ETH Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Diploma Thesis

TopsySMP

A Small Multi-Threaded Microkernel for Symmetrical Multiprocessing Hardware Architectures

Dominik Moser

November 1998 – March 1999

Tutor: George Fankhauser
Supervisor: Prof. Dr. Bernhard Plattner

TIK Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

This thesis has been written using $\LaTeX\,2_\varepsilon$ and the KOMA-SCRIPT styles.

To Janine

Abstract

A symmetric multiprocessor (SMP) is the most commonly used type of multiprocessor system. All CPUs and I/O devices are tightly coupled over a common bus, sharing a global main memory to which they have symmetric and equal access. Compared to a uniprocessor system, an SMP system imposes a high demand for memory bus bandwidth. The maximum number of CPUs that can be used for practical work is limited by this bandwidth, being proportional to the number of processors connected to the memory bus. To reduce memory bus bandwidth limitations, an SMP implementation should use a secondary cache and a cache-consistency protocol.

Since the need to synchronize access to shared memory locations is so common on SMP systems, most implementations provide the basic hardware support for this through atomic *read-modify-write* operations. However, since the sequential memory model does not guarantee a deterministic ordering of simultaneous reads and writes to the same memory location from more than one CPU, any shared data structure can cause a race condition to occur. Such nondeterministic behavior can be catastrophic to the integrity of the OS kernel's data structures and must be prevented.

The operating system for a multiprocessor must be designed to coordinate simultaneous activity by all CPUs and maintain system integrity. To simplify their design, many uniprocessor kernel systems have relied on the fact that there is never more than one process executing in the kernel at the same time. However, this policy fails on SMP systems when kernel code can be executed simultaneously on more than one processor. Therefore, a uniprocessor kernel cannot be run on an SMP system without modification.

The easiest way to maintain system integrity within an SMP kernel is with *spin locks*. Spin locks work correctly for any number of processors but are only efficient if the associated critical section is short. Overall system performance will be lowered if the processors spend too much time waiting to acquire locks or if too many processors frequently contend for the same lock. To reduce lock contention, the kernel has to use different spin locks for different critical sections.

Overall system performance can be significantly improved by allowing parallel kernel activity on multiple processors. The amount of concurrent kernel activity that is possible across all the processors is partially determined by the degree of multithreading. A *coarse-grained* implementation uses few locks, whereas a *fine-grained* implementation protects different data structures with different locks. The goal is to ensure that unrelated processes can perform their kernel activities without contention.

This thesis presents *TopsySMP*, a small multithreaded microkernel for an SMP architecture. It is based on Topsy which has been designed for teaching purpose at the Department of Electrical Engineering at ETH Zürich. It consists of at least three independent kernel modules each represented by a control thread: the thread manager, the memory manager, and the I/O manager. Besides, there are a number of device driver threads, and a simple command-line shell. TopsySMP provides parallel thread scheduling, and allows threads to be pinned to specific processors. The number of available CPUs is determined upon startup which leads to a dynamic configuration of the kernel. The uniprocessor system call API is preserved, therefore all applications written for the original Topsy can run on the SMP port.

This thesis shows that the implementation of an SMP kernel based on a multithreaded uniprocessor kernel is straightforward and results in a well structured and clear design. The amount of concurrent kernel activity is determined by the degree of multithreading and leads to a coarse-grained implementation using only a few spin locks. The spin lock times for a small-scale system with up to eight processors are reasonably short compared to a context switch; no other synchronization primitives are used within the kernel. The overhead caused by the integration of SMP functionality was kept to a minimum, resulting in a small and efficient kernel implementation.

Furthermore, a suggestion is made on how to improve system performance by multiplicating the control thread of a kernel module, allowing the throughput of module specific kernel activity to be ideally multiplied.

Zusammenfassung

Der symmetrische Multiprozessor (SMP) ist die bekannteste Multiprozessor-Architektur. Alle CPUs und I/O Geräte sind lokal über einen gemeinsamen Bus verbunden und teilen sich einen globalen Hauptspeicher, zu dem sie symmetrischen, gleichberechtigten Zugang haben. Verglichen mit einem Einprozessorsystem stellt eine SMP-Architektur viel höhere Anforderungen an die Busbandbreite. Die maximale Anzahl CPUs, die für eine gegebene Architektur verwendet werden kann, ist durch die Busbandbreite limitiert, welche proportional zur Anzahl vorhandener Prozessoren ist. Um die Buszugriffe zu verringern, sollte eine SMP-Architektur sowohl 2nd-level Cache-Speicher als auch Cache-Konsistenz Protokolle verwenden.

Da beim Zugriff auf einen globalen Hauptspeicher fast immer Synchronisationsmechanismen benötigt werden, stellen viele SMP-Architekturen hardwaremässig eine read-modify-write Operation zur Verfügung. Diese Operation erlaubt es einem Prozessor einen Wert aus dem Speicher zu lesen, diesen zu verändern und innerhalb der gleichen Bus-Operation wieder in den Speicher zu schreiben. Das verwendete Speichermodell garantiert jedoch keine Ordnung beim gleichzeitigen Zugriffen auf identische Speicheradressen. Somit kann es zu Wettbewerbssituationen (race conditions) kommen, die die Integrität der Datenstrukturen des Kernels verletzen.

Ein Betriebssystem für eine Multiprozessor-Architektur muss so geschaffen sein, dass es die gleichzeitigen Aktivitäten aller Prozessoren koordiniert und die Integrität des Systems gewährleistet. Die Entwickler der meisten herkömmlichen Betriebssysteme haben sich die Sache dadurch vereinfacht, dass sie davon ausgingen, dass sich jeweils nur ein Prozess gleichzeitig im Kernel-Modus befinden kann. Diese Annahme kann jedoch nicht ohne weiteres auf ein Multiprozessor-System übertragen werden.

Die einfachste Art, um den Kernel vor gleichzeitigem, unkontrollierten Zugriff zu schützen, stellt die Verwendung von Spin-Locks dar. Spin-Locks funktionieren mit einer beliebigen Anzahl Prozessoren, sind jedoch nur dann effizient, wenn die geschützten Programmabschnitte kurz sind. Die Gesamtleistung des Systems wird massiv abnehmen, wenn Prozessoren zuviel Zeit beim Warten auf Spin-Locks verbrauchen bzw. wenn zuviele Prozessoren sich um den gleichen Spin-Lock streiten. Um das zu vermeiden, sollte der Kernel über mehrere Spin-Locks für verschiedene kritische Abschnitte verfügen. Der Gesamtdurchsatz des Systems kann nur dann erhöht werden, wenn der Kernel parallelen Zugriff von verschiedenen Prozessoren zulässt. Der maximale Grad der Parallelität ist dabei durch die Aufteilung des Kernels in unabhängige Threads gegeben.

In Laufe dieses Berichts wird *TopsySMP* vorgestellt, ein kleiner Mikrokernel für eine SMP-Architektur. Entstanden ist er aus Topsy, einem Betriebssystem, dass an der ETH Zürich für Unterrichtszwecke entwickelt wurde. Der Kernel besteht aus drei unabhängigen Modulen mit je einem dazugehörigen Thread: Thread-Manager, Memory-Manager und I/O-Manager. Daneben gibt es einen Idle-Thread pro Prozessor, eine Anzahl Gerätetreiber und eine einfach Kommandozeilen-Shell. TopsySMP unterstützt paralleles Thread-Scheduling und erlaubt es, Threads zur Laufzeit an Prozessoren zu binden. Die Anzahl der vorhandenen Prozessoren wird während der Startphase bestimmt und führt zu einer dynamischen Konfiguration des Kernels. Das Systemcall-API von Topsy wurde unverändert übernommen. Somit laufen alle Programme, die für das ursprüngliche Topsy geschrieben wurden ohne Änderung auch auf TopsySMP.

Diese Diplomarbeit zeigt, dass die Implementierung eines SMP-Kernels basierend auf einem bestehenden Einprozessor-Kernel einfach möglich ist und zu einem klaren, strukturierten Design führt. Der Grad an Kernel-Parallelität ist durch die Strukturierung des Kernels in einzelne Threads gegeben. Die Beibehaltung der Strukturierung von Topsy führte dazu, dass nur wenige Spin-Locks benötigt werden. Die Spinlock-Zeiten – gemessen auf einem SMP-System mit bis zu acht Prozessoren – sind verglichen mit der Zeit für einen Kontextwechsel kurz. Daher werden keine zusätzlichen Synchronisationsmechanismen benötigt. Der zusätzliche Verwaltungsaufwand für die SMP-Version hält sich dabei in Grenzen und hat keine messbaren Auswirkungen auf die Gesamtleistung des Systems.

Contents

Preface xi

I GENERAL CONCEPTS OF MULTIPROCESSING 1

1 Introduction to Multiprocessing 3

1.1 Project Goals 5

1.2 Development Environment 5

2 Review of Kernel Internals 7

2.1 Introduction 7

2.2 Processes, Programs, and Threads 8

2.2.1 Context Switch 9

2.3 Summary 10

3 Multiprocessor Hardware Architectures 11

3.1 Single vs. Multiple Instruction Stream 11

3.2 Message-Passing vs. Shared-Memory Architectures 12

3.3 Cache Coherence 15

3.3.1 Uniprocessor Caches 15

3.3.2 Multiprocessor Caches 15

3.4 Scalability 17

3.4.1 Scalable Interconnection Networks 18

3.4.2 Scalability of Parallel Software 18

3.5 Performance Considerations 19

3.6 Summary 20

4 Multiprocessor Kernel Architectures **21**

4.1 MP Operating System . . . 21
4.2 The Tightly Coupled, Shared Memory, Symmetric Multiprocessor . . . 22
4.3 The MP Memory Model . . . 23
4.3.1 The Sequential Memory Model . . . 23
4.3.2 Atomic Reads and Writes . . . 23
4.3.3 Atomic Read-Modify-Write Operations . . . 23
4.4 Mutual Exclusion . . . 24
4.5 Review of Mutual Exclusion on Uniprocessor UNIX Systems . . . 25
4.5.1 Short-Term Mutual Exclusion . . . 25
4.5.2 Mutual Exclusion with Interrupt Handlers . . . 25
4.5.3 Long-Term Mutual Exclusion . . . 26
4.6 Problems Using UP Mutual Exclusion Policies on MPs . . . 27
4.7 Summary . . . 28

5 Multiprocessor Kernel Implementations **29**

5.1 Master-Slave Kernels . . . 29
5.1.1 Spin Locks . . . 30
5.1.2 Master-Slave Kernel Implementation . . . 31
5.1.3 Performance Considerations . . . 32
5.2 Spin-Locked Kernels . . . 32
5.2.1 Giant Locking . . . 33
5.2.2 Coarse-Grained Locking . . . 33
5.2.3 Fine-Grained Locking . . . 33
5.2.4 Kernel Preemption . . . 34
5.2.5 Performance Considerations . . . 34
5.3 Semaphored Kernels . . . 34
5.3.1 Mutual Exclusion with Semaphores . . . 35
5.3.2 Synchronization with Semaphores . . . 35
5.3.3 Resource Allocation with Semaphores . . . 35
5.3.4 Implementing Semaphores . . . 36
5.4 Summary . . . 36

II REVIEW OF EXISTING MP SYSTEMS 39

6 SMP Hardware 41

6.1 SPARC Multiprocessor System Architecture 41
6.1.1 Multi-level Bus Architecture 42
6.1.2 MBus Multiprocessor System Design 42
6.1.3 Multiprocessor System Implementation 42
6.2 Intel Multiprocessor Specification 43
6.2.1 System Overview 43
6.2.2 Hardware Overview 43
6.2.3 BIOS Overview 44
6.2.4 MP Configuration Table 45
6.2.5 Default Configurations 45

7 SMP Operating Systems 47

7.1 Solaris 47
7.1.1 System Overview 47
7.1.2 SunOS Kernel Architecture 48
7.2 Mach 50
7.2.1 The Mach Microkernel 50
7.2.2 Process Management 51
7.3 Linux 53
7.3.1 Evolution of Linux SMP 54
7.3.2 Changes to the Kernel Components 54
7.3.3 Architecture Specific Code for the Intel MP Port 55

III TOPSY SMP 57

8 The Topsy Operating System 59

8.1 System Overview 59
8.2 Thread Management 60
8.3 Memory Management 61
8.4 I/O 62
8.5 User Programs 62

9 An SMP Machine for Topsy **63**

9.1 Choosing SimOS as a Simulation Environment . . . 63

9.2 Porting Topsy to MIPS R4000 . . . 64

9.2.1 Memory Management . . . 64

9.2.2 Exception Handling . . . 66

9.2.3 MIPS R4000 Synchronization Primitives . . . 67

9.3 Porting Device Drivers to SimOS . . . 70

9.3.1 Console Device Driver . . . 70

9.3.2 Clock Device Driver . . . 72

9.4 Adding additional Devices to SimOS . . . 73

9.4.1 Definition of the Device Register Set . . . 73

10 Design of TopsySMP **77**

10.1 Principal Design Goals of TopsySMP . . . 77

10.2 SMP Operating System Design Issues . . . 77

10.2.1 Simplicity . . . 78

10.2.2 Multithreading . . . 78

10.2.3 Kernel Design . . . 78

10.2.4 High degree of parallel Kernel Activity . . . 79

10.2.5 Parallel Thread Scheduling . . . 81

10.2.6 Efficient Synchronization Primitives . . . 81

10.2.7 Uniprocessor API . . . 83

10.2.8 Scalability . . . 83

11 Implementation of TopsySMP **85**

11.1 Implementation Steps . . . 85

11.1.1 SMP System Configuration . . . 85

11.1.2 Data Structures . . . 86

11.1.3 Bootstrapping . . . 87

11.1.4 Scheduler . . . 89

11.1.5 System call API . . . 90

11.1.6 User Thread Exit . . . 90

11.1.7 Exception Handler . . . 92

11.1.8 Synchronization Primitives . . . 92

11.1.9 Enhancement of Parallel Kernel Activity . . . 98

12 Performance Analysis of TopsySMP 101

12.1 Introduction 101
12.2 Simulation Environment 101
12.3 Benchmarks 102
12.3.1 Sum 102
12.3.2 Reserve 102
12.3.3 Syscall 103
12.4 Benchmark Results 103
12.5 Kernel Locking 104
12.5.1 Spinlock Times vs. Context Switch 104
12.5.2 Complex locking strategies 105
12.5.3 Detailed Spin Lock Times 105
12.6 Internal Kernel Operations 105
12.6.1 Exception Handling 105
12.6.2 System Call 105
12.6.3 SMP Management Overhead 106

13 Conclusions 109

13.1 Future work 110

IV APPENDIX 111

A MIPS R4000 Architecture 113

A.1 Introduction 113
A.2 Processor General Features 113
A.3 Memory Management 114
A.3.1 System Control Coprocessor, CP0 114
A.3.2 Format of a TLB Entry 114
A.3.3 CP0 Registers 117
A.4 Exception Handling 119
A.4.1 Exception Processing Registers 120
A.4.2 Exception Vector Location 123

A.5 Instructions Set Details . . . 123
A.5.1 ERET – Exception Return . . . 124
A.5.2 LL – Load Linked . . . 124
A.5.3 SC – Store Conditional . . . 124

B SimOS 125

B.1 Introduction . . . 125
B.2 The SimOS Environment . . . 126
B.2.1 Interchangeable Simulation Models . . . 126
B.3 Data Collection Mechanisms . . . 128

C Project Description 131

C.1 Einleitung . . . 131
C.2 Mehrprozessorsysteme mit gemeinsamem Speicher . . . 131
C.3 Aufgabenstellung . . . 132
C.3.1 Verwandte Arbeiten . . . 132
C.3.2 Plattform . . . 132
C.3.3 Der SMP-Kernel . . . 132
C.3.4 Testen . . . 133
C.3.5 Messungen . . . 133
C.3.6 Dokumentation . . . 133
C.4 Bemerkungen . . . 133
C.5 Ergebnisse der Arbeit . . . 134

Glossary 135

Bibliography 136

List of Figures

2.1 Logical layering of the UNIX system. 7

3.1 Single Instruction Multiple Data (SIMD) 12
3.2 Multiple Instruction Multiple Data (MIMD) 12
3.3 Programmer's View of (a) Message-Passing and (b) Shared-Memory Architectures . 13
3.4 (a) Distributed-Memory and (b) Shared-Memory Architectures 14
3.5 Example of an invalidation protocol 16
3.6 Bus-Based MP with a simple Snoopy Cache Structure 16

