creation
          this->Example::~Example();
          new ((void *)this) ExampleWrapper();
       }
     } while(retryCount--);
     elevatePrivilegeLevel();
     restoreHeap();
     restoreStack();
     if (allRetriesFail) throw e; // Escalate exception
   }
};
```

Figure 6.1: Pseudo code for the protected object restart implementation

threads are executing inside the object. A related scenario occurs when a restart is issued during a recursive call into the protected object by a single thread. This problem is not new and has been previously addressed by the Hot-Restart technology in the Chorus operating system [39]. In order to prevent interference from new threads that would like to execute code in the component, Chorus freezes the component. CuriOS achieves the same effect by freezing the entire system for the duration of recovery. Chorus terminates all threads that were executing inside the component at the time of the restart and requires an external "Site Personality Manager" to reactivate the component and create new threads if required. An alternative recovery strategy is to rewind all the threads and resume their execution from their entry points into the component. CuriOS does not completely support multiple thread recovery and this is a rich topic for additional research. In the current version of CuriOS, an exception is raised in each thread, resulting in multiple restarts before complete recovery is possible.

6.2 Server State Management

Operating system servers that need to maintain state information about clients use the state management functionality provided by CuiK to distribute, isolate, and persist client-related state.

A *Server State Region* (SSR) is an object representing a region of memory that is allocated to store an operating system server's client-related information. An SSR is created when a client establishes a connection to the server. For accounting purposes, the memory associated with the SSR is charged to the client. SSRs are protected from both the server and the client through hardware-supported virtual memory protection mechanisms. A client is never granted access to its SSR. A server is only granted write access to a client's SSR when it is processing a request from that client (see figure 6.2). The SSR is passed as an argument to the server's protected method call. The server can then use the SSR to store

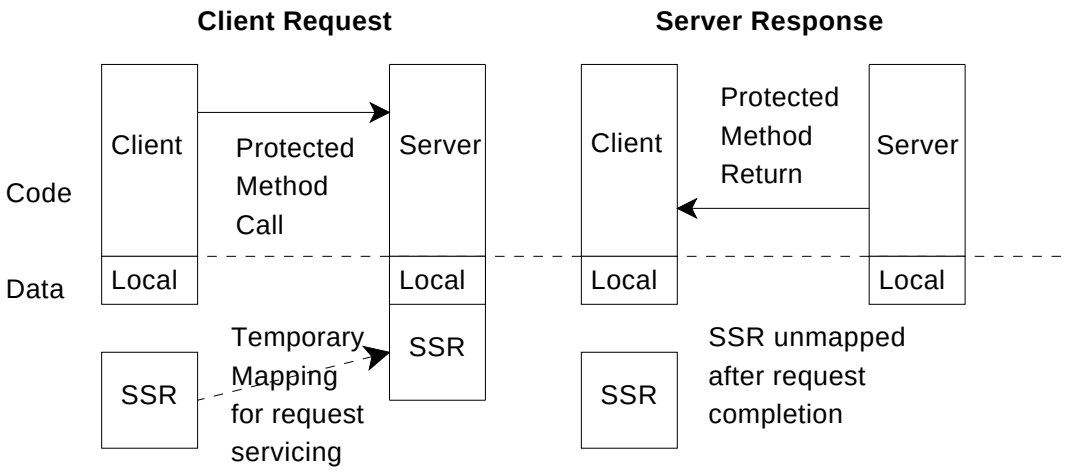

Figure 6.2: Request processing

client-related information. It has full control over management of the memory within the SSR. Write permissions to the SSR are revoked when the protected method call returns. SSRs are implemented using a C++ object that holds a pointer to a hardware-protectable region of memory.

All SSRs in CuriOS are managed by a singleton object called the `SSRManager`. The `SSRManager` provides the functions used to register a new server, bind a client to a server (resulting in the creation of an SSR), undo a client-server binding (deletion of the associated SSR) and enumerate all the SSRs associated with a server. Each server using SSRs is required to provide a recovery routine that is invoked immediately after the server object is re-created upon a failure. This routine can query the `SSRManager` to obtain all associated SSRs in order to re-create the internal state of the restarted server.

Clients do not require any knowledge of the presence of SSRs and the SSR-based interaction with servers is managed completely by CuiK. The functions that implement the mapping and unmapping of an SSR into a server's address space are written as part of the wrapper that implements protected objects. The pseudo code for the this implementation is shown in figure 6.3. The changes from figure 6.1 are underlined.

The `mapSSR` and `unmapSSR` functions manipulate the private protected object page

tables in order to control access to the relevant SSR. While these functions can interact with the `SSRManager` to locate the appropriate SSR, the current implementation simplifies this process by requiring clients to present the handles obtained when binding to the server, for every protected method call. Unfortunately, this limits SSR transparency to clients and also limits drop-in replacements of normal objects with wrappers implementing protected objects. It should be noted that this is just a limitation of the implementation and is not a limitation of the design. We expect that the next major revision of the CuriOS code will address this issue and provide complete transparency.

It should be noted that the changes in figure 6.3 are only used for servers that use SSRs. Servers that don't require SSRs are implemented as plain restartable protected objects.

An SSR-based service provides protection from unbounded internal error propagation when compared to a traditional microkernel service. This protection is due to the fact that errors that occur when a particular SSR is mapped in cannot directly corrupt the state in SSRs that are not mapped in. This reduces the probability of intra-component error propagation. It may not be possible to completely eliminate intra-component error propagation because of the presence of other channels such as function call arguments and local protected object state. We expect that such cases will not occur often. Nevertheless, strong function argument checking should help mitigate error propagation in these cases.

6.3 Operating System Service Construction

Servers that don't maintain any client-related state can be easily restarted and do not require state management functionality described in the previous section. Similar to popular usage, we refer to such servers as *stateless*. Note that this does not imply that the server has no internal state; it only implies that the server maintains no session state. Each request to such a server is treated independently.

On the other hand, many operating system servers maintain client-related state and are

```
class Example {
public:
   Example() {
     // Constructor
   }
   ~Example() {
     // Destructor
   }
   void runExample() {
     // Code executed in protected object context
   }
};

class ExampleWrapper: public Example {
public:
   void runExample() {
     switchStack();
     switchHeap();
     mapSSR();
     dropPrivilegeLevel();
     int retryCount = N;
     do {
       try {
          Example::runExample();
          break;
       } catch (Exception e) {
          // In-place destruction and re-creation
          this->Example::~Example();
          new ((void *)this) ExampleWrapper();
       }
     } while(retryCount--);
     elevatePrivilegeLevel();
     unmapSSR();
     restoreHeap();
     restoreStack();
     if (allRetriesFail) throw e; // Escalate exception
   }
};
```

Figure 6.3: Pseudo code for the SSR mapping and unmapping implementation

stateful. How should the state of a generic stateful server be structured in order to use the state management support provided by CuiK? Stateful servers can be classified into two types. The first type is a server that does not require information about all of its clients in order to service a request. A server that provides pseudo-random numbers based on a per-client seed is one such example. It only needs to know the client's seed in order to service a request from that client. Such servers can store client information in SSRs and can be transparently restarted upon a failure. All future client requests will continue to work correctly because its SSRs and the information stored in them is not lost.

The second type is a server that requires knowledge about all of its clients in order to service a request. Examples of this type are operating system services like the scheduler and timer managers. Such servers can store client related information in SSRs and can redundantly cache this information locally to process requests. Upon a restart, such a server should be able to re-create its internal state from its distributed SSRs. Depending on the design or implementation of the server, it is possible that some internal state is irrecoverable and cannot be completely reconstructed. For most operating system services, we believe that this is a reasonable trade-off when compared with the complete state loss alternative. Several stateful CuriOS components are structured as one of these two types of servers. The key observation here is that the use of SSRs makes stateful servers look like stateless servers from a restartability viewpoint.

There are a few other design issues that need to be considered when writing operating system services using SSRs. It is possible that the SSR that was in use at the time an error occurs is corrupted. The recovery routine in the server can check the consistency of the objects in SSRs using simple heuristics before using them. CuriOS uses magic numbers in objects and these are checked for corruption. CuriOS also uses server-specific checks to ensure that pointers and numbers are within expected ranges. Multi-threaded servers may use locks to control access to data structures. It is possible for a failure to occur during the update of a lock-protected structure. The recovery routine should be able to re-create

the internal data structure to ensure consistency as well as ensure the correct state of the internal locks.

Unlike EROS which does consistency checks of all state during normal running time, these SSR consistency checks in CuriOS are only performed on exceptional conditions that require server recovery.

CuriOS has some limitations that are common with several other microkernel systems that execute operating system code in unprivileged mode. It is expensive to switch processor modes to perform privileged operations. A reasonable balance between performance and reliability should be considered when designing operating system services as protected objects. In some cases it is possible to split functionality so that some privileged operations can be performed by an helper object that resides inside CuiK.

6.4 Intra-Component Error Propagation

The microkernel-like partitioning of operating system components in CuriOS limits inter-component error propagation. The further partitioning of individual service state into SSRs provides protection from intra-component error propagation as well. In this section, we will illustrate this property by examining in detail all possible avenues for the propagation of an error that occurs within a CuriOS service.