4.1 Example SMP block diagram. 22
4.2 Test-and-set implementation using swap-atomic. 24
4.3 Code to lock an object. 26
4.4 Code to unlock an object. 27

5.1 Initializing a spin lock. 30
5.2 Atomically locking a spin lock. 30
5.3 Unlocking a spin lock. 30
5.4 Critical region protected by a semaphore. 35
5.5 Resource reservation with semaphores. 36

6.1 SPARC/MBus Multiprocessing Architecture 41
6.2 Intel MP System Architecture 43
6.3 MP Configuration Data Structures 45

7.1 Abstract Model for UNIX Emulation using Mach 51

8.1 Modular structure of Topsy 59
8.2 User and Kernel Address Space 61

8.3 Virtual to Physical Address Mapping of the MIPS R3000A 61

9.1 Format of a R3k TLB Entry 64
9.2 Test-and-Set using LL and SC 68
9.3 lock function in Topsy 69
9.4 testAndSet function in TopsySMP 69
9.5 unlock function in Topsy 70
9.6 Device Driver Service Routine 71
9.7 Handling Clock Reset 72
9.8 Macro Definition for a SimOS Device 74
9.9 Service Routine for the IPIC Device 75
9.10 Interprocessor Communication using the IPIC Device 75

10.1 Inner Loop of the Scheduler in Topsy. 78
10.2 Modified System Call Interface. 80

11.1 Starting idle thread(s) in tmMain(). 86
11.2 Function smpBootCPUs. 88
11.3 Inner Scheduler Loop of TopsySMP 89
11.4 Function schedulerRemove 91
11.5 Function schedulerRemoveExit 91
11.6 Exception Handler Code 92
11.7 Function semInit 93
11.8 Function semP 93
11.9 Function semV 94
11.10 Function mrlockInit 95
11.11 Function mrEnterReader 95
11.12 Function mrExitReader 96
11.13 Function mrEnterWriter 96
11.14 Function mrExitWriter 97

12.1 Benchmark Results 103

A.1 Block Diagram of the MIPS R4000 114

A.2 CP0 Registers and the TLB 115
A.3 Format of a TLB Entry 115
A.4 Fields of the PageMask and EntryHi Registers 116
A.5 Fields of the EntryLo0 and EntryLo1 Registers 116
A.6 Wired Register Boundary 118
A.7 Wired Register 118
A.8 Processor Revision Identifier Register Format 119
A.9 Status Register 120
A.10 Cause Register Format 122

B.1 The SimOS Environment 126
B.2 Speed vs. Details 127
B.3 Annotation Script for SimOS 129
B.4 Processor Mode Breakdown 130

List of Tables

6.1 MP Configuration Entry Types 46

9.1 Differences between R3k and R4k 64
9.2 TLB entries for Topsy on R3k 65
9.3 TLB entries for Topsy on R4k 65
9.4 Mapping of the Interrupt Ids between SimOS and IDT-Board 72

12.1 UP vs. MP with one Processor 106

A.1 TLB Page Coherency Bit Values 117
A.2 Wired Register Field Descriptions 118
A.3 PRId Register Field Descriptions 119
A.4 Status Register Fields 121
A.5 Cause Register Fields 122
A.6 Cause Register ExcCode Field 122
A.7 Exception Vector Base Addresses 123
A.8 Exception Vector Offsets 123

Preface

For much of the history of computer system development, the desire to build faster overall systems has focused on making the three main components of a system — the central processing unit (CPU), the memory subsystem, and the I/O subsystem — all faster. Faster CPUs are built by increasing the clock speed or using instruction pipelines. Faster memory subsystems are built by reducing the access time. Faster I/O subsystems are built by increasing the data transfer rate and replicating buses. As clock speeds and transfer rates increase, however, it becomes increasingly more difficult, and therefore expensive, to design and build such systems. Propagation delays, signal rise time, clock synchronization and distribution, and so forth all become more critical as speeds increase. The increased cost of designing for such high speeds makes it more difficult to achieve an effective price-performance ratio. Because of this and other factors, system designers have widened their focus to find other ways to increase overall system performance and thus frequently look to multiprocessors as an alternative.

This report concentrates on one of the many aspects of multiprocessing: The design and implementation of a small multithreaded microkernel for a shared-memory architecture. Is is the result of a Diploma Thesis written at the Computer Engineering and Networks Laboratory at ETH Zürich.

Structure of the Thesis

Although this thesis can be read in sequence from cover to cover, it can also serve as a reference guide to specific design components and procedures.

Part I, "General Concepts of Multiprocessing", deals with the basic concepts behind multiprocessor implementations, both from the hardware and the software point of view.

Chapter 1, "Introduction to Multiprocessing", gives a short introduction to multiprocessor systems, and names the basic goals of this project. Furthermore, it takes a look at the development environment used throughout this thesis.

Chapter 2, "Review of Kernel Internals", reviews kernel internals to provide context and background for the operating system portion of this thesis. UNIX was chosen as an example because most of todays operating systems are based somehow on the original UNIX design.

Chapter 3, "Multiprocessor Hardware Architecture", introduces various multiprocessor architectures and focuses on system scalability.

Chapter 4, "Multiprocessor Kernel Architectures", introduces the tightly coupled, shared memory multiprocessor and describes its organization in preparation for subsequent chapters that examine how the UNIX operating system can be adapted to run on this type of hardware.

Chapter 5, "Multiprocessor Kernel Implementations", takes the reader deeper into the multiprocessor environment by introducing three techniques for modifying a uniprocessor kernel implementation to run on an SMP system without race conditions: the *master-slave* kernel, the *spin-locked* kernel, and the *semaphored* kernel.

Part II, "Review of Existing MP Systems", examines existing multiprocessor hardware and operating systems.

Chapter 6, "SMP Hardware", reviews two existing multiprocessor hardware architectures: The *SPARC Multiprocessor System Architecture* from Sun, and the *Multiprocessor Specification* from Intel.

Chapter 7, "SMP Operating Systems", focuses on the software aspect of multiprocessor systems by providing a study on the kernel implementations of Solaris (SunOS), Mach, and Linux.

Part III, "TopsySMP", covers the main result of this thesis, an SMP port of the Topsy operating system, called *TopsySMP*.

Chapter 8, "The Topsy Operating System", reviews the design and implementation of the Topsy operating system on which this thesis is based.

Chapter 9, "An SMP Machine for Topsy", covers the porting of the existing Topsy kernel to the MIPS R4000 processor family and the computer simulation environment SimOS from Stanford University.

Chapter 10, "Design of TopsySMP", details the design goals of an SMP operating system in general, and of TopsySMP in particular.

Chapter 11, "Implementation of TopsySMP", presents the implementation details of TopsySMP.

Chapter 12, "Performance Analysis of TopsySMP", presents the results of the performance measurements using different benchmarks.

Chapter 13, "Conclusions", concludes this thesis and offers a glimpse of future work based on this work.

Appendix A, "MIPS R4000 Architecture", conveys the information needed to understand the hardware abstraction layer (HAL) of the TopsySMP operating system.

Appendix B, "SimOS", examines some details of the SimOS computer simulation environment from Stanford University.

Appendix C, "Project Description", includes the project description (in German) of this thesis.

The **Glossary** defines unfamiliar terms.

The **Bibliography** provides a list of references used for writing this thesis.

Acknowledgments

I'd like to express my gratitude to the many people who helped making this thesis possible. First and foremost to my supervisor, George Fankhauser, whose support, encouragement, and enthusiasm were instrumental in realizing this work. Thanks to Prof. Dr. Bernhard Plattner, head of the Communication Systems group at TIK, allowing me to perform this thesis at his research group.

Some of this work was furthered by e-mail exchange with researchers from the SimOS group at Stanford University. I'd like to thank Edouard Bugnion for his patience and for explaining me some of the internals of SimOS. Special thanks to Ben Gamsa from the Department of Computer Science at the University of Toronto. His tips on how to port an operating system to SimOS were of great help.

Finally, I'd like to thank my parents for making this all possible.

Zürich, March 10, 1999

Part I

GENERAL CONCEPTS OF MULTIPROCESSING

1 Introduction to Multiprocessing

A *multiprocessor (MP)* consists of two or more CPUs combined to form a single computer system. With the multiprocessor approach, the designer alleviates the need to build higher speed CPUs by instead making multiple CPUs available. The workload can then be distributed across all available CPUs. If we compare a uniprocessor (UP) system and a multiprocessor system designed with the same CPU, the multiprocessor will typically not perform any task faster than the uniprocessor, since the CPU speeds are the same, but it can perform more tasks in parallel per unit time. This is the primary appeal of a multiprocessor: more tasks performed per unit time using more economical CPU technology than if one tried to build a uniprocessor capable of processing the same task load in the same amount of time. In addition, some applications can be rewritten to make use of the inherent parallelism of an MP system. The application can be divided into a set of cooperating subprograms, each of which executes on different processors. In this case, the time required to run the application can be reduced.

Multiprocessing provides advantages from a marketing standpoint as well. Multiprocessor systems can be scaled by adjusting the number of CPUs to fit the application environment. This is appealing to end users and customers who can start out with a one- or two-processor system, for example, and upgrade it by adding CPUs as their computing needs expand. In addition, there is the possibility of increased system availability. If one CPU were to fail, the remaining CPUs may be able to continue functioning, maintaining system availability but at reduced performance. This provides a degree of *fault tolerance*.

The idea of multiprocessors dates back to the first electronic computers. As early as the 1950s, MPs were advanced as a technique both to increase reliability and to improve performance. In the early 1980s, the first commercially successful multiprocessors became available. Almost all of these designs were bus-based, shared-memory machines. Their development was enabled by two key factors: the incredible price-performance advantages of microprocessors and the extensive use of caches in multiprocessor-based systems. These technological factors made it possible to put multiple processors on a bus with a single memory.

Multiprocessor systems have long held the promise of substantially higher performance than traditional uniprocessor systems. Because of a number of difficult problems, however, the potential of these machines has been difficult to realize. One of the primary problems is that the absolute performance of many early parallel machines was not significantly better than uniprocessors — both because of the tremendous rate of increase in the performance of uniprocessors and also because multiprocessor architectures are more complex and take longer to develop. Another problem is that programming a parallel machine is more difficult than a sequential one, and it takes much more effort to port an existing sequential program to a parallel architecture than to new a uniprocessor architecture.

Recently, there has been increased interest in large-scale or massive parallel processing systems incorporating hundreds or even thousands of processors. This interest stems from the very high aggregate performance of these machines and other developments that make such systems more tenable, such as the advent of high-performance microprocessors and the widespread use of small-scale multiprocessors.

Improvements in integrated circuit technology now allow microprocessors to approach the performance of the fastest supercomputers. This development has implications for both uniprocessor and multiprocessor systems. For uniprocessors, the pace of performance gains because of further integration is likely to slow. In the past, microprocessor designs had to scarify performance in order to fit on a single die. Today, virtually all the performance features found in the most complex processors can be implemented on a single chip. RISC technology has reduced the number of clocks per instruction to approximately one, and many studies indicate that more advanced superscalar designs may not offer more than a factor of two to four in improvement for general applications. On-chip multiprocessors with two to four CPUs per die, may be the most effective way to utilize the increased silicon area available in next-generation technology.

The widespread use of small-scale multiprocessors has also contributed to improved parallel processing technology. Multiprocessing has even come to the desktop in the form of multiprocessor workstations and high-end multiprocessor PCs. On the hardware side, almost all high-end microprocessors directly support small-scale multiprocessing. Likewise, modern bus standards include mechanisms for keeping the processor caches consistent with one another and with main memory. Because standard components can be used, both the cost and development time of parallel machines have been reduced.

On the software side, improvements have been made in the usability of parallel machines. Standard multiprocessor operating systems such as Mach, Solaris, IRIX, Linux, and Windows NT are widely available. These operating systems provide the basis for the management of multiprocessor resources and create standard parallel processing environments. Parallel languages and tools have also greatly improved. Parallelizing compilers and performance debugging tools that help automate porting sequential code to parallel machines have become commercially available. Likewise, parallel language extension, such as Linda, and the Parallel Virtual Machine (PVM) package, make it possible to write portable parallel programs.

Of course, many challenges must still be overcome to achieve the potential of large-scale multiprocessors. Two of the most difficult problems are scalability and ease of programming. *Scalability* refers to maintaining the cost-performance of a uniprocessor while linearly increasing overall performance as processors are added. Programming a parallel machine is inherently more difficult than a uniprocessor, where there is a single thread of control.

1.1 Project Goals

Topsy is a small multithreaded operating system which has been designed for teaching purpose at the Department of Electrical Engineering at ETH Zürich. Goal of this thesis was the design and implementation of an SMP port of Topsy. Furthermore, an SMP environment had to be developed or evaluated, respectively, in order to simulate the SMP hardware. Finally, the performance gain of an SMP kernel should be measured with suitable benchmarks.

The design of a multiprocessor OS is complicated because it must fulfill the following requirements: A multiprocessor OS must be able to support concurrent task execution, and it should be able to exploit the power of multiple processors. The principal design goals of the SMP port were defined as follows:

- **Simplicity.** The simple structure of the Topsy OS should not be complicated by an over-sized SMP mechanism.
- **High degree of parallel Kernel Activity.** The kernel should scale well running applications with a realistic job mix.
- **Uniprocessor API.** The system call API of Topsy should not be changed, allowing any Topsy application to be run without modification.
- **Scalability.** Scalability means, that additional CPUs can be added to (or removed from) the system without recompiling or even reconfiguring the kernel.

1.2 Development Environment

Topsy was developed initially to run on a IDT 7RS385 evaluation board with a 25 MHz MIPS R3000 processor and 1 MByte RAM. Besides, an R3000 simulator (the *MipsSimulator* written in Java) was developed to simplify the development process of Topsy [Fan]. The GNU tools gcc and binutils were used for code generation, both configured for cross-development with the MIPS platform as target.

As the *MipsSimulator* was not sufficient to develop and run a multiprocessor kernel, SimOS from Stanford University was chosen as a simulation environment (refer to Chapter 9). This environment consists of a simulator (written in C) running on a Sun Ultra-30 workstation. The GNU tools were used further on for code generation and debugging. However, the output format of the linker was changed from ecoff/srec to ELF32, because SimOS does not support the SREC format formerly used.

2 Review of Kernel Internals

This chapter provides a review of the relevant OS kernel internals that are referenced in later chapters. It is not a complete discussion of the topic but is meant instead as a summary of fundamental concepts and terminology. This chapter covers uniprocessor UNIX systems because most of todays operating systems are based somehow on the original UNIX design. Multiprocessor system implementation are the subject of later chapter in this thesis.

2.1 Introduction

The UNIX system is a multiuser, multitasking operating system that provides a high degree of program portability and a rich set of development tools. Part of the reason for the success of the UNIX system is the portable set of application interfaces that it provides. Another part of the success of UNIX is that the operating system, commands, and libraries are themselves written to be easily ported to different machines.

The UNIX system is logically layered and divided into two main portions: the kernel and user programs. This is shown in Figure 2.1.

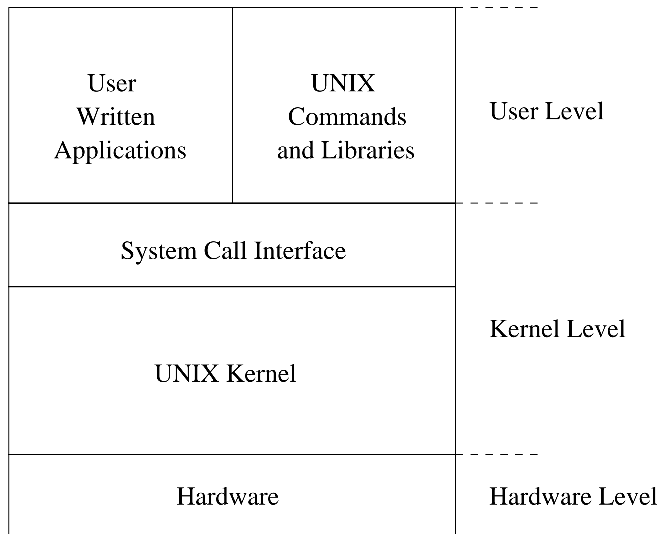


Figure 2.1: Logical layering of the UNIX system.

The purpose of the kernel is to interface with and control the hardware. The kernel also provides user programs with a set of abstract system services, called *system calls*, that are accessed using portable interfaces. The kernel runs at *kernel level*, where it can execute privileged operations. This allows the kernel to have full control over the hardware and user-level programs, as well as provide an environment where all programs share the underlying hardware in a coordinated fashion.

User applications, UNIX commands, and libraries coexist at *user level*. User level refers to the unprivileged execution state of hardware. User-level programs therefore execute in a restricted environment, controlled by the kernel, that prevents simultaneously executing programs from interfering with one another (either unintentionally or maliciously). When user programs request services by executing a system call, the system call will cause a transition into the kernel where it acts on behalf of the requesting user program to perform a service. Permission checks may be made to ensure that the program has permission to access the requested service.

Figure 2.1 depicts how the UNIX system, and most other operating systems, has traditionally been implemented: as a monolithic program. Over time, this implementation has been migrating toward a structured approach, where the kernel services are partitioned into independent modules. This increases the flexibility of the implementation and makes it easier to add, change, and port system services. It also makes it possible to migrate some services outside of the kernel and run them at user level in special server processes. This reduces the services the kernel itself must provide, allowing it to be reduces to a *microkernel*.

2.2 Processes, Programs, and Threads

A *program* is defined to be a set of instructions and data needed to perform some task. A *process* is a combination of a program plus the current state of its execution, which minimally includes the values of all variables, the hardware (e.g., the program counter, registers, condition code, etc.), and the contents of the address space. In short, a process is a program in execution.

When a user requests a program to be run, a new process is created to encompass its execution. The process exists within the system until it terminates, either by itself voluntarily, by the kernel, or by request of the user. The process itself is an abstraction the kernel provides to manage the program while it is executing.

Through the process abstraction, the kernel gives the program the illusion that it is executing on the hardware by itself. User programs need not concern themselves with interactions with other programs on the system unless they explicitly wish to communicate with them in some way. Each process is given its own virtual address space and is time-sliced (on most implementations) so that many processes may share the hardware. The existence of other processes on the system is transparent to the user program.

Many modern UNIX systems provide a mechanism known as *threads*. A thread holds the state of a single flow of execution within a process. The state of a thread consists at a minimum of the hardware state and a stack. All UNIX processes have at least one thread of control within them, which represents the execution of the program. This is true for all UNIX systems, past and present. Systems that support threads allow multiple threads of control within a process at the same time.

In this case, each has its own hardware state, but all execute within the same address space. On a uniprocessor, only one thread may execute at a time. On a multiprocessor, different threads within one process may execute simultaneously on different processors. One of the advantages of threads is that thread creation involves less overhead within one process than process creation, making it more efficient to implement a set of cooperating threads within one process than to implement a set of separate cooperating processes.

With few exceptions, all program execution, both user and kernel level, takes place within the context of some process. (This is true of most traditional UNIX kernel implementations.) All user programs execute within the context of their own processes. When these user processes request kernel services through system calls, the execution of the kernel code that implements the system call continues within the requestors process context. This conveniently gives the kernel access to all of the process's state and its address space. If execution of a system call needs to be suspended to await I/O completion, for example, the kernel's state regarding the processing of the system call is saved in the process data structure.

The UNIX kernel schedules only processes for execution since all system activity, whether at user or kernel level, occurs within the context of some process. When using the traditional time-sharing scheduling policies, processes executing at user level may be time-sliced at any time in order to share the CPU fairly among all processes. Process executing at kernel level are exempt from time-slicing. A switch to a different process while executing at kernel level is done only when the current kernel process explicitly allows it to occur.

One of the exceptions to the rule that all system activity occurs within processes is the execution of *interrupt handlers*. Interrupts are asynchronous to the execution of processes in that they can occur without warning at any time. When they occur, the UNIX kernel allows them to interrupt the execution of the current process. The system then executes the interrupt handler until it completes or until it is interrupted by a higher priority interrupt. Kernel-level processes have the privilege of blocking interrupts if they wish. This is done only to protect the integrity of data structures shared by the process level and interrupt handler code.

2.2.1 Context Switch

The act of the kernel changing from executing one process to another is called a *context switch*. This involves saving the state of the current process so that it may be resumed later, selecting a new process for execution, and loading the saved state of the new process into the hardware. The minimal state of the process that must be saved and restored at context switch time is the content of the CPU's registers, the PC, the stack pointer, the condition code, and the mapping of the virtual address space.

The acting of switching from one thread to another within the same process is called a *thread switch*. Since the process is not changed, there is no need to alter the address space mapping. Only the registers and other items listed above need to be saved and restored. The reduced overhead of a thread switch compared to a process context switch in another advantage of using threads.

2.3 Summary

This chapter has reviewed the basic internals of the UNIX kernel. The UNIX system is a multiuser, multitasking operation system that provides a high degree of program portability between UNIX implementations by presenting the process with machine-independent abstract services. The execution of programs is contained within processes that maintain the current state of the program, including the virtual address space, the values of its variables, and the hardware state. The kernel provides each process with an environment that makes it appear as though is were the only process executing on the system. This is done primarily by giving each process its own virtual address space. User programs request services of the kernel by executing system calls.

3 Multiprocessor Hardware Architectures

The architecture of a multiprocessor defines the relationship between the processors, memory, and I/O devices within the system. There are two major dimensions to multiprocessor architectures. The first dimension is based on whether all processors executes a single instruction stream in parallel (*single instruction multiple data — SIMD*), or whether each processor executes instructions independently (*multiple instructions multiple data — MIMD*). The second dimension is bases on the mechanism by which the processors communicate. Communication can either be through explicit messages sent directly from one processor to another, or through access to a shared-memory address space.

3.1 Single vs. Multiple Instruction Stream

As originally defined by Flynn [Fly66], SIMD refers to an architecture with a single instruction stream that is capable of operating on multiple data items concurrently. In an SIMD machine, all processors do the same work on separate data elements. The only variance between processor execution is based on local condition codes, which can temporarily enable or disable individual processing steps. Conversely, in an MIMD architecture, each processor executes its own instruction stream independently.

Simplified block diagrams of each of these system types are shown in Figures 3.1 and 3.2. The SIMD structure consists of multiple data processing elements that operate on data in the local registers and memory. Each data processing element receives instructions over a common instruction bus from a central control processor. The control processor is a complete computer that runs the main thread of the program. The MIMD structure consists of a set of independent processors with local memory, in which each processor executes its own instruction stream. Like the SIMD machine, the processor-memory pairs are connected by an interconnection network. While each MIMD processor may execute the same program, each processor executes only the instructions relevant to its data items and can make progress independently.

An SIMD machine has two primary advantages over an MIMD machine. First, each SIMD data processing element is simpler than its MIMD counterpart, since it has only a data path and shares its instruction sequencer with the other processing elements. Many of the early large-scale parallel machines were SIMD machines because of this reduced hardware requirement. For a given fixed amount of hardware, an SIMD machine can have more processors, and thus a higher peak performance rate, than an MIMD machine. The second advantage of an SIMD architecture is that there is no performance penalty for synchronizing processors since all processors operate in lockstep, which reduces the overhead of coordinating the processors.

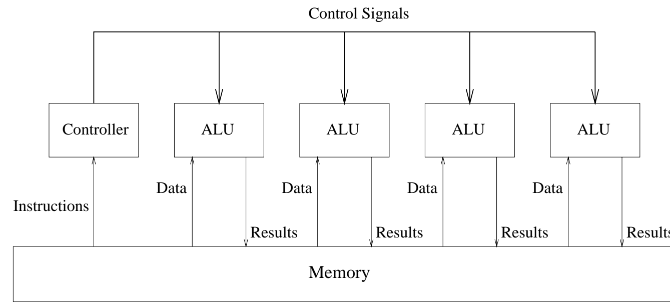


Figure 3.1: Single Instruction Multiple Data (SIMD)

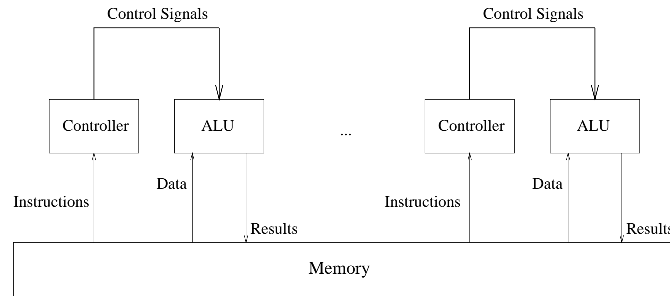


Figure 3.2: Multiple Instruction Multiple Data (MIMD)

Conversely, there are significant advantages of MIMD over SIMD. First, since each processor executes its own instruction stream, only instructions useful to each data item need to be executed. Thus, while an SIMD machine may have more processors, the utilization of these processors is lower than in an MIMD machine. Independent processors also make MIMD machines more flexible to program that their SIMD counterparts. Another important practical advantage of MIMD machines is that they can utilize commodity microprocessors while SIMD machines are based on custom parts whose performance lags behind that of the latest microprocessors. Overall, the trend has been away from SIMD and toward MIMD architectures.

3.2 Message-Passing vs. Shared-Memory Architectures

Message-passing and shared memory architectures are distinguished by the way processors communicate with one another and with memory. In message-passing systems, also called *distributed-memory systems*, the programmer sees a collection of separate computers that communicate only by sending explicit messages to one another. In a shared-memory system, the processors are more tightly coupled. Memory is accessible to all processors, and communication is through shared variables or messages deposited in shared memory buffers. This logical difference is illustrated in Figure 3.3 and has profound effects on the ease of programming these systems. In a shared-memory machine, processes can distinguish communication destination, type, and values through shared-memory addresses. There is no requirement for the programmer to manage the movement of data. In contrast, on a message-passing machine, the user must explicitly communicate all information

passed between processes. Unless communication pattern are very regular, the management of this data movement is difficult.

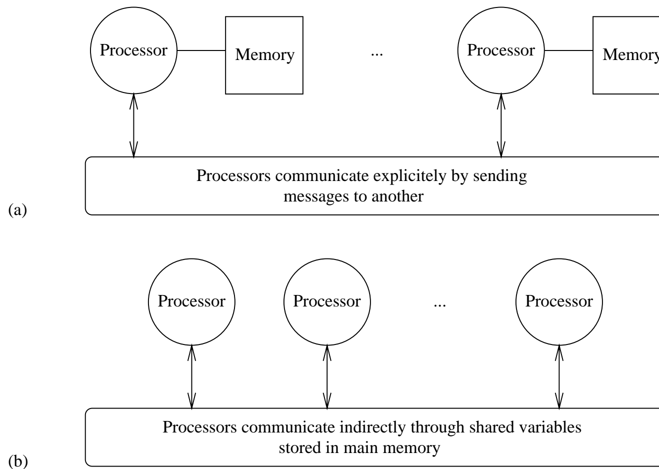


Figure 3.3: Programmer's View of (a) Message-Passing and (b) Shared-Memory Architectures

Interprocessor communication is also critical in comparing the performance of message-passing and shared-memory machines. Communication in a message-passing system benefits from its directness and its initiation by the producer of the data. In shared-memory systems, communication is indirect, and the producer typically moves the data no further than memory. The consumer must then fetch the data from memory, which can decrease the efficiency of the receiving processor. However, in a shared-memory system, communication requires no intervention on the part of a run-time library or operating system. In a message-passing system, access to the network port is typically managed by system software. This overhead at the sending and receiving processors make the start-up cost of communication much higher on a message-passing machine.

It is also possible to build a shared-memory machine with a distributed-memory architecture. Such a machine has the same structure as the message-passing system shown in Figure 3.4(a), but instead of sending messages to other processors, every processor can directly address both its local memory and remote memories of every other processor. This architecture is referred to as *distributed shared-memory (DSM)* or *nonuniform memory access (NUMA)* architecture. The latter term is in contrast to the *uniform memory access (UMA)* structure shown in Figure 3.4(b). The UMA architecture is easier to program because there is no need to worry about where to allocate memory — all memory is equally close to all processors. Since most shared-memory machines only support a small number of processors, where there is minimal benefit from the local memory of the NUMA structure, UMA is more popular for those machines.

There are many trade-offs between message-passing and the two form of shared-memory architectures. One of the biggest advantages of message-passing systems is that they minimize the hardware overhead relative to a collection of uniprocessor systems. Even a group of workstations can be treated as a single message-passing system with appropriate cluster software. The primary advantage of a shared-memory over a message-passing system is its simple programming model, which is an extension of the uniprocessor model. In this model, the data is directly accessible to every processor, and explicit communication code in only needed to coordinate access to shared variables.

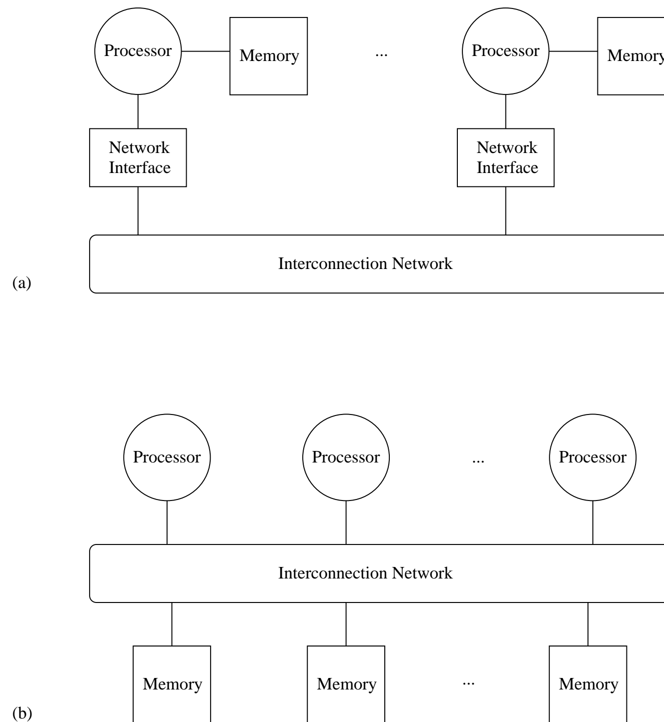


Figure 3.4: (a) Distributed-Memory and (b) Shared-Memory Architectures

In a message-passing system, the user is responsible for explicitly partitioning all shared data and managing communication of any updates to that data.

Overall, the shared-memory paradigm is preferred since it is simpler to use and more flexible. In addition, a shared-memory system can efficiently emulate a message-passing system, while the converse is not possible without a significant performance degradation. Unfortunately, a shared-memory system has two major disadvantages relative to a message-passing system. First, processor performance can suffer if communication is frequent and not overlapped with computation. Second, the interconnection network between the processors and memory usually requires higher bandwidth and more sophistication that the network in a message-passing system, which can increase overhead costs to the point that the system does not scale well.

Solving the latency problem through memory caching and hardware cache coherence is the key to a shared-memory multiprocessor that provides both high processor utilization and high communication performance. While the coherence mechanism adds cost to the shared-memory machine, careful design can limit overhead. Furthermore, the growing complexity of microprocessors allows for a more sophisticated memory system without drastically altering the cost-performance of the overall system.

3.3 Cache Coherence

Caching of memory has become a key element in achieving high performance. In the last 10 years, microprocessors cycle times have shrunk from 100–200 ns to less than 4 ns, a factor of more than 25 times. In the same time frame, DRAM access times have only improved from 150–180 ns to 60–80 ns, a factor of about three. This gap has been bridged only with a memory hierarchy consisting of varying levels of high-speed cache memories that reduce average memory latency and provide the additional bandwidth required by today's processors.

3.3.1 Uniprocessor Caches

Caches consist of high-speed memory components that hold portions of the memory space addressable by the processor. Access to cache memory is much faster than main memory because the cache consists of a small number of very fast memory parts located close to the processor (often on the processor chip). In a cache-based system, the average memory access time (T_{avg}) is given by

$$T_{avg} = T_{cache} + F_{miss} T_{mem}.$$

F_{miss} is the fraction of references that are not found in the cache and T_{cache} and T_{mem} are the access times to the cache and memory, respectively. Cache access time is typically one or two processor cycles, while main memory access times range from 30 to 60, or more, processor cycles. Thus, if a high percentage of accesses are satisfied by the cache, average memory latency can be drastically reduces.