When an error occurs in a server during the processing of a client request, the memory protection enforced by the protected object mechanism limits damage to the writeable regions of the server's address space. This includes server local state (if any) and all the SSRs that are mapped in at that time. The SSRs that are not mapped in are not immediately affected.

Can an error eventually propagate to unmapped SSRs? There are two possible propagation paths that may allow this to happen. These are illustrated in figure 6.4. The image on the left represents an operating system service that has just encountered an error. The

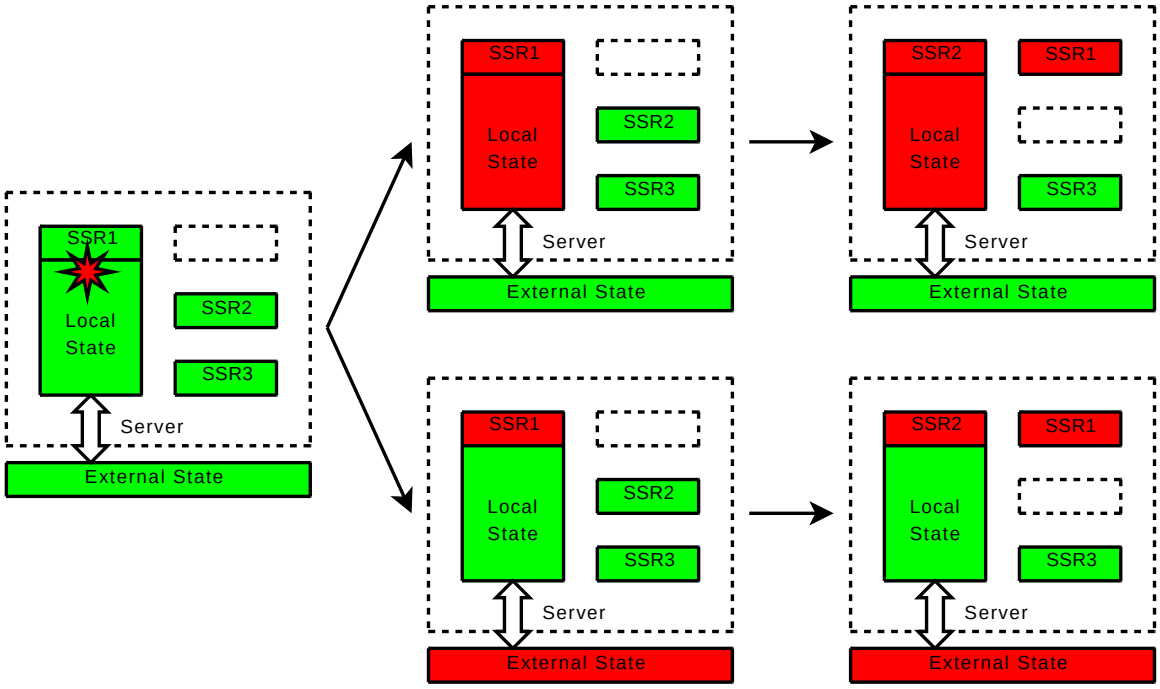

Figure 6.4: Error propagation between SSRs

darker boxes represent state that has been corrupted by the error.

One path involves undetected corruption of server local state, which in turn results in the error propagating to an SSR that gets mapped in at a later point in time. In cases where the server has no writeable internal state and all required state is maintained in the SSRs, this avenue for error propagation does not exist. The other path involves corruption of some external state as an intermediate step in propagating the error to other SSRs. The external state can be present in either software or hardware. External software state is affected by function call arguments or return values. Hardware state can be affected through either directly or indirectly allowed access.

Because CuriOS servers are multi-threaded, it is possible that multiple SSRs are mapped in when an error occurs. In this case, the error propagation is limited to the SSRs that are currently mapped in. This can be further improved by including support for thread-level memory protection.

6.5 Recoverable Errors

Protected method calls to servers are designed to retry the request after a server fails and restarts during the processing of the request. SSR-based recovery addresses a large class of errors that result in corrupted local operating system service state. Complete reconstruction of service local state during recovery can remedy any such corruption. When an SSR is corrupted and multiple attempts at recovery fail, only the client associated with the corrupted SSR is affected and will need to be notified of a failure. The other clients of the service can continue to function normally. Thus, SSR-based recovery can minimize the impact of a software bug that is triggered by a specific client request and a consequent failure. When repeated attempts at processing the request fail, the client can be notified and the service can continue processing other requests that do not trigger the bug. This illustrates that SSR-based restart-recovery mechanisms are effective against both heisenbugs and bohrbugs [68].

Restarting a service that has visible external effects may not always result in correct behavior. For example, restarting a printer driver due to a failure may cause another copy of the print job to be dispatched. This problem may be ameliorated to some degree by writing code that is restart-aware. This is achieved by incorporating some means of recording the progress made in servicing a request. This limitation has also been acknowledged for device driver restarts in Minix3 [35]. Similar to the approach taken by Minix3, we advocate notification of possible non-transparent recovery to applications or users.

It is important to note that our design does not attempt to provide recovery for generic distributed processes that have arbitrary levels of complexity and interaction. We are only attempting to show that many typical operating system services can be constructed using such a scheme in order to build a highly reliable operating system layer.

Chapter 7

Restartable Components

In this chapter, we discuss the details of implementing several CuriOS components as restartable protected objects. As described earlier in Chapter 6, these components are either stateless or stateful. Stateless components can be easily restarted whereas stateful components are designed to use SSRs for transparent restarts.

7.1 Stateless Components

7.1.1 Hardware Device Drivers

Several hardware device drivers are encapsulated in protected objects to contain error propagation. The serial port driver in CuriOS is implemented as a protected object with complete access to the memory mapped registers of the serial port controller. This protected object is stateless and only has one client: the CuriOS console object. Errors that occur when reading or writing to the serial port are handled by restarting this protected object and retrying the request. CuriOS has a NOR flash driver that is implemented as a stateless protected object. This protected object is created with read/write access rights to physical memory regions that map to NOR flash chips. An error that occurs in this protected object can lead to potential corruption of arbitrary data stored in NOR flash, but cannot easily corrupt read-only mapped system memory. Protected objects are also used to encapsulate drivers for interacting with the hardware timers. These are used to start, query and stop the hardware timers. Interrupts from the hardware timers are also dispatched to them. These

are currently stateless and can be restarted.

7.1.2 File System Drivers

CuriOS inherits the file system design from Choices. Low-level file system drivers implement individual file systems. Additional file system functionality such as buffer management and application interaction is managed by other layers. We modified two stateless low-level file system drivers in CuriOS to use protected objects.

`CramFSFileObject` is a class that provides access to a compressed file on the read-only CramFS file system [69]. When a file is opened, an instance of this class is created as a protected object. This instance only has information about its backing storage and does not maintain any state regarding clients. Hence it does not require usage of the server state management functionality. The method call to read a file provides both the offset into the file and the required number of bytes. The protected object is only granted privileges to modify its own data and the destination buffer. Calls to other objects like the backing storage are mediated by CuiK. Using a protected object for each file has several reliability benefits. An error that occurs when processing one file is contained within the protected object and cannot corrupt arbitrary memory in the system. If the error were transient, a restarted protected object can continue serving clients. If there is an error in a compressed file stored on the disk that causes the decompression routines to fail, it only causes an error in the clients that were reading that particular file.

CuriOS also includes support for the Linux ext2 file system. An `Ext2Container` protected object is created for every ext2 file system on disk. This manages the inode and free space bitmaps. If this protected object crashes and restarts, it can re-read this information from disk. An `Ext2Inode` protected object is tasked with managing all interaction with a file. This protected object only has privileges to modify the inode it represents, which, in turn, has all the pointers to disk blocks comprising the file. This has similar reliability benefits as the `CramFSFileObject` protected object. Since protected objects

are re-created in-place, the same objects can be used to access the file after the service is restarted.

It is important to realize that when we refer to file system service recovery, we are not dealing with recovering corrupted file system state on disk. There are many other programs that are designed to handle corrupted disk blocks and this is an orthogonal problem with recovering the state information for a live file system server. It is possible that an error that occurs in the file system service results in corruption of state on the disk before the service is restarted and recovered. The combination of file system service recovery provided by CuriOS and disk state recovery using other tools results in an extremely reliable file store.

7.2 Stateful Components

7.2.1 Timer Management

A `PeriodicTimerManager` service provided by CuriOS allows clients to access timer functionality. There is only one instance of this class in the system and it is created as a protected object. The manager itself depends on device driver objects for the hardware timers. Clients interact with this service through a `PeriodicTimer` helper object and can request to be notified periodically. The `PeriodicTimer` registers with the `PeriodicTimerManager` on behalf of the client when the timer is started. When the client calls the `await()` method, it starts to wait on a semaphore. The job of the `PeriodicTimerManager` is to signal the semaphore when the client's time is up. In order to support recovery from a restart of the `PeriodicTimerManager` service, SSRs are used to persist and distribute information regarding each `PeriodicTimer` client. This includes the semaphore, starting time and the timer period. Within the service, timer functionality is implemented using a linked list that holds pending notification events. Upon a restart, the `PeriodicTimerManager` can re-create its internal linked list from

the timer period and starting time information in the distributed SSRs. This allows it to continue sending notifications to registered clients.