Caching memory is effective because programs exhibit temporal and spatial locality in their memory access pattern. *Temporal locality* is the propensity of a program to access a location that it has just accessed. Access to loop indices and the stack are examples of data that exhibit a high degree of temporal locality. A cache can exploit temporal locality by allocating locations in the cache whenever an access misses the cache. *Spatial locality* is the tendency of a program to access variables at locations near those that have just been accessed. Examples of spatial locality include accessing a one-dimensional array in different loop iterations and the sequential fetching of instructions. A cache captures spatial locality by fetching locations surrounding the one that actually caused the cache miss.

3.3.2 Multiprocessor Caches

Just as in uniprocessor systems, caching is also vital to reducing memory latency in multiprocessor systems. In fact, caching offers even more benefit in a multiprocessor because memory latencies are usually higher. Unfortunately, multiprocessor caches are more complex because they introduce coherence problems between independent processor caches.

Caches introduce two major coherence problems. First, when multiple processors read the same location they create *shared* copies of memory in their respective caches. By itself, this is not a problem. However, if the location is written, some action must be taken to insure that the other processor caches do not supply stale data. These cached copies can either be updated or simply

eliminated through *invalidation* (Figure 3.5). After completing its write, the writing processor has an exclusive copy of the cache line (i.e., it holds the line *dirty* in its cache). This allows it to subsequently write the location by *updating* its cached copy only; further accesses to main memory or the other processor caches are not needed. After a write invalidation, other processors that reread the location get an updated copy from memory or the writing processor's cache.

CPU activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of memory location X
A reads X	Cache miss for X	0		0
B reads X	Cache miss for X	0	0	0
A writes 1 to X	Invalidation for X	1		0
B reads X	Cache miss for X	1	1	1

Figure 3.5: Example of an invalidation protocol

The second coherence problem arises when a processor holds an item dirty in its cache. When lines are dirty, simply reading a location may return a stale value from memory. To eliminate this problem, reads also require interaction with the other processor caches. If a cache holds the requested line dirty, it must override memory's response with its copy of the cache line. If memory is updated when the dirty copy is provided, then both the dirty and requesting cache enter a shared state. This state is equivalent to the initial state after multiple processors have read the location. Thus three cache states — invalid (after a write by another processor), shared (after a read), and dirty (after a write by this processor) — provide the basis for a simple multiprocessor coherence scheme.

Most small-scale multiprocessors maintain cache coherence with *snoopy caches*. The structure of a typical snoopy cache-coherence system is shown in Figure 3.6. Snoopy coherence protocols rely on every processor monitoring all requests to memory. The monitoring, or snooping, allows each cache to independently determine whether access made by another processor require it to update its cache state. Because snooping relies on broadcasting every memory reference to each processor cache, snoopy systems are usually built around a central bus.

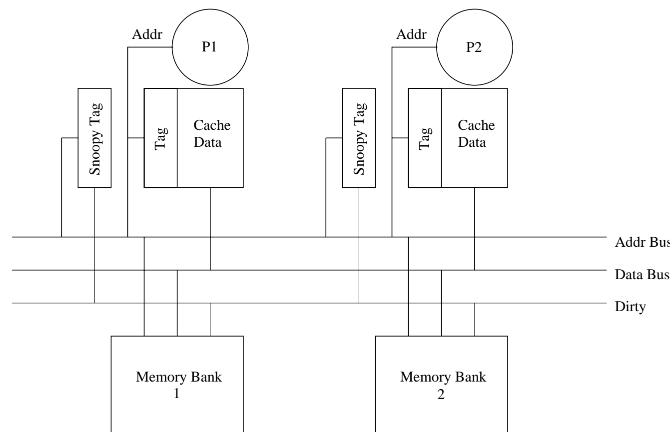


Figure 3.6: Bus-Based MP with a simple Snoopy Cache Structure

3.4 Scalability

Having established the basis of shared-memory multiprocessing and cache coherence, we now turn to the issue of extending the SMP model to a larger number of processors. Intuitively, a *scalable system* is a system whose performance varies linearly with the cost of the system. What is more meaningful is the degree to which a system scales. Scalability can be measured on three dimensions:

1. How does the performance vary with added processors? That is, what is the speedup ($S(N)$) over a uniprocessor when N processors are used to solve a problem in parallel?
2. How does the cost of the system vary with the number of processors in the system?
3. What range of processors (N) does the system support, or provide useful speedup over?

The most complex issue with scaling is how performance scales as processors are added. If speedup is defined as the ratio of the execution time of a program on one processor versus N processors, then speedup can be expressed as

$$S(N) = \frac{T_{exec}(1)}{T_{exec}(N)}$$

Ideally, speedup would be linear with N such that $S(N) = N$. If $T_{exec}(N)$ is broken down into the time that the application spends computing and the time spent communicating, then $S(N)$ is given as

$$S(N) = \frac{T_{comp}(1) + T_{comm}(1)}{T_{comp}(N) + T_{comm}(N)}$$

where $T_{comm}(1)$ represents the time that the UP accessed global memory, while $T_{comm}(N)$ represents the time for N processors to access global memory, synchronize, and communicate with one another through global memory. If one assumes that the total amount of communication is independent of N, then perfect speedup will only be achieved if

$$\begin{aligned} T_{comp}(N) &= T_{comp}(1)/N \\ T_{comm}(N) &= T_{comm}(1)/N \end{aligned}$$

From a hardware architecture perspective, the requirement of scaling T_{comp} is easily met with N processors working independently. Unfortunately, this requires the software algorithm to be perfectly split between the N processors.

Overall, we see that no system can be ideally scalable since the cost of communication can not be ideally scaled because of increases in the latency of individual communication. Thus, linear performance growth cannot be maintained for large systems. Furthermore, communication bandwidth cannot grow linearly with system size unless costs grow more than linearly. Although these architecture limits are important to understand, practical considerations make the goal of ideal scalability itself dubious. In particular, ideal scalability requires software algorithms that are perfectly parallel and have a constant amount of communication independent of the number of processors. All algorithms break these rules to some degree, and thus limit useful system sizes.

3.4.1 Scalable Interconnection Networks

As shown in the previous section, the interconnection network is a key element of system scalability. To summarize, an ideal scalable network should have

1. a low cost that grows linearly with the number of processors N,
2. minimal latency independent of N, and
3. bandwidth that grows linearly with N.

The most widespread interconnection network used in multiprocessors is a simple bus in which the processors and memory are connected by a common set of wires. The advantages of this structure are that it has constant latency between all nodes and a linear cost with the number of processors. The big disadvantage, however, is that the amount of bandwidth is fixed and does not grow with the number of processors, which puts an obvious limit on the scalability of bus-based systems. It also forces a difficult trade-off to be made as to the width and speed of the bus. Increasing both of these permits a larger maximum system size. Unfortunately, increasing width increases costs, especially due to degradation in the electrical performance as system size increases. Some bus-based systems get around the fixed bandwidth problem by supporting multiple buses and interleaving them on low-order address bits. For example, the Sun SparcCenter 1000 uses one 64-bit bus to support 8 processors, the SparcCenter 2000 replicates this bus to support up to 20 processors, and the Cray SuperServer 6400 utilizes four copies of the same bus to support up to 64 processors. Electrical constraints on the bus put a limit on how well this technique can be scaled, and also imply increased cost with system size.

3.4.2 Scalability of Parallel Software

Consider the simple problem of forming the sum of M numbers where M is much larger that the number of processors N. In this case, N partial sums can be formed in parallel by breaking the list into N lists, each with M/N numbers in it. Combining the N partial sums, however, can not utilize all of the processors. At best, the sum can be formed in $\log N$ steps using $N/(2i)$ processors at the ith step. For 100 processors to sum 10 000 numbers, it would take the order of 100 time steps to from the partial sums, and then at least another 7 steps to add the partial sums. Thus, for this size input the speedup is $10000/107$ or 93.4 times. The amount of communication also grows with the number of processors since the number of partial sums is equal to the number of processors. Further, when adding the partial sums, communication delays dominate. In many cases, using a single processor to add the partial sums may be just as fast as using a tree of processors. If a single processor is used in the final phase, then speedup is limited to $10000/200 = 50$.

The execution pattern of the above example is typical of many parallel programs. Parallel programs typically exhibit alternating phases of limited and full parallelism: limited parallelism while the computation is initialized or the results are summarized, and full parallelism while the computation is done across the data by all the processors. If we approximate the phases of execution as a strictly

serial portion followed by a full parallel section, then the time to do a computation on N processors can be written in terms of the percentage of serial code (s) as

$$T_{comp}(N) = T_{comp}(1)(s + (1-s)/N)$$

Likewise, if we ignore the $\log N$ factor in parallel communication time from increased hardware latency, and assume that the total amount of communication does not increase, then communication, $T_{comm}(N)$, is given by

$$T_{comm}(N) = T_{comm}(1)(s + (1-s)/N)$$

resulting in an overall speedup equation of

$$S(N) = \frac{T_{comp}(1) + T_{comm}(1)}{(T_{comp}(1) + T_{comm}(1))(s + (1-s)/N} = \frac{1}{s + (1-s)/N}$$

This formula was originally derived by Gene Amdahl in 1967 and is known as *Amdahl's Law*. An important implication of Amdahl's Law is that the maximum speedup of an application (on an infinite number of processors) is given by

$$S(N) \leq \frac{1}{s}$$

For the simple summation example, the final partial sum takes either 7 or 100 cycles compared with the approximately 10 000 cycles that can be parallelized to form the partial sums. Thus, $s = 7/10000 = 0.07\%$ for the parallel partial sum addition and $s = 100/10000 = 1\%$ for the serial addition, and the resulting maximum speedup given by Amdahl's Law is 1.429 and 100 respectively.

3.5 Performance Considerations

Ideally, the overall system throughput of an SMP system will increase linearly as more processors are added and will equal the product of the number of processors and the throughput of a single processor. Thus a two-processor system should be able to handle twice the throughput of an UP. How close an MP implementation can approach this ideal depends on three main factors: the hardware architecture, the application job mix, and the kernel implementation.

If the hardware design is not suitable for an SMP system, then no amount of software tuning will allow an implementation to approach the ideal goal of linear performance increase as additional processors are added. For example, if the memory subsystem does not provide sufficient bandwidth for all processors, then the extra processor will not be fully utilized.

The application job mix refers to the number and type of applications that are run on the system. By using the same application mix on different kernel implementation, a benchmark can be formed by which system performance can be measured and compared with other kernel implementations. Is is important to understand the application job mix of any benchmark in order to interpret the results correctly.

3.6 Summary

Microprocessor-based multiprocessors are becoming an increasingly viable way to achieve high performance because of the narrowing gap between microprocessors and the fastest supercomputers, and the tremendous cost-performance advantage of microprocessor-based systems. Programming a parallel machine remains a challenge, however. Of the alternative parallel architectures, a multiple instruction, multiple data (MIMD) architecture supporting global shared-memory is preferred because of its flexibility and ease for programming.

The biggest problems with shared-memory multiprocessors are keeping their performance per processor high, allowing scalability to large processor counts, and dealing with their hardware complexity.

4 Multiprocessor Kernel Architectures

This chapter introduces the tightly coupled, shared memory multiprocessor. It is the type of multiprocessor most commonly used with UNIX systems, since it parallels the execution environment assumed by standard uniprocessor UNIX kernel implementations. The following sections describe its organization in preparation for subsequent chapters that examine how the UNIX operating system can be adapted to run on this type of hardware. All multiprocessor systems presented in this chapter operate without caches, to illustrate better the fundamental issues that multiprocessor operating systems must solve.

4.1 MP Operating System

Recall from Chapter 1 that a multiprocessor consists of two or more CPUs combined to form a single computer system. The operating system for a multiprocessor must be designed to coordinate simultaneous activity by all CPUs. This is a more complex task than managing a uniprocessor system.

All MP kernel implementations must maintain *system integrity*. This means that the CPUs' parallel activity is properly coordinated so that the kernel's data structures are not compromised. This ensures the correct functioning of the system under all possible situations, regardless of the timing of external events and activities by the CPUs in the system.

Once integrity is achieved, the implementation can be modified and tuned to maximize *performance*. The third factor, the *external programming model*, determines how the presence of multiple CPUs affects the system call interface. The operating system designer for an MP system has to choose whether or not to "disguise" the MP system to appear as UP system. If the system call interface is compatible with that of an UP system, then existing uniprocessor application programs can be run on the MP without change. If, on the other hand, the system call interface is not compatible, the programs will have to be written with explicit knowledge of the multiple CPUs in the system and may be required to use special system calls to communicate with, or pass data to, processes running on other processors.

Because of the high cost of rewriting programs to conform to a new system call interface, most implementations choose to maintain the uniprocessor system call interface so the presence of the multiple CPUs is entirely transparent. This is not to say that the operating system is forbidden from offering new interfaces that allow programmers to make use of the inherent parallelism in an MP; it means that the kernel must provide all the uniprocessor system call interfaces and facilities.

4.2 The Tightly Coupled, Shared Memory, Symmetric Multiprocessor

The multiprocessor architecture of interest for the remainder of the thesis is the *tightly coupled, shared memory, symmetric multiprocessor*, frequently abbreviated as *SMP*. This is the most commonly used type of MP since it readily supports the implementation of an operating system that retains the uniprocessor external programming model. A high-level view of such a system with four CPUs appear in Figure 4.1.

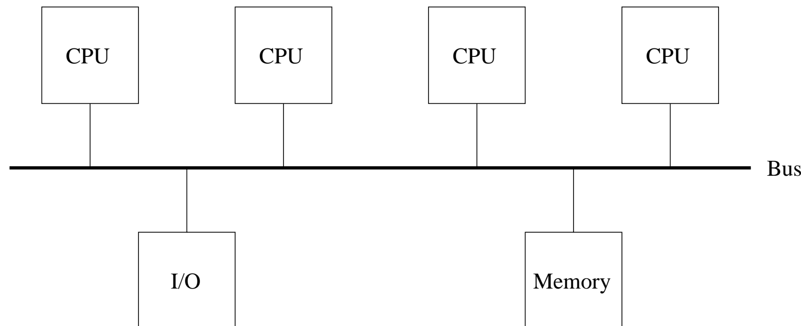


Figure 4.1: Example SMP block diagram.

There are several important factors to understand about this type of multiprocessor architecture. First, all CPUs, memory and I/O are *tightly coupled*. There are several ways to achieve this, but the simplest and most commonly used method is for all units to be directly interconnected by a common high-speed bus. Another meaning of tightly coupled refers to the fact that all components are located within a short distance of one another (usually within the same physical cabinet).

The *shared memory* aspect is easy to see in Figure 4.1: there is a single, globally accessible memory module used by all CPUs and I/O devices. The CPUs themselves have no local memory (except possibly for caches) and store all their program instructions and data in the global shared memory. The important factor here is that data stored by one CPU into memory is immediately accessible to all other CPUs.

The final aspect of importance in the SMP architecture is that memory access is *symmetric*. This means that all CPUs and I/O devices have equal access to the global shared memory. They use the same physical address whenever referring to the same piece of data. Access to the bus and memory is arbitrated, so that all CPUs and devices are guaranteed fair access. In addition, their accesses do not interfere with one another.

The maximum number of CPUs that can be used for practical work in an SMP is limited by the bandwidth of the shared bus and main memory. The bandwidth must be sufficient to supply the needs of all CPUs and I/O devices in the system, or the overall performance of the system will suffer.

4.3 The MP Memory Model

The *memory model* for an MP system defines how the CPUs, and hence the programs running on them, access main memory, and how they are affected by simultaneous accesses by other CPUs. Since main memory is accessed with physical addresses, the effects of virtual address translation are not considered as part of the memory model. The memory model is instead concerned with the transfer of physical addresses and data between the CPUs and main memory.

The memory model for an SMP includes those characteristics mentioned in the previous section: memory is globally accessible, tightly coupled, and symmetric. The memory model then goes on to define load-store ordering and the effects of simultaneous access to memory by multiple CPUs.

4.3.1 The Sequential Memory Model

The simplest and most commonly used memory model is the *sequential memory model* (also known as *strong ordering*). In this model, all load and store instructions are executed in *program order*, the order in which they appear in the program's sequential instruction stream, by each CPU. This model is used by the MIPS RISC processor line, the Motorola MC68000 and MC88000 processor lines, as well as the Intel 80x86 family.