7.2.2 Scheduling

CuriOS schedulers are modeled as a container into which processes can be added or removed. They implement a particular scheduling strategy by determining the order in which processes can be removed. All schedulers inherit from the `ProcessContainer` base class. The interface exported by all schedulers includes the following methods: `add()`, `remove()` and `isEmpty()`. The scheduling logic is implemented by specific subclasses. A FIFO scheduler, for example, is implemented using a list of processes to run and only allows processes to be removed in the order that they were originally inserted.

The system scheduler is created as a protected object with clients as individual processes. SSRs are used by the scheduler to store scheduling information about each process. If the scheduler is restarted after a failure, it queries the SSRManager for all its clients and re-creates its internal list. The ordering of processes in the scheduler queue can be loosely maintained during the reconstruction phase by querying process attributes. For example, a priority-based scheduler can sort the reconstructed queue based on process priority. The original ordering in the scheduler queue may be lost upon restart if multiple processes have the same priority.

7.2.3 Networking

The recovery mechanisms in CuriOS allow for the construction of an extremely reliable network protocol stack. CuriOS uses the lwIP networking stack [70] encapsulated in two restartable protected objects: one for managing TCP connections and the other for UDP.

In order to enable lwIP to work with CuriOS, we had to first refactor the lwIP C code to C++ code by changing C functions to C++ object methods. This was necessary in

order to adhere to the protected object model in CuriOS. The functional changes to lwIP primarily focused on its internal memory management framework. Our changes ensure that allocations of the data structures that manage TCP and UDP state are performed from SSRs instead of the default protected object memory allocator. More specifically, lwIP creates a *tcp_pcb* or a *udp_pcb* data structure to manage state information for every connection. We refactored lwIP code to place these data structures within SSRs. In the case of TCP, for example, this includes all information necessary to service the incoming and outgoing packets of a connection. This includes the network addresses, ports, windows, sequence numbers and so on. If the TCP service crashes and is restarted, this information is used to resume the processing of packets. If this state information is not preserved during a restart, the unfortunate consequence is that all network connections in progress will be terminated.

Each SSR is associated with a client `Socket` object and is mapped into the TCP object's address space when interacting with it. When there is an incoming packet, the corresponding SSR is located and mapped in before sending it through the stack. A similar approach is used to provide access to SSRs for the TCP object's timer driven events.

Integration of the lwIP stack also involved converting all the assert code in lwIP to throw exceptions instead of halting the stack. This comprehensive error detection in lwIP helps reduce error propagation and improves recovery rates.

lwIP only provides higher networking layer functionality and does not include device drivers for the hardware at the lowest level. We were able to modify Linux drivers for the Ethernet chip on our target platforms to work with CuriOS in order to provide this functionality. lwIP only works for Ethernet networks. CuriOS inherits MAC layer demultiplexing code for different networks from Choices and only dispatches Ethernet messages into the lwIP stack. The low-level Ethernet driver is stateless and is not restarted when the rest of the lwIP stack is restarted.

How is the restartable network stack in CuriOS different from other network connection recovery techniques? In order to cope with network disconnection events in traditional

operating systems, a few high-level network protocols such as HTTP and FTP support resuming an interrupted data transfer after establishing a new TCP connection. This requires application support and is not widely adopted. The transparent recovery of the network stack provided by CuriOS ensures uninterrupted operation for applications using non-resumable connection-oriented protocols.

Chapter 8

Evaluation

In this chapter we evaluate the CuriOS implementation in terms of error recovery capabilities as well as performance and memory overheads. A brief analysis of the refactoring effort involved in converting operating system services to use the state management framework in CuriOS is also presented.

8.1 Error Recovery

In order to analyze the error recovery capabilities of CuriOS services, we resort to fault injection experiments using a modified version of QEMU. It should be noted that error recovery works equally well on real hardware and we only use the emulator in order to provide non-intrusive and large-scale automated fault injection.

Our fault injection tool picks a random instruction in the server code and injects a fault just before that instruction is executed. In each experiment run, we inject exactly one fault. We inject two types of faults. The first type of fault is a memory access fault (or data abort). These virtual memory faults are instantaneously detected at the injected instruction and immediately cause an error. The fault latency is zero in this case and error propagation is limited. The other type of fault that we study is a register bit-flip. The tool randomly flips a bit in one of the register operands for the selected instruction. Register bit-flips do not always lead to errors. This is because a corrupted register can be overwritten by the result of the instruction. They can, however, lead to latent errors which may not be detected immediately. This type of fault has been previously used by other researchers to emulate

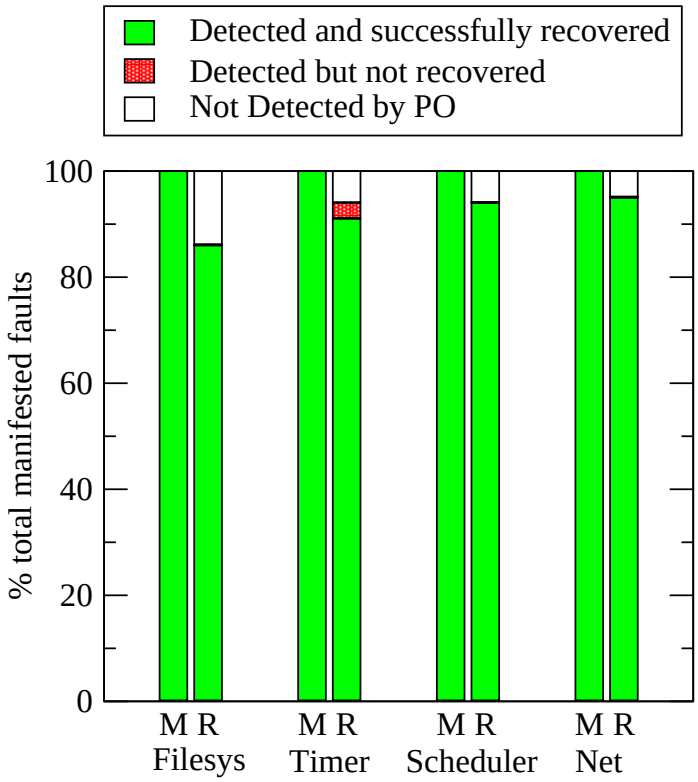

Figure 8.1: Error recovery after fault injection: The “M” columns present results for the memory access fault experiments and the “R” columns present results for the register bit-flip experiments.

several common programming errors such as incorrect assignment statements and pointer corruptions [71].

We study error recovery for the file system, timer manager, system scheduler and the networking stack. We perform hundreds of fault injection experiments per service and the results are shown in figure 8.1. The Y axis represents the percentage of faults that had some visible manifestation (errors). Note that all of these manifested faults would result in a service or system failure in most existing operating systems. On the other hand, restart recovery with state management used by CuriOS allows it to recover from significant numbers of such errors.

An error is reported as successfully recovered if the operating system is usable after service recovery. This is confirmed by testing its ability to schedule new processes and

access the disk. In some cases, a client's connection to a server is terminated because of a corrupted SSR or repeated errors while the system as well as other clients using the restarted service still remain usable. We count such cases as successful system recovery in the figure shown. In the face of arbitrary errors that corrupt a client's SSR or request, there is no possibility of maintaining the unfortunate client's connection.

8.1.1 File System

We run several programs that access files on disk and inject faults into the ext2 file system code that handles requests from these programs. This code is present in the `Ext2Inode` protected object.

The memory abort faults are always completely recovered after a retry. 13% of the manifested register bit-flip faults are not detected by the protected object mechanism and are therefore not recovered. These cause the system to fail. This is because register bit-flip errors can propagate to other CuriOS subsystems through invalid method call arguments. We do not yet perform an exhaustive check of the validity of all method call arguments in CuriOS. The addition of such checks would help improve error confinement and recovery rates.

8.1.2 Timer

This experiment is set up so that a couple of processes that use the timer are started before faults are injected into the service. These are simple applications like a clock that displays the time of day.

The memory abort errors are fully recovered by reconstructing the internal timer list. In the register bit-flip injections we see some cases (3%) where errors are detected but are not recovered. 6% of the register-bit flips evade detection by the protected object mechanism and cause unrecoverable errors in other parts of CuriOS.

8.1.3 Scheduler

This experiment is set up similar to the timer manager experiment with several running processes in the system. Faults are injected into the system scheduler. The goal of this experiment is to examine if a failure in the scheduler can be recovered. This would allow CuiK to continue scheduling processes.

For the memory abort experiments, re-creation of the internal linked list is always successful. In the case of the register bit-flip experiments, 6% of the errors are not detected by the protected object mechanism and cause CuriOS to crash. We noted that in a few experiments, one process was sacrificed to allow the rest of the system to continue functioning.

8.1.4 Network

We run a simple web server and an echo server in CuriOS while also running an HTTP client that fetches a half-megabyte file from an external host. Faults are injected into the lwIP code for TCP processing of IP packets on both the send and receive paths. If an error is detected, the TCP stack protected object is restarted and the request is retried. If multiple attempts at executing the request fail, an exception is thrown. If this exception is thrown when processing an incoming IP packet, the packet is silently dropped by the IP layer. This has no effect on the correctness of TCP because this is similar to packet loss on the network and is recovered by the TCP stack. If an exception is thrown back to a client with a TCP connection handle, the TCP connection for that client is terminated. We verify that the network stack is still usable and other connections are unaffected in spite of a single connection failure.

For the network stack, 5% of the manifested register-bit flip faults are not detected and consequently, not recovered. While the system is recovered in 95% of the cases, 45% of these recoveries were at the cost of a single client's TCP connection termination due to state corruption.

Table 8.1: Protected method call performance

Protected call	Instruction overhead	Time overhead (microseconds)
Without SSR	1593.77 ± 3.57	195.65 ± 0.48
With SSR	4893.33 ± 2.83	378.93 ± 0.93

8.2 Performance

A protected method call incurs additional processing overhead in comparison to a normal C++ method call. We made use of both of CuriOS' supported platforms to measure the overhead associated with protected method calls. On the OMAP1610 H2 hardware platform we measured the overhead in terms of microseconds of execution, and on the QEMU emulator we measured the overhead in terms of instructions executed. The OMAP1610 processor was clocked at 96 MHz, and the same test source code was used for both platforms. Table 8.1 shows the overheads for the two types of protected method calls: a protected call into a stateless server that does not require the mapping of an SSR and a protected call info a server that uses an SSR to manage state information. In the second case, additional processing is required to map the SSR into memory. The time overhead for switching into and out of a protected object domain is comparable to the cost of performing two context switches (148 microseconds for two switches) in CuriOS. Since a protected method call is analogous to switching between two microkernel domains, we believe that this represents acceptable performance. The numbers reported here are the average of 100 trials with error estimates provided by the sample standard deviation. We believe that these overheads may be further reduced with careful code optimization.

Apart from the extra code implementing the protected object mechanism, a major source of overhead is the need to flush the TLB when switching between page tables. While the ARM architecture allows for selective flushing of TLB entries, our current implementation does not support this feature. The single address space design of CuriOS helps to keep the costs of protected method calls down by obviating the need to flush the

virtually tagged caches on the OMAP1610 ARM processor.

How fast does recovery happen? When an error is detected, the exception handling framework signals the error and the C++ library unwinds the stack and destroys stack objects. Restarting the server requires re-running the constructor for the protected object and code to recover information from SSRs (if required). Altogether, the time from error detection to a recovered system is usually on the order of a few hundred microseconds.

8.3 Memory Overheads

Protected objects, like user applications, require additional page tables to enforce memory protection and this results in some memory overhead. Each protected object also has an associated heap and a stack for each thread that can execute within the protected domain. The memory overhead due to stacks depends on the number of threads that use the protected object. The use of SSRs also results in some memory overheads. We use hardware protection to isolate SSRs. However, hardware protection is not always available for small memory regions. Thus the minimum size of an SSR is determined by the smallest hardware-protectable region of memory. For example, on the ARM platform, this is a 1 KB page. Our current implementation uses a page for the minimum size of an SSR. This results in some memory waste. If this is a concern for small embedded devices, our design can be extended so that multiple SSRs share the same protected area. This saves space at the cost of better isolation between the SSRs. This problem may be mitigated by future architectural support for finer granularity of access control such as Mondriaan Memory Protection [72]. Nevertheless, the total memory overhead per protected object in the ARM implementation of CuriOS is only on the order of tens of kilobytes when there are a small number of clients. This includes 20 KB for a minimal set of page tables plus memory pages for the heap, per-thread stack and per-client SSR (at least one page for each).

8.4 Refactoring Effort

Our proposed operating system design requires writing operating system service code to encapsulate objects in protected domains and to utilize our state management framework. The protected object support in CuriOS is implemented through wrapper objects. Wrappers are currently written by hand and consist of a one line statement per object method. The statement is a C++ preprocessor macro that expands to the code required to switch into and out of the associated protection domain. This additional complexity may also be avoided by using an automated wrapper generation tool. Code-changes are also required to refactor operating system services so that they can make use of the state management framework. Care should be taken to not introduce new software faults (bugs) in the process of separating state from services.

Our experience with re-engineering operating system service code to separate state indicates that it requires about 12-24 person-hours to design and refactor an operating system service to work with our framework. This includes the time spent in fixing most bugs uncovered using fault injection. The use of SSR-based state management in the file system, scheduler and the timer manager required less than 50 additional lines of code in each component. In order to convert the lwIP networking stack to use SSRs, we had to change around 100 lines of code. This mostly involved replacing calls to its internal allocator with the SSR-based state management code.

Chapter 9

Fault-Tolerance Patterns

The architecture and design of CuriOS illustrate many software patterns that are used to detect, signal and recover from errors. An excellent catalog of patterns for fault-tolerant software is available in a recent book by Robert Hanmer [73]. In this chapter, we visit a number of patterns described in the book and describe their usage within CuriOS.

9.1 Architectural Patterns

Escalation: *When recovery or mitigation is failing and there is no hope that it will suddenly succeed, escalate the action to the next more drastic action.*

CuriOS resorts to escalation when repeated attempts at restarting a server and retrying a client request fail. The exception intercepted at the protected object wrapper is re-raised and error handling is delegated to the client. The client is then responsible for handling the exception and attempting alternate recovery strategies.

Minimize Human Intervention: *Humans make mistakes and are slow; to minimize down-time the system should take care of itself, without human intervention.*

Similar to other high-reliability and high-availability systems like the IBM z/OS, CuriOS attempts to self-heal and does not require human involvement unless all programmed recovery actions fail. This is important for both detection and recovery. While automated support for these actions reduces possible human operator mistakes, such support is also very useful from a speed perspective. Faster detection limits error propagation and fast re-

covery ensures that applications do not encounter timeout conditions and other cascading failures.

Units of Mitigation: *Choosing to consider the entire system as a monolith limits the kinds of recoveries that are possible without the system being unavailable. The entire system will stop and when that happens, a complete restart is required. Choosing clear smaller units of mitigation can help recovery.*

CuriOS adopts a microkernel-type approach to operating system construction and is built as a composition of small structural units encapsulated in protected objects. The strong hardware-supported memory access controls limit error propagation, while also serving as a mechanism for error detection. Each individual component is independently restartable and represents a unit for error recovery. The intuition behind this design is also the foundation for the Microreboot [13] paradigm. Mitigating errors at smaller components allows the larger system to continue functioning without encountering failures.

9.2 Error Detection Patterns

Error Containment Barrier: *Put a barrier into your system so that errors do not spread.*

Unbounded error propagation cannot be allowed in any design for a reliable system. Without error containment, it may be extremely difficult to incorporate recovery actions. The protected object mechanism in CuriOS serves as an error containment barrier. Code executing inside a protected object has restricted privileges and cannot corrupt arbitrary regions of memory. The barrier is enforced through appropriate configuration of the processor's MMU.

Riding Over Transients: *Sometimes the prudent thing to do is to ignore an error if it is something that might be due to a transient situation.*

A number of errors that occur in systems can be attributed to transient faults such as race conditions and flipped-bits in hardware. Such errors usually do not resurface when the code is re-executed. Thus, when an error condition is detected inside a CuriOS server during processing of a request, the request is not immediately denied. It is retried several times before the recovery mechanism gives up and escalates the exception.

Watchdog: *Institute one task watching over another to make sure that it is still behaving well.*

It may not always be possible for a task to ensure correctness of its own operation. An external observer is best suited for this job. The watchdog timer support in CuriOS is a direct application of this pattern. In this case, the monitoring entity is an external hardware timer.

9.3 Error Recovery Patterns

Concentrated Restoration: *Concentrate on the restoration task.*

All recovery actions in CuriOS are performed after placing the system in a quiescent state. This ensures uninterrupted rapid completion of the restoration actions and avoids potentially harmful interference.

Data Reset: *Restore some data to its initial (or a predetermined) value when it is found incorrect.*

The protected object restart mechanism is designed around this pattern and attempts to mitigate the effects of any data corruption by re-initializing the state of the object. The constructor of the object is invoked for this purpose. While more complex automatic data structure recovery approaches have been proposed [74], this simple approach works extremely well in many cases.

Exceptions: *Handle an error in a controlled manner.*

CuriOS directly applies this pattern by using the C++ exception handling support. Defined error handling blocks are provided as exception handlers and errors are handled in a synchronous manner.

Limit Retries: *Do not return to the scene of an error without changing something, unless you want the error to reoccur.*

This well known principle is fully utilized in CuriOS. Simply retrying a failed request may not always work and is the reason why CuriOS performs an additional restart of the protected object before attempting a request retry.

Restart: *Resume execution by restarting the program at hand from the beginning.*

This pattern is again visible in CuriOS as the protected object restart mechanism. It ensures that retries are limited and that data is reset. The entire restart-based recovery is carried out as a concentrated restoration.

Chapter 10

Additional Dependability Benefits

While the primary focus of this thesis is operating system reliability, this chapter highlights other dependability benefits of the CuriOS architecture and implementation.

10.1 Security

Security is an important factor contributing to the overall dependability of operating systems. Some of the reliability enhancing techniques in CuriOS automatically translate into security improvements. In this section, we describe the improvements to system integrity, confidentiality and availability.

10.1.1 Integrity and Confidentiality

In addition to helping confine errors, the restricted memory address spaces provided by protected objects are a good defense against misbehaving code from a security perspective as well.