4.3.2 Atomic Reads and Writes

The sequential memory model defines an individual read or write operation from a CPU or I/O device (through DMA) to main memory to be *atomic*. Once begun, such an operation cannot be interrupted or interfered with by any other memory operation, from any CPU or I/O device, on the system. The atomicity of memory operations is easily guaranteed by virtue of the single shared bus that must be used to access main memory. The bus can be used by only one CPU (or I/O device) at a time. If multiple CPUs wish to access memory simultaneously, special hardware in the bus *arbitrates* between the multiple requestors to determine which will be allowed to use the bus next. When one is chosen, the bus is *granted* to it, and that CPU is allowed to complete a single read or write operation atomically involving one or more physically contiguous words in main memory. During this single operation, all other CPUs and I/O devices are inhibited from performing any of their own read or write operations. After the completion of each operation, the cycle repeats: the bus is rearbitrated and granted to a different CPU. The choice of which CPU gets the next turn on the bus may be done using *first-in-first-out*, *round-robin*, or whatever other type of scheduling is implemented in the bus hardware. I/O devices arbitrate for the bus in the same manner as CPUs.

The amount of data that can be transferred in a single operation is limited to prevent one processor from hogging the bus. While the typical transfer size in actual machines is usually equal to the cache line or subline size, the transfer size is not directly visible to the operating system.

4.3.3 Atomic Read-Modify-Write Operations

Since the need to synchronize access to shared memory locations is so common on SMP systems, most implementations provide the basic hardware support for this through atomic *read-modify-write*

operations. Such operations allow a CPU to read a value from main memory, modify it, and store the modified value back into the same memory location as a single atomic bus operation.

The type of modification that may be done to the data during a read-modify-write operation is implementation specific, but the most common is the *test-and-set* instruction. The Motorola MC68040 and the IBM 370 architecture are examples of processors that use this operation. This instruction reads a value from main memory (usually a byte or word), compares it to 0 (setting the condition code in the processor accordingly), and unconditionally stores a 1 into the memory location, all as a single atomic operation. It is not possible for any other CPU or I/O device to access main memory once a test-and-set instruction begins its bus cycle. With this one basic operation, the operation system can build higher level synchronization operations.

While more complex operations, such as an atomic increment or decrement operation, are possible, modern RISC systems tend to offer simpler operations. For example, probably the most basic single atomic read-modify-write instruction is the *swap-atomic* operation. This type of operation is used in the Sun SPARC processor and in the Motorola MC88100 RISC processor. Such an instruction merely swaps a value stored in a register with a value in memory. One can then construct a test-and-set operation by setting the value in the register to 1, performing the atomic swap, and then comparing the value in the register (the old contents of the memory location) to 0 afterwards. Figure 4.2 illustrates this in C.

```
int test_and_set(int *addr) {
  int old_value = swap_atomic(addr, 1);
  if (old_value == 0) {
     return 0;
  }
  return 1;
}
```

Figure 4.2: Test-and-set implementation using swap-atomic.

Some RISC architectures simplify this a step further and provide a pair of instructions that together can be used to perform an atomic read-modify-write operation. The pair is the *load-linked* and *store-conditional* instructions. This is the approach taken by the MIPS R4000 RISC processor. (Refer to Section 9.2.3 for an implementation of an atomic read-modify-write based on the MIPS R4000 instructions.) Synchronization is still achievable without any atomic read-modify-write operations at all using a software technique known as *Dekker's Algorithm*.

4.4 Mutual Exclusion

Since the sequential memory model does not guarantee a deterministic ordering of simultaneous reads and writes to the same memory location from more than one CPU, any shared data structure cannot be simultaneously updated by more than one CPU without the risk of corrupting the data. The lack of deterministic ordering causes a *race condition* to occur. This happens whenever the outcome of a set of operations in an SMP depends on the relative ordering or timing of the operations between two or more processors. Such nondeterministic behavior can be catastrophic to the integrity of the kernel's data structures and must be prevented.

Any sequence of instructions that update variables or data structures shared between two or more processors can lead to a race condition. The sequence of instructions themselves it referred to as *critical section*, and the data they operate on is the *critical resource*. In order to eliminate the race condition caused by multiple processors simultaneously executing the critical section, at most one processor may be executing within the critical section at one time. This is referred to as *mutual exclusion* and can be implemented in a variety of ways.

Before going to show how mutual exclusion can be implemented in MP systems, it is useful to review how it was achieved in uniprocessor UNIX systems and why these techniques fail on MPs.

4.5 Review of Mutual Exclusion on Uniprocessor UNIX Systems

It is possible to have race conditions even in uniprocessor operating systems. Any system that permits multiple threads of control, such as multiple processes, needs to consider mutual exclusion between the threads. It is also possible for the instructions executed by interrupt handlers to race with the code they interrupt.

4.5.1 Short-Term Mutual Exclusion

Short-term mutual exclusion refers to preventing race conditions in short critical sections. These critical sections occur when the kernel is in the midst of updating one of its data structures. Since the kernel data structures are shared among all executing processes, a race condition would be possible if two or more processes executing in kernel mode were to update the same structure at the same time. Since a uniprocessor can only execute one process at a time, these race conditions are only possible if one process executing in the kernel can be preempted by another. This is why the designers of the UNIX kernel chose to make the kernel nonpreemptable while executing in kernel mode. Recall that processes executing in kernel mode are not time-sliced and cannot be preempted. A context switch to another process occurs only when a kernel mode process allows it.

The nonpreemptability rule for processes executing in kernel mode greatly reduces the complexity of uniprocessor UNIX kernel implementations. Since only one process is allowed to execute in the kernel at a time and is never preempted, race conditions while examining and updating kernel data structure cannot occur. Therefore, nothing more need be done to maintain data structure integrity.

4.5.2 Mutual Exclusion with Interrupt Handlers

If the code executed by an interrupt handler accesses or updates the same data structures used by non-interrupt code, then a race condition can occur. Fortunately, processes executing in kernel mode are permitted to disable interrupts temporarily. Therefore, whenever a process is about to update a data structure that is shared with an interrupt handler, it first disables interrupts, executes the critical section, and then reenables interrupts. The act of disabling and reenabling interrupts implements mutual exclusion.

It is important to understand how this implementation of mutual exclusion differs from that of short-term mutual exclusion. With short-term mutual exclusion, implementing the general policy of non-preemptability of kernel mode processes solved the problem without having to code it into the kernel explicitly. With interrupts, mutual exclusion had to be explicitly coded into the algorithms.

4.5.3 Long-Term Mutual Exclusion

Most of the UNIX system calls provide services that are guaranteed to be atomic operations from the viewpoint of the user program. For example, once a `write` system call to a regular file begins, it is guaranteed by the operating system that any other `read` or `write` system calls to the same file will be held until the current one completes. Uniprocessor UNIX kernels implement this type of mutual exclusion with the `sleep` and `wakeup` functions.

The `sleep` function is an internal kernel routine that causes the process calling it to be suspended until a specified event occurs. This is the primary means by which a process executing in kernel mode voluntarily relinquishes control and allows itself to be preempted. The `wakeup` function is used to signal that a particular event has occurred, and it causes all processes waiting for that event to be awakened and placed back on the run queue. The event is represented by an arbitrary integer value, which is usually the address of a kernel data structure associated with the event.

Each object within the kernel that requires long-term mutual exclusion is represented by an instance of a data structure. To implement atomic operations on the object, the object is "locked" so that only one process can access it at a time. This is done by adding a flag to the data structure that is set if the object is presently locked. For simplicity, assume the flag is stored in a byte in the object's data structure, so that each has a unique flag. One possible way to implement mutual exclusion on an arbitrary object is shown in Figure 4.3 (the actual details vary in different versions of the UNIX system and are not relevant to the discussion here).

```
void lock_object(char *flag_ptr) {
  while (*flag_ptr) {
    sleep(flag_ptr);
  }
  *flag_ptr = 1;
}
```

Figure 4.3: Code to lock an object.

In this example, the flag is set to 1 to indicate that a process currently has the object locked. At a beginning of an atomic operation, the `lock_object` function is called to lock the object by passing a pointer to the flag byte associated with it. If the object is not presently locked, the condition in the `while` statement will fail and the process will lock the object by setting the flag. Any other process that attempts to access the same object will use the same data structure and attempt to lock it by calling the `lock_object` function with the address of the same flag byte. This time, however, the condition in the `while` statement will be true, and the process will execute the `sleep` call, which will suspend the process.

It is important to understand that the operation of testing the flag, finding it clear, and setting it form a critical section and must be done with mutual exclusion. Otherwise a race condition would be

possible, resulting in cases where two or more processes would think they had the object locked. Fortunately, this race condition is prevented by the uniprocessor nonpreemptability policy.

When the process holding a lock has completed its atomic operation on the object, it would call the function shown in Figure 4.4.

```
void unlock_object(char *flag_ptr) {
  *flag_ptr = 0;
  wakeup(flag_ptr);
}
```

Figure 4.4: Code to unlock an object.

Here the flag is cleared and all processes that were waiting for the lock are awakened using the `wakeup` function. An important aspect of the `wakeup` function is that there is no "memory" of the event saved. This allows the `unlock_object` function to call it without having to know whether or not any processes are actually sleeping on the event.

4.6 Problems Using UP Mutual Exclusion Policies on MPs

To achieve a high-performance MP system, it is desirable to allow system calls and other kernel activity to occur in parallel on any processor. This way, the kernel's workload can be distributed throughout the system. Unfortunately, the techniques presented in the previous section, which enable a uniprocessor kernel implementation to avoid race conditions, fail to work properly when more that one processor on an MP system can execute kernel code at the same time.

The primary difficulty that prevents a uniprocessor kernel from running properly on an MP system is that multiple processors executing in the kernel simultaneously violates the assumptions that support short-term mutual exclusion. Once more than one processor begins executing in the kernel, the kernel data structures can be corrupted unless additional steps are taken to prevent races.

Mutual exclusion with interrupt handlers may not function properly on an MP system either. Raising the interrupt level only affects the processor priority on the current processor and do not affect interrupts delivered to other processors. Depending on the design of the hardware, interrupts may be delivered to any CPU in the system, or they may always be directed to one CPU. In either case, a process executing on one processor that is raising the interrupt level to implement mutual exclusion on data structures shared with interrupt handlers will not be properly protecting the data if the interrupt handler begins executing on a different processor.

Finally, the coding technique used to implement long-term mutual exclusion with the `sleep` and `wakeup` functions will not work correctly on MP systems. Recall from Section 4.5.3 that the implementation of `lock_object` relies on short-term mutual exclusion to prevent race conditions between the time the flag is tested and the process goes to sleep or sets the flag itself. Since the short-term mutual exclusion policy is no longer effective on MPs, these code sequences now contain races.

4.7 Summary

SMP is the most commonly used type of multiprocessor system since it parallels the execution environment of uniprocessor systems. All CPUs and I/O devices are tightly coupled, share a common global main memory, and have symmetric and equal access to memory. Most MP kernel implementations preserve the external uniprocessor programming model so that application programs do not have to be modified to run on an MP.

The memory model for an MP describes the ordering of load-store instructions within a program and how simultaneous access to main memory by multiple CPUs results. The *sequential memory model* provides for atomic read and write operations that are executed according to program order on each CPU, but it does not specify the relative ordering of simultaneous operations to the same memory location from different CPUs. Because of this, the sequential memory model usually provides some type of atomic read-modify-write operation that CPUs can use for synchronization purposes.

Any multiple instruction operation to a shared memory location or data structure is subject to race conditions since multiple processors could be attempting the operation at the same time. To prevent races, the kernel must implement mutual exclusion to sequentialize access to shared data and prevent it from being corrupted.

To simplify their design, uniprocessor UNIX kernel systems have relied on the fact that processes executing in kernel mode are nonpreemptable. This eliminates most race conditions since no other process can access any kernel data until the currently executing process voluntarily relinquishes the CPU. *Long-term mutual exclusion*, needed to support atomic file operations, for instance, is done using explicit locks and calls to the `sleep` and `wakeup` functions. Mutual exclusion with interrupt handling code is implemented by explicitly blocking and unblocking interrupts for the duration of the critical regions in the kernel code. However, these policies fail to provide mutual exclusion on MP systems when kernel code can be executed simultaneously on more than one processor.

5 Multiprocessor Kernel Implementations

This chapter presents three techniques for modifying a uniprocessor kernel implementation to run on an SMP system without race conditions: the *master-slave* kernel, the *spin-locked* kernel, and the *semaphored* kernel.

5.1 Master-Slave Kernels

The short-term mutual exclusion implementation technique presented in Section 4.5.1 relies on the fact that there is never more than one process executing in the kernel at the same time. A simple technique for doing this on an MP system is to require that all kernel execution occur on one physical processor, referred as the *master*. All other processors in the system, called *slaves*, may execute user code only. A process executing in user mode may execute on any processor in the system. However, when the process executes a system call, it is switched to the master processor. Any traps generated by a user-mode process running on a slave also cause a context switch to the master, so that the mutual exclusion requirements of the kernel trap handlers are also maintained. Finally, all device drivers and device interrupt handlers run only on the master processor.

The master-slave arrangement preserves the uniprocessor execution environment to the kernel's point of view. This allows a uniprocessor kernel implementation to run on an MP system with a few modifications and works for any number of processors. One of the main areas of modifications is in how processes are assigned to individual processors. A simple technique for this is to have two separate run queues, one containing kernel-mode processes that must be run on the master, and one containing user-mode processes for the slaves. At each context switch, each slave selects the highest priority process on the slave run queue, while the master processor selects the highest priority process on the kernel process queue. A process running on a slave that executes a system call or generates a trap is placed on the run queue for the master processor. When the master processor performs a context switch, the old process it was executing is placed on the slave queue if it was executing in user mode; otherwise, it goes back on the master queue.

Since there could be multiple processors enqueuing, dequeuing, and searching the run queues at the same time, a way to prevent race conditions is needed. The run queues are the only data structure that require an explicit MP short-term mutual exclusion technique, since all other data structures are protected by running all kernel code on the master. The easiest way to provide such short-term mutual exclusion is with spin locks.

5.1.1 Spin Locks

A *spin lock* is an MP short-term mutual exclusion mechanism that can be used to prevent race conditions during short critical sections of code. A spin lock is acquired prior to entering the critical section and released upon completion. Spin locks derive their name from the fact that a processor will busy-wait (spinning in a loop) when waiting for a lock that is in use by another processor. Spin locks are the only MP primitive operation needed to implement a master-slave kernel.

Spin locks are implemented using a single word in memory that reflects the current status of the lock. A lock is acquired for exclusive use by a particular processor when that processor is able to change the status of the spin lock from the unlocked to the locked state atomically. This must be done as an atomic operation to ensure that only one processor can acquire the lock at a time. For the examples that follow, a value of zero will be used to represent the unlocked state of a spin lock. All routines take a pointer to the spin lock status word to be acted upon. A spin lock can be initialized with the routine shown in Figure 5.1.

```
typedef int lock_t;

void initlock(lock_t *lock_status) {
  *lock_status = 0;
}
```

Figure 5.1: Initializing a spin lock.

Using the test-and-set instruction presented in section 4.3.3, the function in Figure 5.2 can be used to lock a spin lock atomically. Recall that the `test_and_set` function returns 1 if the previous state was nonzero, and 0 otherwise.

```
void lock(lock_t *lock_status) {
  while (test_and_set(lock_status) == 1)
    ;
}
```

Figure 5.2: Atomically locking a spin lock.

The function in Figure 5.2 locks a spin lock by atomically changing its state from 0 to 1. If the lock status is already 1 (meaning the lock is in use by another processor), then the `test_and_set` function returns 1, and the processor spins in the loop until the lock is released. A spin lock is released by simply setting the lock status to 0, as the code in Figure 5.3 shows.

```
void unlock(lock_t *lock_status) {
  *lock_status = 0;
}
```

Figure 5.3: Unlocking a spin lock.

Spin locks work correctly for systems with any number of processors. If multiple processors try to acquire the same lock at exactly the same time, the atomic nature of the `test_and_set` function allows only one processor at a time to change the lock status from 0 to 1. The other processors will

see the lock is already set and spin until the processor owning the lock releases it. The kernel can now form a critical section by surrounding it with `lock` and `unlock` function calls:

```
lock(&spin_lock);
/* perform critical section */
...
unlock(&spin_lock);
```

Spin locks work well if the critical section is short (usually no more than a few hundred machine instructions). They should not be used as a long-term mutual exclusion technique, because processors waiting for the lock do not perform any useful work while spinning. Overall system performance will be lowered if the processors spend too much time waiting to acquire locks. This can also happen if too many processors frequently contend for the same lock.

Two things can be done to reduce lock contention. First, the kernel can use different spin locks for different critical resources. This prevents processors from being held up by other processors when there is no threat of a race condition. Second, the `lock` and `unlock` functions should be enhanced to block interrupts while the lock is held. Otherwise, an interrupt occurring while the processor holds a spin lock will further delay other processors waiting for that lock and might result in a deadlock if the interrupt handler tries to acquire the same lock.

5.1.2 Master-Slave Kernel Implementation

With a master-slave kernel implementation, the only critical resources are the two run queues. The operation of enqueuing and dequeuing processes from them must be done with mutual exclusion to prevent the queues from being corrupted. This is easily accomplished by protecting each queue with a spin lock.

The slave processor can only execute processes that are on the slave run queue. If there are no processes on the queue, the slave simply busy-waits in the loop until one becomes available. When a process running on a slave executes a system call or generates a trap, it is simply switched over to the master processor and placed on the master run queue. Since the master processor can run either kernel- or user-mode processes, it can select processes from either queue. The process selection algorithm should give preferences to choosing a process off the master run queue since those processes can only be run on the master.

Each processor in a master-slave implementation receives and handles its own clock interrupts. To keep the uniprocessor short-term mutual exclusion policy in place, all the usual kernel activity associated with a clock interrupt is handled by the master processor. The clock interrupt handler on the slave simply has to check for situations where a context switch should be performed. This can happen in three instances: when the current process's time quantum expires, when a higher priority process is added to the slave run queue, or when a signal has been posted to the process. In the first two cases, the executing process is put back on the slave run queue. In the third case, the process must be switched to the master processor so that the kernel code required to handle the signal can run without the risk of race conditions.

5.1.3 Performance Considerations

Consider two different benchmarks run on a master-slave MP implementation. The first benchmark consists of a group of processes that are completely compute bound. Once started, they generate no page faults, traps, nor system calls, and perform no I/O. Such a benchmark will show a nearly ideal linear increase in system throughput as additional processors are added to the system. Since the benchmark spends all his time in user mode, all the slave processors can be fully utilized. On the other hand, a second benchmark that consists of the same number of processes, except that the processes are system call bound, will show the opposite result. If each process merely executes a trivial system call continuously in a tight loop, with as little user-level processing as possible, this benchmark will show *no performance improvement over an UP system*, regardless of the number of processors present in the MP system. In this scenario, all processes in the benchmark require continuous service by the kernel. Since only the master processor can provide this service, the slaves will sit idle throughout the benchmark.

It can be concluded from this that a master-slave implementation is a poor choice for highly interactive (or otherwise I/O intensive) application environments because of the high system call and I/O activity of these applications. It would be a good choice for compute bound scientific application environments.

Application mixes that lie between the extremes of compute bound or system call bound may still be able to benefit from additional slave processors. If an application mix is run on a UP and found to spend 50 percent of its time executing in kernel, 40 percent executing at user-level, and 10 percent waiting for I/O to complete, for example, then the 40 percent executing at user-level can be distributed to the slave in a two-processor master-slave MP system. This will result in, at most, a 40-percent performance improvement. This assumes that all the user-level work can be done in parallel with the kernel and I/O activities. If it turns out that only half of the user work can be done in parallel with the kernel and I/O work, then a 20-percent performance improvement will be seen. Adding additional processors will not increase performance any further because they can do nothing to reduce the 50 percent of the time the application mix spends running on the master, nor can they reduce the time spent waiting for I/O. This is primary problem with master-slave kernels when used in non-compute bound environments: the master quickly becomes a bottleneck and prevents throughput from increasing when additional processors are added. It may not be economical to add a second processor for only a 20-percent performance improvement. Anything beyond a two-CPU master-slave MP system is almost always uneconomical for situations such as these.

5.2 Spin-Locked Kernels

To make SMP systems cost effective for highly interactive applications, the kernel must be made to support simultaneous system calls from different processes running on different processors. Such a kernel implementation allows multiple threads of kernel activity to be in progress at once and is referred to as a *multithreaded* kernel. To multithread an operating system, all critical regions must be identified and protected in some way. Spin locks are one such mechanism that can provide this protection.

When spin locks are used, the *granularity* of the locks must be decided. This refers to how many spin locks are used and to how much data is protected by any one lock. A *coarse-grained* implementation uses few locks, each of which protects a large amount of the kernel, perhaps an entire subsystem. A *fine-grained* implementation uses many locks, some of which may protect only a single data structure element. The choice of using a coarse- or fine-grained implementation is both a space and time trade-off. If each lock requires one word, then an extremely fine-grained implementation will probably add one word to every instance of every data structure in the system. At the other extreme, a very coarse-grained implementation uses little extra space but may hold locks for long periods of time, causing other processors to spin excessively while waiting for the locks to free.

5.2.1 Giant Locking

The use of only a handful of spin locks throughout the kernel is referred to as *giant locking*. The easiest giant-locked kernel to implement is one that uses only a single lock. At this extreme, the lock protects all kernel data and prevents more that one process from executing in kernel mode at a time. The lock is acquired on any entry to the kernel, such as system call or trap, and released only when the process context-switches or returns to user mode. Interrupt handlers must also acquire the kernel giant lock, since they represent an entry point into the kernel as well.

The giant-locking technique is similar to master-slave kernels. The difference is that with a giant-locked kernel any processor can execute kernel code, making context switching to the master unnecessary. This may seem like an advantage at first, but a giant-locked kernel may perform worse than a master-slave kernel. If a process executes a system call while another processor is holding the kernel lock, then the processor will sit idle while spinning for the lock. In a master-slave kernel, the processor could have selected another user-mode process to run instead of simply sitting idle.

5.2.2 Coarse-Grained Locking

The giant-locking technique that employs only a single lock can be expanded to allow some additional parallelism by adding more locks. By using separate locks for each subsystem within the kernel, for example, processes on different processors can be simultaneously performing system calls if they affect only different kernel subsystems. While a coarse-grained MP kernel implementation begins to allow parallel kernel activity on more than one processor, it is still too restrictive for system-call-intensive workloads. It will not generally provide much performance increase over master-slave kernels to those application job mixes that require large amounts of kernel services.

5.2.3 Fine-Grained Locking

The goal of fine-grained locking is to increase the amount of parallel kernel activity by different processors. While many processes may want to use the same kernel subsystem, it is frequently the case that they will be using separate portions of the associated data structure. Each subsystem can be divided into separate critical resources, each with its own lock. This requires an analysis of all data structures that were protected by the uniprocessor short-term mutual exclusion technique, and adding the appropriate spin locks.

5.2.4 Kernel Preemption

The original uniprocessor UNIX kernel was nonpreemptable by design to simplify the implementation of short-term mutual exclusion. As seen, all cases where short-term mutual exclusion is required must be explicitly protected by a spin lock in order for a multithreaded kernel implementation to function correctly on an MP system. Since short-term mutual exclusion has now been made explicit, the old UP approach of not preempting processes executing in kernel mode can be relaxed. Anytime a process is not holding a spin lock and has not blocked any interrupts, then it is not executing in a critical region and can be preempted. When a kernel process is preempted under these circumstances, it is guaranteed that it is only accessing private data. This observation is useful on real-time systems, for example, as it can reduce the dispatch latency when a higher priority process becomes runnable.

5.2.5 Performance Considerations

As before, compute bound jobs can run in parallel on all processors without contention. Not surprisingly, a mutithreaded kernel will neither help nor hinder the performance of such applications. The difference lies in the performance of the system when running system-call-intensive applications. If the processes in the application job mix uses separate kernel resources, each of whose data structures are protected by separate locks, then these processes will not contend for the same lock and will be able to run simultaneously on different CPUs, whether they are in user or kernel mode.

5.3 Semaphored Kernels

Operating system designers have turned to other primitives to replace `sleep` and `wakeup` in multiprocessor UNIX systems. As with spin locks, these primitives are designed to function correctly in an MP environment regardless of the number of processors in the system. They differ from spin locks in that they allow a process to be suspended if certain criteria cannot be immediately satisfied. While a variety of such primitives has been invented, one of the simplest is the Dijkstra *semaphore* [Dij65].

The state of a semaphore is represented by a signed integer value upon which two atomic operations are defined. The $P(s)$ operation decrements the integer value associated with semaphore s by 1 and blocks the process if the new value is less than zero. If the new value is greater than or equal zero, the process is allowed to continue execution. The converse is the $V(s)$ operation, which increments the semaphore value by one and, if the new value is less than or equal to zero, unblocks one process that was suspended during a P operation on that semaphore. The process executing the V operation is never blocked. A queue of processes blocked on the semaphore is maintained as part of the semaphore state, in addition to the integer value. When the value of the semaphore is negative, the absolute value indicates the number of processes on the queue.

The distinguishing factor between P and V and the uniprocessor `sleep` and `wakeup` functions is that semaphores are higher level operations that make both the decision to block and the act of blocking the process an atomic operation. In addition, the V operation has the desired effect of unblocking only a single process at a time. Since semaphores have a state associated with them, it is

unnecessary to keep external flags to record the state of the lock. Furthermore, this state information means that semaphores retain a "memory" of past operations.

5.3.1 Mutual Exclusion with Semaphores

By initializing the semaphore value to 1, long-term mutual exclusion locks can be implemented. A critical section is then protected by using a *P* operation before entering it, and using *V* upon exiting (see Figure 5.4). This will allow only one process in the critical region at a time. Once the first process enters the critical region, it will decrement the semaphore value to zero. The process will be allowed to proceed, but any other processes following will block when they execute the *P* operation. When the first process exits the critical region, the *V* operation will allow another to enter by unblocking a waiting process if there is one.

```
P(s)
perform critical section
V(s)
```

Figure 5.4: Critical region protected by a semaphore.

Semaphores used for mutual exclusion in this fashion are called *binary semaphores*. This refers to the fact that the binary semaphore logically has only two states: locked and unlocked. The semaphore value for a binary semaphore is never greater than 1, as this would allow more than one process to be in the critical section at a time.

5.3.2 Synchronization with Semaphores

Semaphores can be used for process synchronization by initializing the value to zero. This allows one process to wait for an event to occur by using the *P* operation. Since the semaphore value is initially zero, the process will block immediately. Completion of the event can be signaled by another process using a *V* operation. The *V* operation causes a process waiting for the event to be awakened and run. Because the semaphore is incremented by a *V* operation even if no processes are blocked on the semaphore, an event that is signaled before the first process can perform the *P* operation causes that process to continue without waiting.

5.3.3 Resource Allocation with Semaphores

Finally, semaphores can be used to control the allocation of resources from a finite pool. For example, assume there is a pool a special-purpose buffers in the system that processes must periodically allocate. If no buffers are presently available, a process requesting one will wait until one becomes available. By initializing a semaphore to the number of buffers in the pool, a *P* operation can be done to reserve a buffer. As long as the semaphore value remains greater then zero, each process reserving a buffer will be allowed to proceed without blocking. A *V* operation is done when a buffer is released. Once all buffers are in use, new processes reserving a buffer are blocked. When a buffer is released, a process waiting for a buffer is awakened and allowed to continue (see Figure 5.5).

```
P(s)
allocate and use buffer
...
return buffer to pool
V(s)
```

Figure 5.5: Resource reservation with semaphores.

5.3.4 Implementing Semaphores

The advent of RISC processors means that operations like semaphores, which require multiple memory accesses to be done atomically, are rarely implemented in hardware. Instead, semaphores must be built on top of the basic hardware atomic operations by software. An example of an implementation of semaphores will be shown in Section 11.1.8 as part of the TopsySMP project.

In some cases, critical sections can be parallelized by allowing multiple processes that need merely to read various data structures to do so concurrently. As long as none of the processes modify any of the data structures, no races can occur. When a process does need to make a modification, it can do so by waiting for the reader processes to finish and then make its modifications with mutually exclusive access from all other readers and writers. Such a locking scheme is termed a *multireader, single writer* lock (or simply a multireader lock for short). An implementation of a multireader lock based on semaphores is shown in Section 11.1.8 as part of the TopsySMP project.

5.4 Summary

A uniprocessor kernel cannot be run on an SMP system without modification. The short-term mutual exclusion technique used by UP kernels relies on the fact that there is never more that one process executing in the kernel at the same time. One way an MP system can keep this policy in effect is to restrict all kernel activity to one processor in the system. The *master* processor services all system calls, interrupts, and many other kernel activity. The other processors in the system, the *slaves*, can execute processes only while they are running in user mode. Once a process running on a slave begins a system call or generates a trap, it must be switched to the master.

In such an implementation, the only critical resource is the run queue. A technique is needed to serialize access to the run queues by multiple processors so that race conditions are prevented. *Spin locks* can be used for this. Spin locks can be implemented by atomically updating a memory location so that only one processor can successfully acquire the lock at any time. Once a processor acquires a spin lock, all other processors attempting to acquire the lock will busy-wait until it is released.

In a master-slave kernel, the master tends to be the limiting factor in the overall performance of the system. Once the master processor is saturated, adding additional slaves will not improve performance since they cannot share the kernel load in the general case. A number of simple system calls can be run on the slaves, but since these are not the system calls that consume most of the time on the master, running them on the slaves will not improve performance. Performance can be significantly improved only by allowing parallel kernel activity on multiple processors.

The amount of concurrent kernel activity that is possible across all the processors is partially determined by the degree of multithreading. A *coarse-grained* implementation uses few locks, but can suffer the same drawbacks as a master-slave kernel when the applications require frequent kernel service. A *fine-grained* implementation attempts to overcome these drawbacks by protecting different data structures with different locks. The goal is to ensure that unrelated processes can perform their kernel activities without contention.

An advantage of a multithreaded kernel is that all critical sections are protected either by a long-term lock or spin lock, or by raising the interrupt priority level. This makes it possible to preempt kernel processes that are not holding any spin locks and that do not have any interrupt blocked.

Semaphores provide a useful MP primitive that can replace all uses of the uniprocessor `sleep/wakeup` mechanism. A semaphore can implement either mutual exclusion or process synchronization and works correctly for any number of processors in the system, including the uniprocessor case. The advantage of semaphores over the `sleep/wakeup` mechanism is that they update the status of the semaphore and block or unblock a process, all as an atomic operation. Second, semaphores awaken only a single process at a time, which eliminates unnecessary trashing. Consequently, semaphores are good for implementing long-term mutual exclusion. They are also superior to spin locks when implementing giant locks in coarse-grained kernels. Semaphores should not, however, be used as a complete replacement for spin locks. Spin locks are still preferable for implementing mutual exclusion in short critical sections. Attempting to use semaphores in these cases introduces unnecessary context switching overhead, which can be greater than the time that would have been spent spinning.

Part II

REVIEW OF EXISTING MP SYSTEMS

6 SMP Hardware

This chapter gives a short overview over two existing MP hardware architectures. The first one is the *SPARC Multiprocessor System Architecture* from Sun, and the second one is the *Multiprocessor Specification* from Intel.

6.1 SPARC Multiprocessor System Architecture

The Sun MP implementation is based on a tightly coupled, shared memory architecture. At the processor core level, processors are tightly coupled by a high-speed module interconnection bus, *MBus*, and share the same image of memory, although each processor has a cache where recently accessed data and instructions are stored (see Figure 6.1). All CPUs have equal access to system services, such as I/O.