Our design follows the Need-to-Know security principle. CuriOS only maps in memory that is necessary for a protected object to execute. The heap and stack for the object as well as the SSR region for the request are mapped in with read-write privileges. Writes are disallowed to all other operating system data regions and code, thus protecting their integrity. This design is most closely related to Nooks, which uses similar protection policies for kernel memory. We differ from Nooks in that protected objects are further constrained because they execute in an unprivileged processor mode.

Our design ensures that a server only has access to SSRs for clients currently being serviced. This prevents a malfunctioning or compromised server from affecting the integrity of information used by inactive clients. By default, CuriOS presents code executing in a protected object with read access to the complete operating system. Confidentiality requirements may be addressed by enforcing additional memory access constraints to disallow reads of sensitive information.

Although we restrict the scope of possible damage, we have not yet evaluated the impact of intentionally malicious components in CuriOS. A promising direction for future work includes building a threat model and identifying a comprehensive set of restrictions that need to be imposed on protected objects. This may involve fortifying the protected method call and server state management mechanisms by borrowing ideas from systems like EROS.

10.1.2 Availability

Because the SSR is allocated out of the client process assigned memory pool, the client process cannot perform a denial of service attack by issuing requests that result in the allocation of large amounts of operating system state. This helps ensure the availability of operating system services. Existing designs such as the reincarnation server in Minix3 ensure high availability by simply restarting managed servers when they fail. We take advantage of this technique to improve server availability in our system as well.

While our initial goal was to ensure that an error in the server did not unnecessarily affect clients, our design also helps with creating robust client applications. A client can request that its SSRs be persisted across client failures and restarts. Since the error that caused the failure cannot corrupt the SSR, it can be used to resume an interrupted session if the client can re-create its internal state using other techniques such as persistent memory. For example, if the SSR holds TCP connection information from the network server, the TCP connection can be resumed without interruption after a crash and restart of the client. This can improve both the availability and reliability of the client.

10.2 Maintainability

The design of CuriOS makes it possible to easily upgrade a running service. A transparent upgrade is achieved by simply terminating the old server and starting a newer version while preserving the SSRs. If the interface of the new server is backwards compatible and it can interpret the existing SSRs, it can continue serving existing clients. Some related work in this area include Pannus [75] and Ksplice [76] for Linux and dynamic updates in the K42 operating system [77].

Chapter 11

Related Work

11.1 Fault-Tolerance

A number of standard fault tolerance techniques are available in literature. These include redundancy in hardware and software, transactions, error correction codes for memory, majority or Byzantine voting, and other software fault tolerance approaches [78]. Some of these techniques can be directly applied to CuriOS to further improve its fault tolerance. These techniques may be used to ensure that the core of the system (CuiK and recovery code) itself is protected from failure.

In the past, designs for reliable computer systems have used redundancy in hardware and operating system software to detect and attempt recovery from transient and permanent hardware faults [79, 80]. Redundancy can be used to detect and mask some types of software faults [81]. But it is ineffective against errors due to software bugs which affect all replicas. Neither does it immediately address the insidious problem of the propagation of undetected errors [4]. Additionally, these systems are extremely expensive to build and use [82].

The QuickSilver operating system [83] uses transactions to recover to a consistent system state after a failure. But server failures are not recovered and clients receive error codes when they try to communicate with a failed server. Quicksilver also provides a log manager interface that can be used by servers to store data required for recovery. But, there is no mechanism that allows for isolation of per-client state. Also, logging encounters substantially more overheads than our lightweight memory isolation approach. VINO [84]

also used transactions to roll-back changes made by misbehaving kernel extensions. Arjuna [85] is a middleware system that supports transaction-like semantics on objects to provide failure atomicity.

We have also investigated the use of software transactional memory techniques to protect component state in CuriOS [27]. The use of transactional semantics alone to recover complete component state is only effective when errors are detected before commits. When this property cannot be enforced, there are no constraints on error propagation within the component. However, when used together with our SSR-based approach that reduces error propagation, transactions can provide an additional layer of protection to SSRs while they are being manipulated by a service.

In addition to some of the operating systems discussed in Chapter 2, many other system designs incorporate virtual memory protection to improve reliability. In the Rio project [67], virtual memory access control was used to protect the file cache from corruption by errors occurring elsewhere in the system. The protected object concept is similar to a virtual memory protected region in Nooks [5]. However, unlike Nooks, a protected object executes in an unprivileged processor mode. More importantly, while Nooks is designed to wrap operating system extensions such as device drivers, a protected object can encapsulate core operating system components.

Unlike the shadow driver mechanism [86] used by Nooks, the SSR-based recovery mechanisms can isolate requests that cause crashes because of a software bug and continue servicing requests that do not trigger the bug. This is possible because of the rigorous partitioning of per-client state in CuriOS. While the shadow driver approach may work for heisenbugs, a bohrbug will be triggered in the shadow driver just as it was in the original driver since the same code is used.

Operating system service design using SSRs is closely related to the principle of crash-only software [87]. Similar to crash-only components, recovery involves a component restart and component crashes are masked from end users using transparent component-

level retries.

There is a large body of work related to isolating operating system extensions so that they don't affect the rest of the system. Chapter 2 described several microkernel systems that achieve this goal. SPIN [88] exploits the safety properties of Modula-3 to allow users to download code into the kernel. The OKE [56] system allows users to load arbitrary code into the kernel. Kernel safety is ensured using a combination of trust management, a trusted C compiler and language customizations. A couple of other software-based solutions are XFI [21] and SafeDrive [20].

11.2 Hardware Protected Objects

The protected object paradigm in CuriOS has been previously adopted in several other systems as well. In fact, as far back as the early 80s, Intel's iAPX 432 microcomputer included extensive hardware support for object-based computing. iMAX [89, 90] was a multiprocessor operating system written in Ada and built by Intel for the iAPX 432. It was completely designed around the idea of data abstraction and provided a uniform Ada view of the underlying hardware and the operating system extensions. Protection was enforced within operating system modules in order to limit the damage caused by errors.

In the ProtectOS system [91], under-utilized hardware security features of the Intel 80286 processor were exploited in order to provide isolation between objects within the same process. More specifically, separate segments were used to store the code and data associated with each object and achieve isolation. While the hardware mechanisms used to implement protected objects are different from CuriOS, ProtectOS has similar motivation and goals. Also, ProtectOS was designed for objects written in C and had special compilation requirements. CuriOS implements protection using wrappers built around standard C++ objects.

11.3 Protection Domains

The Mungi operating system [46] used a technique called protected domain extensions [92] to dynamically extend a caller's protection domain for the duration of a procedure call. While there are semantic differences with the SSR framework in CuriOS, this is similar to mapping an SSR into a server's protection domain in order to expand its privileges.

Chapter 12

Future Work

12.1 Improved Error Detection

Early and comprehensive detection of errors is the first step in achieving high reliability. While CuriOS supports several error detection methods, there are many other improvements that can be made in this direction. Thorough checking of function call arguments can reduce error propagation outside of protected object boundaries. Stronger type checking through the use of other programming languages or code-rewriting tools such as those used in SafeDrive [20] can provide an additional defensive layer.

A direct extension of this work is to take advantage of additional hardware support available in newer ARM processors. Features such as the eXecute-Never (XN) bit and TrustZone can be used to design better isolation containers and improve error detection.

The advent of virtualization technology provides additional opportunities for designing error detection techniques. Processor vendors are including hardware support for virtualization in order to improve performance as well as isolation. Extensions to our work can leverage the strong isolation support provided by such technologies as a replacement for the memory protection techniques currently used to implement protected objects. In this scenario, the hardware's virtualization support is used for improving the security and reliability of a single operating system, rather than supporting multiple operating systems. When coupled with technologies such as the I/O-MMU [93], protected objects that use DMA capabilities in the hardware can detect a larger subset of errors that result in invalid memory accesses.

12.2 Improved Error Recovery

The current version of CuriOS does not attempt to recover from unexpected errors that occur inside the recovery routine of a previous error. In such cases, exceptions that are not handled within the recovery routine result in system failure. It is possible to address this limitation by escalating the original exception upon failure of the recovery routine. This requires additional changes to the protected object wrapper. While our experimental results indicate that this issue is not common, the addition of recursive recovery support may result in a more robust system.

The state-preserving restarts used by CuriOS can be combined with upcoming technologies such as hardware or software transactional memory in order to enhance its error recovery capabilities. Transactional support can be used to roll back SSR state when an error is encountered. This is an improvement over the original design because it allows better recovery of the SSR that is mapped into the server's domain when it encountered an error. Without such transactional support, arbitrary corruption of the mapped-in SSR cannot be addressed.

We performed some limited exploration of software transactional memory for state roll back and the results were promising [27]. Our implementation was based on the RSTM library [94] developed at the University of Rochester. RSTM enables high-performance non-blocking transaction-based code. Our use of the library, on the other hand, was primarily motivated by the object state checkpointing feature. We believe that this direction requires further attention and is a promising avenue for future research. Some related work in this area include the Arjuna system [85] and VINO [84].

12.3 Parallel Computing

The recent push towards multi-core technology has revived interest in the areas of highly parallel computation. While CuriOS was developed and tested on uniprocessor ARM

platforms, it does include complete multiprocessor support, partly due to its roots in the multiprocessor-capable Choices system. We envisage that future extensions to this research will address parallelism and study performance as well as reliability improvements.