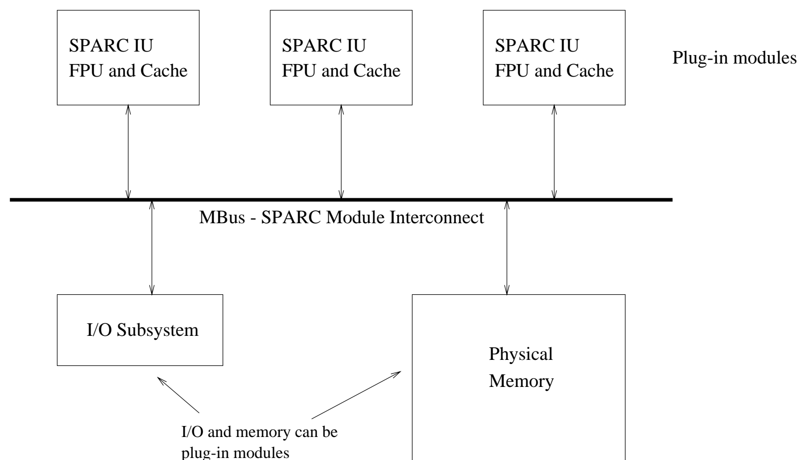


Figure 6.1: SPARC/MBus Multiprocessing Architecture

Providing a standard interface between CPUs and the rest of the system allows modular upgrade to future processor implementations. The interface also provides a convenient way to add more processors to a basic system. Adding or changing processors is accomplished by installing a SPARC module, which attaches to the MBus connectors on the system board. A module's processor core consists of a CPU, a floating-point unit (FPU), a memory management unit/cache controller (MMU/CC), and local cache. In its first multiprocessor implementation, SPARC modules contained two processor cores (each with a 64 Kbyte cache). A single system board supports one or two modules for two- or four-way multiprocessing.

6.1.1 Multi-level Bus Architecture

The SPARC MBus is a high-speed interface bus that connects SPARC compute modules to physical memory modules and special purpose I/O modules. It is a fully synchronous 64-bit, multiplexed address and data bus that supports multiple masters at 40 MHz. It supports data transfer of up to 128 bytes and can support up to 64 Gbytes of physical address space. Data transfer larger that 8 bytes occur as burst transfer, allowing MBus to achieve a peak bandwidth of 320 Mbytes/sec. Bus arbitration among masters is defined to be handled by a central bus arbiter. Requests from MBus modules are sent to the arbiter, and ownership is granted to the module with the highest priority. The MBus can accommodate up to 16 modules on a single printed circuit board.

6.1.2 MBus Multiprocessor System Design

The MBus Interface Specification defines two levels of compliance, Level 1 and Level 2. Level 1 allows for the operations of read and write of sizes ranging from 1–128 bytes. Level 2 adds the additional signals and transaction needed to design a fully symmetric, cache-coherent, shared-memory multiprocessor system.

The system performance bottleneck in designing a shared memory MP system tends to be bus bandwidth. MBus modules include a private cache in order to reduce memory bus traffic. Either write-back or write-through caches are supported, but in order to be cache-consistent in Level 2, the caches in MBus systems must be write-back. MBus caches must also have a write-allocate policy where the contents of the cache line are updated on a write miss.

While a write-back cache reduces bus traffic, it also introduces a problem, since a cache can have a modified line that is inconsistent with memory. When another module requests that line, the most up-to-date copy must be provided. This problem is solved by MBus implementing a write-invalidate, ownership-based protocol modeled after that used by the IEEE Futurebus.

6.1.3 Multiprocessor System Implementation

The following table gives a survey of three system implementations of a generation of symmetric, high-performance, highly configurable SPARC multiprocessor systems available from Sun Microsystems.

	SPARCserver 600MP	**SPARCserver 1000**	**SPARCcenter 2000**
Number of CPUs	1–4	1–8	2–20
Number of System Boards	1	1–4	1–10
Processor	SuperSPARC		
Cache Size	1 MB per CPU		
Clock Speed	40–45 MHz	50 MHz	40 MHz
Main Memory	64 MB–1 GB	32 MB–2 GB	64 MB–5 GB

6.2 Intel Multiprocessor Specification

The Multi-Processor Specification from Intel (referred to as the MP specification), defines an enhancement to the standard to which PC manufacturers design DOS-compatible systems. MP-capable operating systems will be able to run without special customization on multiprocessor systems that comply with the MP specification. Details about the Intel MP specification can be found in [Int97].

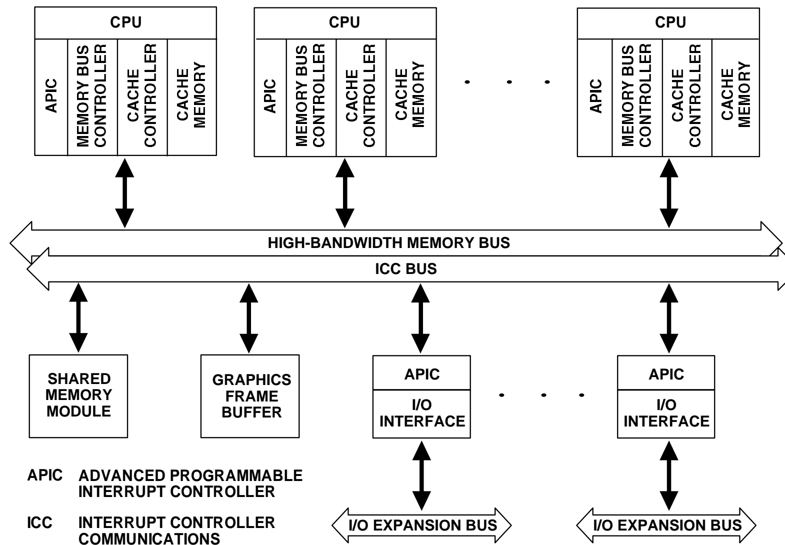


Figure 6.2: Intel MP System Architecture

6.2.1 System Overview

Figure 6.2 shows the general structure of a design based on the MP specification. The MP specification's model of multiprocessor system incorporates a tightly-coupled, shared-memory architecture with a distributed interprocessor and I/O interrupt capability. It is fully symmetric; that is, all processors are functionally identical and of equal status, and each processor can communicate with every other processor. There is no hierarchy, no master-slave relationship, no geometry that limits communication only to "neighboring" processors. The model is symmetric in two important respects:

- *Memory symmetry*. Memory is symmetric when all processors share the same memory space and access that space by the same addresses.
- *I/O symmetry*. I/O symmetry is when all processors share access to the same I/O subsystem and any processor can receive interrupts from any source.

6.2.2 Hardware Overview

The MP specification defines a system architecture based on the following hardware components:

- One or more processors that are Intel architecture instruction set compatible
- One or more Advanced Programmable Interrupt Controller (APIC)
- Software-transparent cache and shared memory subsystem
- Software-visible components of the PC/AT platform

System Processors

To maintain compatibility with existing PC/AT software products, the MP specification is based on the Intel486 and the Pentium processor family. While all processors in a compliant system are functionally identical, the MP specification classifies them into two types: the *bootstrap processor (BSP)* and the *application processors (AP)*. Which processor is the BSP is determined by the hardware or by the hardware in conjunction with the BIOS. This differentiation is for convenience and is in effect only during the initialization and shutdown process. The BSP is responsible for initializing the system and for booting the operating system; APs are activated only after the operating system is up and running.

APIC

Multiple local and I/O APIC units operate together as a single entity, communicating with one another over the *Interrupt Controller Communications (ICC)* bus. The APIC units are collectively responsible for delivering interrupts from interrupt sources to interrupt destinations throughout the multiprocessor system. In a compliant system, all APICs must be implemented as memory-mapped I/O devices.

System Memory

Compared to a uniprocessor system, a symmetric multiprocessor system imposes a high demand for memory bus bandwidth. The demand is proportional to the number of processors on the memory bus. To reduce memory bus bandwidth limitations, an implementation of the MP specification should use a secondary cache that has high-performance features, such as write-back update policy and a snooping cache-consistency protocol.

6.2.3 BIOS Overview

A BIOS functions as an insulator between the hardware on one hand, and the operating system and application software on the other. For a MP system, the BIOS may perform the following functions:

- Pass configuration information to the operating system that identifies all processors and other multiprocessing components of the system.
- Initialize all processors and the rest of the multiprocessing components to a known state.

6.2.4 MP Configuration Table

The operating system must have access to some information about the multiprocessor configuration. The MP specification provides two methods for passing this information to the operating system: a minimal method for configurations that conform to one of a set of common hardware defaults, and a maximal method that provides the utmost flexibility in hardware design. Figure 6.3 shows the general layout on the MP configuration data structures.

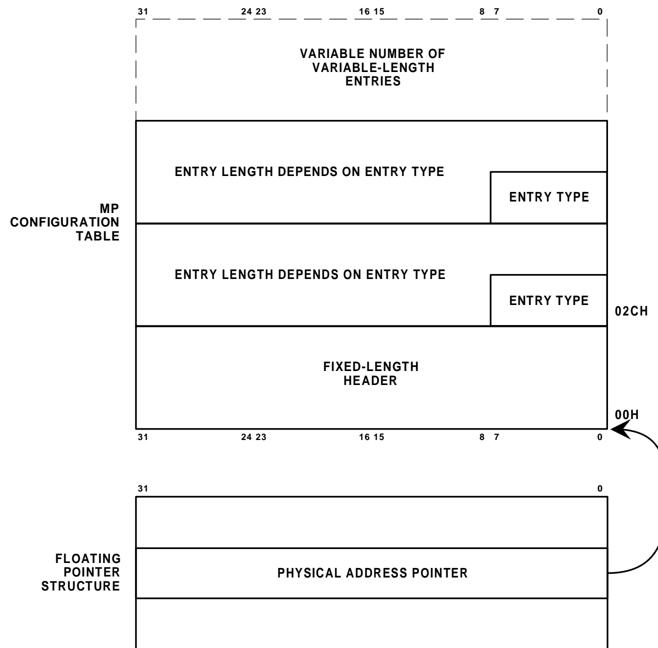


Figure 6.3: MP Configuration Data Structures

The following two data structures are used:

1. The *MP Floating Point Structure*. This structure contains a physical address pointer to the MP configuration table and other MP feature information bytes. When present, this structure indicates that the system conforms to the MP specification.

2. The *MP Configuration Table*. The MP configuration table contains explicit configuration information about APICs, processors, buses, and interrupts.

A variable number of variable length entries follow the header of the MP configuration table. The first byte of each entry identifies the entry type. Each entry has a known, fixed length. Table 6.1 gives the meaning of each entry type.

6.2.5 Default Configurations

The MP specification defines several default MP system configurations. The purpose of these defaults is to simplify BIOS design. If a system conforms to one of the default configurations, the

Entry Description	Comments
Processor	One entry per processor.
Bus	One entry per bus.
I/O APIC	One entry per I/O APIC.
I/O Interrupt Assignment	One entry per bus interrupt source.
Local Interrupt Assignment	One entry per system interrupt source.
System Address Space Mapping	Entry to declare system visible memory or I/O space on a bus.
Bus Hierarchy Descriptor	Entry to describe I/O bus interconnection.

Table 6.1: MP Configuration Entry Types

BIOS will not need to provide the MP configuration table.

To use the default configuration, a system must meet the following basic criteria:

1. The system supports two processors.
2. Both processors must execute the common Intel architecture instruction set.
3. The system uses discrete or integrated APICs at fixed base memory addresses.

The following table specifies two of the most commonly used default configurations:

Config Code	Number of CPUs	Bus Type	APIC Type
1	2	ISA	82489DX
5	2	ISA + PCI	Integrated

7 SMP Operating Systems

This chapter covers a review of existing SMP operating systems. Among the many existing operating systems, three were selected to study their implementation of an SMP kernel. First, as a representative of commercial operating systems, Solaris was chosen. Second, the Mach kernel as a member of the academic world. And third, the Intel i386 version of Linux was picked to represent a freely distributable OS developed by a variety of programmers all over the world. Besides, Linux is an example of an operating system running on implementations of the MP specification by Intel.

7.1 Solaris

AT&T's UNIX System V Release 4 (SVR4) was the result of evolution and conformance in the industry. Efforts began in the mid-1980s to unify the variants into a single UNIX operating system that would serve as an open computing platform for the 1990s. In 1987, Sun and AT&T formally announced a joint effort to develop this platform. In 1988, UNIX International was formed to provide industry-wide representation to the process of creating an evolving SVR4. As a result of these efforts, SVR4 complies with most existing industry standards and contains important functionality from the main variants of UNIX: SVR3, BSD 4.2/4.3, SunOS, and Xenix.

In addition to having all the functionality and interfaces provided by SVR4, the Solaris operating environment also has several value-added features above and below the standard interfaces. The most important feature is the SunOS multithreaded architecture.

7.1.1 System Overview

SunOS 5.x contains some core modules and other modules, such as device drivers, file systems, and individual system calls which are dynamically loaded into the kernel as needed. The core of SunOS 5.x is a real-time nucleus that supports kernel threads of control. Kernel threads are also used to support multiple threads of control, called *lightweight processes (LWP)* within a single process. Kernel threads are dispatched in priority order on the pool of available processors. The kernel also provides preemptive scheduling with very few nonpreemptive points.

SunOS 5.x is intended to run on uniprocessor systems and tightly coupled shared-memory multiprocessors. The kernel assumes all processors are equivalent. Processors select kernel threads from the queue of runnable threads. The shared memory is assumed to be symmetrical.

SimOS 5.x provides a relatively fine-grained locking strategy to take advantage of as many processors as possible. Each kernel subsystem has a locking strategy designed to allow a high degree

of concurrency for frequent operations. In general, access to data items are protected by locks as opposed to entire routines. Infrequent operations are usually coarsely locked with simple mutual exclusions. Overall, SunOS 5.x has several hundred distinct synchronization objects statically, and can have many thousands of synchronization objects dynamically. In order to protect and arbitrate to critical data structures, synchronization locks use the invisible test-and-set instructions (`swap` and `ldstub`), provided by the SPARC architecture. Unlike traditional UNIX implementations, interrupt levels are not used to provide mutual exclusion.

7.1.2 SunOS Kernel Architecture

Each kernel thread is a single flow of control within the kernel's address space. The kernel threads are fully preemptible and can be scheduled by any of the scheduling classes in the system. Since all other execution entities are built by using kernel threads, they represent a fully preemptive, real-time "nucleus" within the kernel.

Interrupts are also handled by kernel threads. The kernel synchronizes with interrupt handlers via normal thread synchronization primitives. For example, if an interrupt thread encounters a locked mutex, it blocks until the mutex is unlocked. In SunOS, synchronization variables, rather than processor priority levels are used to control access to all shared kernel data.

Data Structures

In the traditional UNIX kernel, the user and proc structures contained all kernel data for the process. Processor data was held in global variables and data structures. The per-process data was divided among non-swappable data in the proc structure and swappable data in the user structure. The kernel stack of the process, which is also swappable, was allocated with the user structure in the user area, usually one or two pages long.

The SunOS kernel separates this data into data associated with each LWP and its kernel thread, the data associated with each process, and the data associated with each processor. The per-process data is contained in the proc structure. To speed access to the thread, LWP, process, and CPU structures, the SPARC implementation uses a global register to point to the current thread structure.

Scheduling

SunOS 5.x provides several scheduling classes. Every kernel thread is associated with a scheduling class, which determines how kernel-level threads are dispatched with respect to each other. The scheduling classes currently supported are *sys* (system), *timesharing*, and *realtime* (fixed-priority). The scheduler chooses the thread with the greatest global priority to run on the CPU. If more than one thread has the same priority, they are dispatched in round-robin order.

SunOS 5.x is fully preemptible, which enables the implementation of a real-time scheduling class and support for interrupt threads. Preemption is disabled only in a few places for short period of time; that is, a runnable thread runs as soon as possible after its priority becomes high enough.

Synchronization Architecture

The kernel implements the same synchronization objects for internal use as are provided by the user-level libraries for use in multithreaded application programs. These objects are mutual exclusion locks, conditional variables, semaphores, and multiple readers, single writer locks.

Synchronization objects are all implemented such that the behavior of the synchronization object is specified when it is initialized. Synchronization operations, such as acquiring a mutex lock, take a pointer to the object as an argument and may behave somewhat differently, depending on the type and optional type-specific argument specified when the object was initialized. Most of the synchronization objects have types that enable collecting statistics, such as blocking counts or timers. A patchable kernel variable can also set the default types to enable statistics gathering. This technique allows the selection of statistics gathering on particular synchronization objects or on the kernel as a whole.

The semantics of most of the synchronization primitives cause the calling thread to be prevented from processing past the primitive until some condition is satisfied. The way in which further progress is impeded (e.g., sleep, spin, or other) is a function of the initialization. By default, the kernel thread synchronization primitives that can logically block, can potentially sleep.