Microkernel designs such as CuriOS should perform well on multiprocessor systems when the scheduler ensures some level of processor affinity for the operating system servers. This helps by reducing cache misses and cache synchronization issues. The state separation approach advocated in this thesis can also exploit parallel threads to potentially speed up operating system services. If an operating system server on one core is busy, new requests may be dispatched to a duplicate server on another core. This is possible for stateless servers since any required state is dispatched along with the request.

On the reliability front, it may be possible to replicate services in order to significantly reduce the possibility of failure. Techniques such as N-modular redundancy and process-pairs [68] can be applied in future systems with large numbers of cores. Hardware redundancy helps with many types of hardware faults and a few types of software faults as well. When combined with the error detection and recovery techniques presented in this thesis, such systems can address many additional classes of software faults.

12.4 Other Hardware Architectures

The error detection, signaling and recovery techniques presented in this thesis were developed and are currently implemented only on the ARM architecture. However, the principles and ideas illustrated by CuriOS are not architecture-specific. Porting CuriOS to other popular architectures such as the Intel x86 is a straightforward extension of this work. The exploration of the different features provided by other architectures may reveal additional possibilities for dependability improvements.

Because the Choices operating system, on which CuriOS is based, already runs on several other architectures, the porting effort for CuriOS is not expected to be difficult.

The low-level exception handling, protection and domain-switching optimizations are the primary architecture-dependent portions of CuriOS that will need to be ported for other architectures.

12.5 Other Operating Systems

The state separation approach described in this work may also be applied to other microkernel systems which provide isolation for operating system services such as L4 and Minix3. This would require some modifications to these kernels to incorporate SSR management and changes to server APIs. These systems would need to also be augmented to support the other requirements for transparent recovery detailed in Chapter 2. The benefits of component restarts and state partitioning for operating systems that do not use inter-component isolation is debatable. This is because there are no constraints on error propagation.

Chapter 13

Conclusions

In this thesis, we have described the results of our efforts in building an operating system that is resilient to errors. We analyzed the reasons why current designs for reliable microkernel operating systems struggle with client-transparent recovery. Through simple fault injection experiments with various systems, we gained insights into properties that are essential for successful client-transparent recovery of operating system services. The design and architecture of CuriOS preserves these properties. CuriOS minimizes error propagation and persists client information using distributed and isolated operating system service state. This enhances the transparent restartability of several system components.

We explored several error detection techniques and were able to use language exception handling to efficiently signal many detected errors. Our experimental results show that it is possible to isolate and recover core operating system services from a significant percentage of errors with acceptable performance.

By building CuriOS from Choices, we demonstrated that it is possible to take a modular monolithic operating system and easily convert it to use microkernel design principles. This attests to the power of object-oriented programming.

CuriOS has allowed us to explore many innovative reliability-enhancing techniques. System dependability, however, is still not a solved problem. We hope that our work with CuriOS inspires future research in this area.

The source code for our CuriOS implementation and the code for the QEMU based fault injector can be found on our website at `http://choices.cs.uiuc.edu/`.

Appendix A

The Choices Operating System

Choices is a full featured object-oriented operating system developed at the University of Illinois at Urbana-Champaign. The Choices kernel is written in C++ and is implemented as a dynamic collection of interacting objects. System resources, policies and mechanisms are represented by objects organized in class hierarchies. The system architecture consists of a number of subsystem design frameworks [95] that implement generalized designs, constraints, and a skeletal structure for customizations. Key classes within the frameworks can be subclassed to achieve portability, customizations and optimizations without sacrificing performance [96].

Choices includes an object-oriented framework for file systems [97] and supports several file system formats. Files are treated as persistent memory objects. A collection of file objects is managed by a container object. Further customization using inheritance allows the interpretation of individual file formats using specialized objects. A networking framework was also built for Choices using object-oriented techniques [98].

User applications interact with the kernel through a novel object-oriented interface [99]. A compatibility layer [100] allows users to build UNIX applications to run on Choices.

Choices is designed to be multi-processor capable [101]. It has been ported to and runs on the SPARC [102], Intel x86 [103], ARM [29] and a couple of other platforms [104, 105]. Similar to User Mode Linux [106], a user-mode port called Virtual Choices [107] which runs on Solaris and Linux has also been developed. The design frameworks in Choices are inherited and customized by each hardware specific implementation of the system providing a high degree of reuse and consistency between implementations.

Unlike CuriOS, Choices was not originally designed with support for comprehensive error management. Exception handling was not used in the original operating system and this support was added recently [25]. The object-oriented nature of Choices, however, provided a strong foundation for our reliability research with CuriOS. Choices is currently being used as a learning tool in operating systems classes at the University of Illinois at Urbana-Champaign. It is also being used as an experimental platform for research in operating systems for mobile devices and multi-core processors. Additional information, research publications, and source code are available on the Internet at `http://choices.cs.uiuc.edu/`.

Appendix B

The ARM Processor Architecture

ARM processors power a large percentage of the world's mobile devices [108]. Because most of the research presented in this thesis was performed in the context of this architecture, this Appendix provides a helpful brief introduction. We only describe the the architecture generation found in two specific platforms that we use: the Texas Instruments OMAP1610 based H2 development boards [22] and the ARM Integrator CP platform [24] emulated by QEMU [23]. The processors on these platforms are part of the ARMv5 architecture generation. After we commenced our research, new processors based on the ARMv6 and ARMv7 generation have been released. These new processors incorporate many additional features such as physically tagged caches and more stringent isolation technology called TrustZone. The interested reader is referred to the Internet for information on these additions.

ARM is a 32 bit RISC architecture. It supports a native full-fledged 32-bit instruction set. ARM also specifies two other instruction sets: a 16-bit compressed RISC set called Thumb, and an 8-bit instruction set for Java byte-codes called Jazelle. These are only available on selected versions of the ARM processor core. Both the ARM926EJ-S processor in the TI OMAP1610 H2 board and the ARM1026EJ-S processor in the Integrator support these extended instruction sets.

The ARM processors in both the OMAP and the Integrator support seven modes of operation. There are shown in table B.1. ARM_USR is the unprivileged processor mode that is typically used for user programs. The other six are privileged processor modes and are used by operating systems. Manipulation of security and configuration settings such

Table B.1: ARM processor modes

Mode	Description
ARM_ABT	Abort - used for data and instruction fetch aborts
ARM_FIQ	FIQ - used for fast interrupts
ARM_IRQ	IRQ - used for normal interrupts
ARM_SVC	Supervisor - privileged mode for resets and software interrupts
ARM_SYS	System - privileged mode for operating system code
ARM_UND	Undefined instruction - entered when instruction decode fails
ARM_USR	User - unprivileged mode for user applications

as interrupts and memory management functions can only be performed by code executing in a privileged processor mode. The processor mode is switched when the processor is interrupted or when privileged mode code directly changes it.

The ARM architecture has 37 registers as shown in table B.2. 31 of these registers are general purpose registers including a program counter and 6 are status registers. Some of these registers are banked and are hidden except when executing in specific processor modes. These registers such as the stack pointer are automatically switched when entering a different processor mode. This design allows fast processing of interrupts as the handler code does not need to manually switch to a new stack.

An application normally has access to 16 general purpose registers (R0-R15) and one current program status register (CPSR). The following registers have special meaning: R15 is the program counter (PC), R14 is the link register (LR). By convention, the other general purpose registers are assigned meanings as well: R13 is the stack pointer (SP), R12 is the scratch register (IP), R11 is the frame pointer (FP) and R10 is the stack limit (which is currently unused). Registers R4-R9 are callee-preserved. Registers R0-R3 are used to pass arguments and return values when performing function calls. These register usage conventions are defined by an Application Binary Interface (ABI) [109]. The CPSR register holds the current processor mode, flags that control interrupt delivery and the codes used by conditional instructions. A special set of banked registers, namely, saved program status

Table B.2: ARM registers: Banked registers have an underscore in their names

User	System	Supervisor	Abort	Undefined	Interrupt	Fast Interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
R15	R15	R15	R15	R15	R15	R15
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
-	-	SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

registers (SPSR) are used to save a copy of CPSR when switching modes. These cannot be accessed by code executing in the ARM_SYS or ARM_USR modes.

The ARM processor enters special interrupt handling modes when certain events occur. These events and modes are shown in table B.3. When an interrupt is received, the PC register is set to either 0x00000000+offset or 0xFFFF0000+offset depending on whether the processor is configured to use low or high interrupt vectors.

The ARM processor supports address translation using an MMU (Memory Management Unit). The MMU uses a two-level page table to map virtual to physical addresses. Each first level page table entry can directly specify a mapping for a 1 MB memory region. A first level entry may also indirect to a second level page table, which maps memory in increments of either 16 KB, 4 KB, or 1 KB. Both first level and second level entries include bits that determine access permissions based on processor mode. The entries also include bits that determine the caching policy used for the covered virtual memory region.

Table B.3: ARM interrupt vectors and handling modes

Interrupt	Mode	Cause	Offset
Reset	ARM_SVC	Board is powered on or watchdog reset	0x0
Prefetch Abort	ARM_ABT	An instruction cannot be fetched from memory	0x4
Data Abort	ARM_ABT	A load or store failed	0x8
Illegal opcode	ARM_UND	An instruction could not be decoded	0xC
SWI	ARM_SVC	Software interrupt	0x10
IRQ	ARM_IRQ	An interrupt has occurred	0x14
FIQ	ARM_FIQ	A fast interrupt has occurred	0x18

The ARMv5 architecture uses a split (Harvard) cache architecture with separate instruction and data caches. The cache can be disabled, configured for write-through or write-back for individual pages. An on-chip write buffer can be used to queue writes to memory. It also includes hardware extensions that support fast context switching without requiring a cache flush.

References

[1] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos. Can We Make Operating Systems Reliable and Secure? *IEEE Computer*, 39(5):44–51, 2006.

[2] Brian Randell. Operating Systems: The Problems of Performance and Reliability. In *Proceedings of IFIP Congress 71 Volume 1*, pages 281–290, 1971.

[3] Peter J. Denning. Fault Tolerant Operating Systems. *ACM Computing Survey*, 8(4):359–389, 1976.

[4] Inhwan Lee and Ravishankar K. Iyer. Faults, Symptoms, and Software Fault Tolerance in the Tandem GUARDIAN90 Operating System. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 20–29, 1993.

[5] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 207–222, New York, NY, USA, 2003. ACM Press.

[6] Rolf Johansson. On Single Event Upset Error Manifestation. In *Proceedings of the 1st European Dependable Computing Conference*, pages 217–231, London, UK, 1994. Springer-Verlag.

[7] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. An Empirical Study of Operating System Errors. In *Proceedings of the Symposium on Operating Systems Principles*, pages 73–88, 2001.

[8] Weining Gu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Error Sensitivity of the Linux Kernel Executing on PowerPC G4 and Pentium 4 Processors. In *Proceedings of the International Conference on Dependable Systems and Networks*, page 887, Washington, DC, USA, 2004. IEEE Computer Society.

[9] Long Wang, Zbigniew Kalbarczyk, Weining Gu, and Ravishankar K. Iyer. An OS-level Framework for Providing Application-Aware Reliability. In *Proceedings of the 12th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 55–62, 2006.

[10] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Reorganizing UNIX for Reliability. In *Asia-Pacific Computer Systems Architecture Conference*, pages 81–94, 2006.

[11] Jochen Liedtke. On μ-Kernel Construction. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 237–250, New York, NY, USA, 1995. ACM Press.

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

[13] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – A Technique for Cheap Recovery. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA, December 2004.

[14] Martin Hiller, Arshad Jhumka, and Neeraj Suri. PROPANE: An Environment for Examining the Propagation of Errors in Software. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 81–85, New York, NY, USA, 2002. ACM Press.

[15] Brent Ballinger Welch. *Naming, State Management, and User-Level Extensions in the Sprite Distributed File System*. PhD thesis, University of California, Berkeley, CA 94720, February 1990. Technical Report UCB/CSD 90/567.

[16] Sunil Kittur, François Armand, Douglas Steel, and Jim Lipkis. Fault Tolerance in a Distributed CHORUS/MiX System. In *Proceedings of the USENIX Annual Technical Conference*, pages 219–228, 1996.

[17] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network Filesystem. In *Proceedings of the USENIX Conference*, pages 119–130, Portland, OR, USA, 1985.

[18] Wietse Venema. Murphy's law and computer security. In *Proceedings of the 6th USENIX Security Symposium*, page 187, 1996.

[19] Rick Kennell and Leah H. Jamieson. Establishing the Genuinity of Remote Computer Systems. In *Proceedings of the 12th USENIX Security Symposium*, pages 295–308, 2003.

[20] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harre, George Necula, and Eric Brewer. SafeDrive: Safe and Recoverable Extensions Using Language-Based Techniques. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 45–60, Nov 2006.

[21] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 75–88, Berkeley, CA, USA, 2006. USENIX Association.

[22] Texas Instruments OMAP Platform. http://focus.ti.com/omap/docs/omaphomepage.tsp.

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

[24] ARM™Integrator Family. http://www.arm.com/products/DevTools/Hardware_Platforms.html.

[25] Francis M. David, Jeffrey C. Carlyle, Ellick M. Chan, David K. Raila, and Roy H. Campbell. *Exception Handling in the Choices Operating System*, volume 4119 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 2006.

[26] Francis M. David, Jeffrey C. Carlyle, Ellick M. Chan, Philip A. Reames, and Roy H. Campbell. Improving Dependability by Revisiting Operating System Design. In *Proceedings of the 3rd Workshop on Hot Topics in Dependability*, pages 58–63, Edinburgh, UK, June 2007.

[27] Francis M. David and Roy H. Campbell. Building a Self-Healing Operating System. In *Proceedings of the 3rd IEEE International Symposium on Dependable, Autonomic and Secure Computing*, Columbia, MD, Sep 2007.

[28] Francis M. David, Jeffrey C. Carlyle, and Roy H. Campbell. Exploring Recovery from Operating System Lockups. In *Proceedings of the USENIX Annual Technical Conference*, pages 351–356, Santa Clara, CA, June 2007.

[29] Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, and Roy H. Campbell. Porting Choices to ARM Architecture Based Platforms. Technical Report UIUCDCS-R-2007-2830, University of Illinois at Urbana-Champaign, March 2007.

[30] Francis M. David, Ellick M. Chan, Jeffrey C. Carlyle, and Roy H. Campbell. CuriOS: Improving Reliability through Operating System Structure. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 59–72, San Diego, CA, December 2008.

[31] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

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

[33] Jonathan S. Shapiro. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, 1999.

[34] Jorrit Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Roadmap to a Failure-Resilient Operating System. *USENIX ;login*, 32(1):14–20, February 2007.

[35] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Failure Resilience for Device Drivers. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 41–50, 2007.

[36] Ben Leslie, Carl van Schaik, and Gernot Heiser. Wombat: A Portable User-Mode Linux for Embedded Systems. In *6th Linux.Conf.Au*, April 2005.

[37] Hermann Hartig, Michael Hohmuth, and Jean Wolter. Taming Linux. In *Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems*, Adelaide, Australia, Sept 1998.

[38] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 17–30, San Francisco, CA, December 2004.

[39] Vadim Abrossimov, Frédéric Hemann, Jean-Christophe Hugly, Frédéric Ruget, Eric Pouyoul, and Michel Tombroff. Fast Error Recovery in CHORUS/OS: The Hot-Restart Technology. Technical Report CSI-T4-96-34, Chorus Systems, Inc., August 1996.

[40] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, 1995.

[41] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Brice no, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 52–65, New York, NY, USA, 1997. ACM Press.

[42] Eoin Andrew Hyden. *Operating System Support for Quality of Service*. PhD thesis, University of Cambridge, 1994.

[43] Galen C. Hunt, James R. Larus, Martín Abadi, Mark Aiken, Paul Barham, Manuel Fahndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian Zill. An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

[44] A. J. Wellings, B. Johnson, B. Sanden, J. Kienzle, T. Wolf, and S. Michell. Integrating Object-Oriented Programming and Protected Objects in Ada 95. *ACM Transactions on Programming Languages and Systems*, 22(3):506–539, 2000.

[45] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and Protection in a Single Address Space Operating System. *ACM Transactions on Computer Systems*, 12(4):271–307, 1994.

[46] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi Single-Address-Space Operating System. *Software: Practice and Experience*, 28(9):901–928, July 1998.

[47] Martin Fouts, Tim Connors, Steve Hoyle, Bart Sears, Tim Sullivan, and John Wilkes. Brevix Design 1.01. Technical Report HPL-OSR-93-22, Hewlett-Packard Laboratories, 1993.

[48] Partha Dasgupta, Jr. Richard J. LeBlanc, Mustaque Ahamad, and Umakishore Ramachandran. The Clouds Distributed Operating System. *IEEE Computer*, 24(11):34–44, 1991.

[49] Advantages of Exceptions. `http://java.sun.com/docs/books/tutorial/essential/exceptions/advantages.html`.

[50] Don Cameron, Paul Faust, Dmitry Lenkov, and Michey Mehta. A Portable Implementation of C++ Exception Handling. In *USENIX C++ Conference*, pages 225–243. USENIX, August 1992.

[51] Free Standards Group: DWARF Debugging Information Format. `http://dwarfstd.org`.

[52] Codesourcery. `http://www.codesourcery.com`.

[53] Exception Handling ABI for the ARM™architecture. `http://www.arm.com/miscPDFs/8034.pdf`.

[54] Halldor Isak Glyfason and Gisli Hjalmtysson. Exceptional Kernel: Using C++ Exceptions in the Linux Kernel. `http://netlab.ru.is/exception/KernelExceptions.pdf`, October 2004.

[55] Christophe de Dinechin. C++ Exception Handling. *IEEE Concurrency*, 8(4):72–79, 2000.

[56] Herbert Bos and Bart Samwel. Safe Kernel Programming in the OKE. In *IEEE Open Architectures and Network Programming*, pages 141–152, 2002.

[57] i.MX31 Multimedia Applications Processor. `http://www.freescale.com/files/32bit/doc/ref_manual/MCIMX31RM.pdf`.

[58] International Business Machines Corporation. AIX 5L Version 5.1 System Management Concepts: Operating System and Devices - 3rd Edition. pages 9–10, 2001.

[59] Keith Whisnant, Ravishankar K. Iyer, Zbigniew T. Kalbarczyk, Phillip H. Jones III, David A. Rennels, and Raphael Some. The Effects of an ARMOR-Based SIFT Environment on the Performance and Dependability of User Applications. *IEEE Transactions on Software Engineering*, 30(4):257–277, 2004.

[60] Niall Murphy and Michael Barr. Watchdog Timers. *Embedded Systems Programming*, pages 79–80, Oct 2001.

[61] Soft lockup detector extension. `http://lkml.org/lkml/2005/8/2/216`, Aug 2005.

[62] Nithin Nakka, Zbigniew Kalbarczyk, Ravishankar K. Iyer, and Jun Xu. An Architectural Framework for Providing Reliability and Security Support. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 585–594, 2004.

[63] Craig Chaiken and Stan Stanart. United States Patent 6587966: Operating system hang detection and correction. July 2003.

[64] Gary Hicok. United States Patent 7010724: Operating system hang detection and methods for handling hang conditions. March 2006.

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

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

[67] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The Rio File Cache: Surviving Operating System Crashes. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 74–83, 1996.

[68] Jim Gray. Why Do Computers Stop and What Can Be Done About It? In *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, January 1986.

[69] Compressed ROM filesystem. `http://sourceforge.net/projects/cramfs/`.

[70] Adam Dunkels. Full TCP/IP for 8-bit Architectures. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, pages 85–98, New York, NY, USA, 2003. ACM.

[71] Wee Teck Ng and Peter M. Chen. The Systematic Improvement of Fault Tolerance in the Rio File Cache. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, pages 76–83, 1999.

[72] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian Memory Protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 304–316, New York, NY, USA, 2002. ACM Press.

[73] Robert Hanmer. *Patterns for Fault Tolerant Software*. Wiley, 2007.

[74] Brian Demsky and Martin Rinard. Automatic Data Structure Repair for Self-Healing Systems. In *Proceedings of the 1st Workshop on Algorithms and Architectures for Self-Managed Systems*, San Diego, California, June 2003.

[75] NTT Corporation. Pannus. `http://pannus.sourceforge.net/`.

[76] Jeff Arnold and Frans Kaashoek. Ksplice: Automatic Rebootless Kernel Updates. `http://www.ksplice.com`.

[77] Andrew Baumann, Jonathan Appavoo, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, and Gernot Heiser. Reboots are for Hardware: Challenges and Solutions to Updating an Operating System on the Fly. In *Proceedings of the USENIX Annual Technical Conference*, pages 337–350, Berkeley, CA, USA, 2007. USENIX Association.

[78] Wilfredo Torres-Pomales. Software Fault Tolerance: A Tutorial. Technical Report NASA/TM-2000-210616, NASA Langley Research Center, 2000.

[79] Joel F. Bartlett. A NonStop Kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 22–29, New York, NY, USA, 1981. ACM Press.

[80] Wendy Bartlett and Lisa Spainhower. Commercial Fault Tolerance: A Tale of Two Systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96, 2004.

[81] Joel Bartlett, Jim Gray, and Bob Horst. Fault Tolerance in Tandem Computer Systems. In A. Avizienis, H. Kopetz, and J.-C. Laprie, editors, *The Evolution of Fault-Tolerant Systems*, pages 55–76. Springer-Verlag, Vienna, Austria, 1987.

[82] The Standish Group. TCO in the Trenches 2002. `http://www.himalaya.compaq.com/object/TCO.html`.

[83] Rober Haskin, Yoni Malachi, and Gregory Chan. Recovery Management in QuickSilver. *ACM Transactions on Computer Systems*, 6(1):82–108, 1988.

[84] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 213–227, New York, NY, USA, 1996. ACM.

[85] Graham D. Parrington, Santosh K. Shrivastava, Stuart M. Wheater, and Mark C. Little. The Design and Implementation of Arjuna. *Computing Systems*, 8(2):255–308, 1995.

[86] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering Device Drivers. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 1–16, 2004.

[87] George Candea and Armando Fox. Crash-Only Software. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, Lihue, HI, May 2003.

[88] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, 1995.

[89] Fred J. Pollack, Kevin C. Kahn, and Roy M. Wilkinson. The iMAX-432 Object Filing System. *SIGOPS Oper. Syst. Rev.*, 15(5):137–147, 1981.

[90] Kevin C. Kahn, William M. Corwin, T. Don Dennis, Herman D'Hooge, David E. Hubka, Linda A. Hutchins, John T. Montague, and Fred J. Pollack. iMAX: A Multiprocessor Operating System for an Object-Based Computer. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 127–136, New York, NY, USA, 1981. ACM.

[91] John Holford and George M. Mohay. ProtectOS: Operating System and Hardware Support for Small Objects. In *Proceedings of the 2nd Australasian Conference on Information Security and Privacy*, pages 102–113, 1997.

[92] Jerry Vochteloo, Kevin Elphinstone, Stephen Russell, and Gernot Heiser. Protection Domain Extensions in Mungi. In *Proceedings of the 5th International Workshop on Object Orientation in Operating Systems*, page 161, Washington, DC, USA, 1996. IEEE Computer Society.

[93] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. Intel Virtualization Technology for Directed I/O. *Intel Technology Journal*, 10(3):179–191, August 2006.

[94] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, and Michael L. Scott. Lowering the Overhead of Software Transactional Memory. Technical Report TR 893, Computer Science Department, University of Rochester, Mar 2006.

[95] Roy H. Campbell, Nayeem Islam, Ralph Johnson, Panos Kougiouris, and Peter Madany. *Choices*, Frameworks and Refinement. In Luis-Felipe Cabrera and Vincent

Russo, and Marc Shapiro, editor, *Object-Orientation in Operating Systems*, pages 9–15, Palo Alto, CA, October 1991. IEEE Computer Society Press.

[96] Vincent F. Russo, Peter W. Madany, and Roy H. Campbell. C++ and Operating Systems Performance: a Case Study. In *USENIX C++ Conference*, pages 103–114, San Francisco, CA, April 1990.

[97] Peter W. Madany. *An Object-Oriented Framework for File Systems*. PhD thesis, University of Illinois at Urbana-Champaign, May 1992.

[98] Jonathan Zweig and Ralph E. Johnson. The Conduit: A Communication Abstraction in C++. In *Proceedings of the USENIX C++ Conference*, pages 191–204, San Francisco, California, April 1990.

[99] David Dysktra. *Object-Oriented Hierarchies Across Protection Boundaries*. PhD thesis, University of Illinois at Urbana-Champaign, May 1992.

[100] John L. Coolidge. The Choices Unix Compatibility Framework. Master's thesis, University of Illinois at Urbana-Champaign, 1992.

[101] R. H. Campbell, V. Russo, and G. M. Johnston. "The Design of a Multiprocessor Operating System". In *Proceedings of the USENIX C++ Workshop*, pages 109–125, Santa Fe, New Mexico, November 1987.

[102] David Raila. The Choices Object-Oriented Operating System on the SPARC Architecture. Master's thesis, University of Illinois at Urbana-Champaign, Aug 1992.

[103] Lup Lee. PC-Choices Object-Oriented Operating System. Master's thesis, University of Illinois at Urbana-Champaign, Aug 1992.

[104] Bjorn Andrew Helgaas. Porting the Choices Object-oriented Operating System to the Motorola 68030. Master's thesis, University of Illinois at Urbana-Champaign, May 1991.

[105] Roy H. Campbell, Gary M. Johnston, Peter W. Madany, and Vincent F. Russo. Principles of Object-Oriented Operating System Design. Technical Report UIUCDCS-R-89-1510, University of Illinois at Urbana-Champaign, April 1989.

[106] Jeff Dike. A user-mode port of the Linux kernel. In *Proceedings of the 4th Annual Linux Showcase and Conference*, Atlanta, Georgia, October 2000.

[107] See-Mong Tan, David K. Raila, Willy S. Liao, and Roy H. Campbell. Virtual Hardware for Operating System Development. Technical report, University of Illinois at Urbana- Champaign, September 1995.

[108] ARM Press Release. Five Billionth ARM Processor for Mobile Devices. `http://www.arm.com/news/16535.html`, Feb 2007.

[109] Application Binary Interface for the ARM™architecture. `http://www.arm.com/miscPDFs/8061.pdf`, March 2005.

Author's Biography

Francis Manoj David was born in Pondicherry, India on February 3, 1980. He graduated from the Indian Institute of Technology Madras, India in 2001 with a bachelor's degree in Computer Science. He relocated to Champaign, Illinois to pursue graduate study in Computer Science. In 2008, he obtained a doctoral degree from the University of Illinois at Urbana-Champaign while working under the supervision of Professor. Roy H. Campbell. His research spans techniques to improve the dependability of computer systems, focusing in particular on the security and reliability of the operating system. He has presented the results of his research at several influential conferences. His work on hardware supported malware received the best student paper award at the prestigious IEEE Symposium on Security and Privacy held at Oakland in 2008.