A variant of the conditional variable wait primitive and the semaphore inherent primitive are provided for situations where a kernel thread may block for long or indeterminate periods, but still be interruptible when signaled. There is no nonlocal jump to the head of the system call, as a traditional sleep routine might contain. When a signal is pending, the primitive returns with a value indicating that the blocking was interrupted by a signal and the caller must release any resource and return.

Implementing Interrupts as Threads

Since interrupts in SunOS are kernel threads, the kernel synchronizes with interrupt handlers via normal thread synchronization primitives. Most other implementations use processor priority levels.

Interrupts must be efficient, so a full thread creation for each interrupt is impractical. Instead, SunOS 5.x preallocates interrupt threads, already partly initialized. When an interrupt occurs, a minimum amount of work is needed to move onto the stack of an interrupt thread and set it as the current thread.

Interrupt Thread Cost

The additional overhead in taking an interrupt is about 40 SPARC instructions. The savings in the mutex acquire/release path is about 12 instructions. However, mutex operations are much more frequent than interrupts, so there is a net gain in time cost, as long as interrupts do not block too frequently.

There is a cost in terms of memory usage also. Currently, an interrupt thread is preallocated for each potential active interrupt level below the thread level for each CPU. Nine interrupt levels on the Sun SPARC implementation can potentially use threads. An additional interrupt thread is preallocated for the clock (one per system). Each kernel thread requires at least 8 Kbytes of memory for a stack. The memory cost can be higher if there are many interrupt threads.

Clock Interrupt

The clock interrupt is worth noting because it is handled specially. A clock interrupt occurs every 10 ms, or 100 times a second. There is only one clock interrupt thread in the system (not one per CPU); the clock interrupt handler invokes the clock thread only if it is not already active.

7.2 Mach

Mach's earliest roots go back to a system called *RIG (Rochester Intelligent Gateway)* which began at the University of Rochester in 1975. Its main research goal was to demonstrate that operating systems could be structured in a modular way, as a collection of processes that communicated by message passing. When one of its designers, Richard Rashid, left the University of Rochester and moved to Carnegie-Mellon University (CMU) in 1979, he wanted to continue developing message passing operating systems.

By 1984 Rashid began a third generation operating system project called *Mach*. By making Mach compatible with UNIX, he hoped to be able to use the large volume of UNIX software becoming available. Around this time, DARPA, the U.S. Department of Defense's Advanced Research Projects Agency was looking for an operating system that supported multiprocessors as a part of its Strategic Computing Initiative. CMU was selected, and with DARPA funding, Mach was developed further. It was decided to make the system compatible with 4.2BSD by combining Mach and 4.2BSD into a single kernel.

The first version of Mach was released in 1986 for the VAX 11/784, a four-CPU multiprocessor. Shortly thereafter, ports to IBM PC/AT and Sun 3 were done. As of 1988, the Mach 2.5 kernel was large and monolithic due to the presence of a large amount of Berkley UNIX code in the kernel. In 1989, CMU removed all the Berkley code from the kernel and put it in user space. What remained was a microkernel consisting of pure Mach. In this chapter, we will focus on the Mach 3.0 microkernel and one user-level operating system emulator, for BSD UNIX.

7.2.1 The Mach Microkernel

The Mach microkernel has been built as a base upon which UNIX and other operating systems can be emulated. This emulation is done by a software layer that runs outside the kernel, in user space, as shown in Figure 7.1.

The Mach kernel, like other microkernels, provides process management, memory management, communication, and I/O services. Files, directories, and other traditional operating system functions are handled in user space. The idea behind the Mach kernel is to provide the necessary mechanisms for making the system work, but leaving the policy to user-level processes.

The kernel manages five principal abstractions: Processes, threads, memory objects, ports, and messages. A *process* is basically an environment in which execution can take place. It has an address space holding the program text and data, and usually one or more stacks. The process is the basic unit for resource allocation. A *thread* in Mach is an executable entity. Is has a program counter and a set of registers associated with it. Each thread is part of exactly one process. A process with one thread is similar to a traditional (e.g. UNIX) process. A concept that is unique to Mach is the *memory object*, a data structure that can be mapped into a process' address space. Memory objects occupy one or more pages, and form the basis of the Mach virtual memory system. Interprocess communication in Mach is based on message passing. To receive messages, a user process asks the kernel to create a kind of protected mailbox, called a *port*, for it. The port is stored inside the kernel, and has the ability to queue an ordered list on messages.

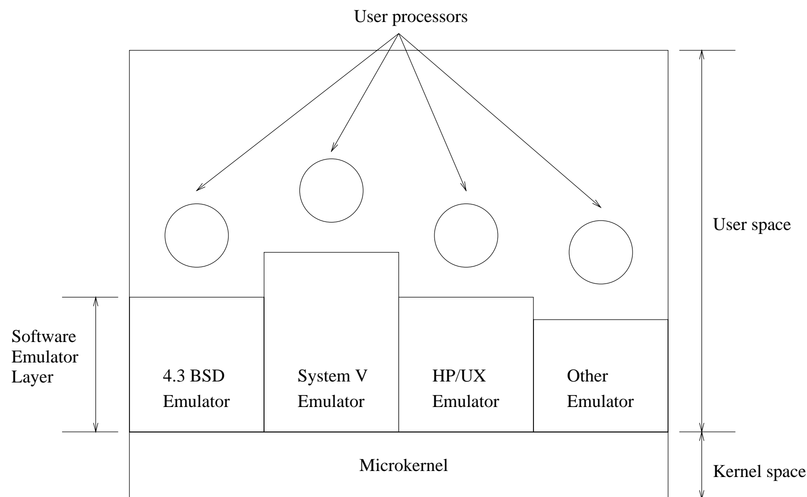


Figure 7.1: Abstract Model for UNIX Emulation using Mach

7.2.2 Process Management

Process management in Mach deals with processes, threads, and scheduling.

Processes

A process in Mach consists primarily of an address space and a collection of threads that execute in that address space. Processes are passive. Execution is associated with the thread. Processes are used for collecting all the resources related to a group of cooperating threads into convenient containers.

A process can be runnable or blocked, independent of the state of its threads. If a process is runnable, then those threads that are also runnable can be scheduled and run. If a process is blocked, its threads may not run, no matter what state they are in. The scheduling parameters include the ability to specify which processors the process' threads can run on. For example, the process can use this power to force each thread to run on a different processor, or to force them all to run on the same processor, or anything in between. In addition, each process has a default priority that is settable. CPU scheduling is done using this priority, so the programmer has a fine-grain control over which threads are the most important and which are the least important.

Threads

The active entities in Mach are the threads. They execute instructions and manipulate their registers and address space. Each thread belongs to exactly one process. A process cannot do anything unless it has one or more threads.

All the threads in a process share the address space and all the process-wide resources. Nevertheless, threads also have private per-thread resources. One of these is the *thread-port*, which is used to invoke thread-specific kernel services, such as exiting when the thread is finished. Since ports are process-wide resources, each thread has access to its siblings' ports, so each thread can control the others.

On a single CPU system, threads are timeshared. On a multiprocessor, several threads can be active at the same time. This parallelism makes mutual exclusion, synchronization, and scheduling more

important then they normally are, because performance now becomes a major issue, among with correctness. Since Mach is intended to run on multiprocessors, these issues have received special attention. Synchronization is done using mutexes and conditional variables. The mutex primitives are `lock`, `trylock`, and `unlock`. Primitives are also provided to allocate and free mutexes. The operations on condition variables are `signal`, `wait`, and `broadcast`.

Scheduling

Mach scheduling has been heavily influenced by its goal of running on multiprocessors. The CPUs in a multiprocessor can be assigned to *processor sets* by software. Each CPU belongs to exactly one processor set. Threads can also be assigned to processor sets by software. The job of the scheduling algorithm is to assign threads to CPUs in a fair and effective way. For purpose of scheduling, each processor set is a closed world, with its own resources and its own customers, independent of all the other processor sets.

This mechanism gives processes a large amount of control over their threads. A process can assign an important thread to a processor set with one CPU and no other threads, thus insuring that the thread runs all the time. It can also dynamically reassign threads to processor sets as the work proceeds, keeping the load balanced.

Thread scheduling in Mach is based on priorities. Priorities are integers from 0 to 31, with 0 being the highest priority and 31 being the lowest priority. Each thread competes for CPU cycles with all other threads, without regard to which thread is in which process. Associated with each processor set is an array of 32 run queues, corresponding to threads currently at priorities 0 through 31. When a thread at priority n becomes runnable, it is put at the end of queue n. A thread that is not runnable is not present on any queue. Each run queue has three variables attached to it. The first one is a mutex that is used to lock the data structure. It is used to make sure that only one CPU at a time is manipulating the queues. The second one is the count of the number of threads on all the queues combined. If this count becomes 0, there is no work to do. The third variable is a hint as to where to find the highest priority thread. It is guaranteed that no thread is at a higher priority, but the highest one may be at a lower priority. This hint allows the search for the highest priority thread to avoid the empty queues at the top.

In addition to the global run queue, each CPU has its own local run queue. Each local run queue holds those threads that are permanently bound to that CPU, for example, because they are device drivers for I/O devices attached to that CPU. These threads can only run on one CPU, so putting them on the global run queue is incorrect.

We can now describe the basic scheduling algorithm. When a thread blocks, exists, or uses up its time quantum, the CPU it is running on first looks on its local run queue to see if there are any active threads. This check merely requires inspecting the count variable associated with the local run queue. If it is nonzero, the CPU begins searching the queue for the highest priority thread, starting at the queue specified by the hint. If the local run queue is empty, the same algorithm is applied to the global queue, the only difference being that the global run queue must be locked before it can be searched. If there are no threads to run on either queue, a special idle thread is run until some thread becomes ready.

If a runnable thread is found, it is scheduled and run for one quantum. At the end of the quantum, both the local and global run queues are checked to see if any other threads at its priority or higher

are runnable. If a suitable candidate is found, a thread switch occurs. If not, the thread is run for another quantum. Threads may also be preempted. On multiprocessors, the length of the quantum is variable, depending on the number of threads that are runnable. The more runnable threads and the fewer CPUs there are, the shorter the quantum. This algorithm gives good response time to short requests, even on heavily loaded system, but provides high efficiency on lightly loaded systems.

For some applications, a large number of threads may be working together to solve a single problem, and it may be important to control the scheduling in detail. Mach provides a hook to give threads some additional control over their scheduling. The hook is a system call that allows a thread to lower its priority to the absolute minimum for a specified number of seconds. Doing so gives other threads a chance to run. When the time interval is over, the priority is restored to its previous value. This system call has another interesting property: it can name its successor if it wants to. For example, after sending a message to another thread, the sending thread can give up the CPU and request that the receiving thread be allowed to run next. This mechanism, called *handoff scheduling*, bypasses the run queues entirely. If used wisely, it can enhance performance. The kernel also uses it in some circumstances, as an optimization.

Mach can be configured to do affinity scheduling, but generally this option is off. When it is on, the kernel schedules a thread on the CPU it last run on, hoping that part of its address space is still in that CPU's cache. Affinity scheduling is only applicable to multiprocessors.

7.3 Linux

Linux is a freely distributable version of UNIX developed primarily by Linus Torvalds at the University of Helsinki in Finland. Linux was developed with the help of many UNIX programmers and wizards across the Internet, allowing anyone with enough know-how and gumption the ability to develop and change the system. The Linux kernel uses no code from AT&T or any other proprietary source, and much of the software available for Linux is developed by the GNU project at the Free Software Foundation in Cambridge, Massachusetts. However, programmers all over the world have contributed to the growing pool of Linux software.

Linux was originally developed as a hobby project by Linus Torvalds. It was inspired by *Minix*, a small UNIX system developed by Andy Tanenbaum, and the first discussions about Linux were on the USENET newsgroup `comp.os.minix`. These discussions were concerned mostly with the development of a small, academic UNIX system for Minix users who wanted more.

On 5 October 1991, Linus announced the first "official" version of Linux, version 0.02. At this point, Linus was able to run bash (the GNU Bourne Again Shell) and gcc (the GNU C compiler), but not very much else was working. Again, this was intended as a hacker's system. The primary focus was kernel development – none of the issues of user support, documentation, distribution, and so on had even been addressed. The last official stable kernel release is 2.0.36; the following section are based on version 2.2 which will be released soon.

Today, Linux is a complete UNIX clone. Almost all of the major free software packages have been ported to Linux, and commercial software is becoming available. Much more hardware is supported than in original versions of the kernel. Linux was first developed for 386/486-based PCs. These days

it also runs on ARMs, DEC Alphas, SUN Sparcs, M68000 machines, MIPS and PowerPC, and others. Many people have executed benchmarks on 80486 Linux systems and found them comparable with mid-range workstations from Sun Microsystems and Digital Equipment Corporation.

Linux 2.0 includes basic SMP support for Intel and Sun hardware. The current 80x86 kernel supports Intel MP v1.1 and Intel MP v1.4 compliant motherboards with between 1 and 16 Intel processors (486/Pentium/Pentium Pro).

7.3.1 Evolution of Linux SMP

Linux 2.0 started with a single lock, maintained across all processors. This lock is required to access the kernel space. Any processor may hold it and once it is held may also re-enter the kernel for interrupts and other services whenever it likes until the lock is relinquished. This lock ensures that a kernel mode process will not be pre-empted and ensures that blocking interrupts in kernel mode behaves correctly. This is guarantees because only the processor holding the lock can be in kernel mode, only kernel mode processes can disable interrupts and only the processor holding the lock may handle an interrupt.

Such a choice is however poor for performance. It was necessary to move to finer grained parallelism in order to get the best performance. This was done hierarchically by gradually refining the locks to cover smaller areas.

7.3.2 Changes to the Kernel Components

The kernel changes are split into generic SMP support changes and architecture specific changes necessary to accommodate each different processor type Linux is ported to.

Initialization

Linux/SMP defines that only a single processor enters the kernel entry point `start_kernel()`. Other processors are assumed not to be started or to have been captures elsewhere. The first processor begins the normal Linux initialization sequences and sets up paging, interrupts and trap handlers. After it has obtained the processor information about the boot CPU, the architecture specific function `smp_store_cpu_info()` is called to store any information about the processor into a per processor array. Having completed the kernel initialization the architecture specific function `smp_boot_cpus()` is called and is expected to start up each other processor and cause it to enter `start_kernel()` with its paging registers and other control information correctly loaded. Each other processor skips the setup except for calling the trap and irq initialization functions that are needed on some processors to set each CPU up correctly. Each additional CPU then calls the architecture specific function `smp_callin()` which does any final setup and then spins the processor while the boot up processor forks off enough idle threads for each processor. This is necessary because the scheduler assumes there is always something to run. Having generated these threads and forked init, the architecture specific `smp_commence()` function is invoked. This does any final setup and indicates to the system that multiprocessor mode is now active. All the processors spinning in the `smp_callin()` function are now released to run the idle processes, which they will run when they have no real work to process.

Scheduling

The kernel scheduler implements a simple but very effective task scheduler. The basic structure of this scheduler is unchanged in the multiprocessor kernel. A processor field is added to each task, and this maintains the number of the processor executing a given task, or a constant (`NO_PROC_ID`) indicating the job is not allocated to a processor.

Each processor executes the scheduler itself and will select the next task to run from all runnable processes not allocated to a different processor. The algorithm used by the selection is otherwise unchanged. This is actually inadequate for the final system because there are advantages to keeping a process on the same CPU, especially on processor boards with per processor second level cache.

Throughout the uniprocessor kernel the variable `current` is used as a global pointer for the current process. In Linux/SMP this becomes a macro which expands to

```
current_set[smp_processor_id()].
```

This enables almost the entire kernel to be unaware of the array of running processors, but still allows the SMP aware kernel modules to see all of the running processes.

The `fork()` system call is modified to generate multiple processes with a process id of zero until the SMP kernel starts up properly. This is necessary because process number 1 must be `init`, and it is desirable that all the system threads are process 0.

The final area within the scheduler of processes that does cause problems is the fact the uniprocessor kernel hard codes tests for the idle threads as `task[0]` and the init process as `task[1]`. Because there are multiple idle threads it is necessary to replace these with tests that the process id is 0 and a search for process id 1, respectively.

Memory Management

The memory management core of the existing Linux system functions adequately within the multiprocessor framework providing the locking is used. Certain processor specific areas do need changing, in particular `invalidate()` must invalidate the TLBs of all processors before it returns.

Miscellaneous Functions

The portable SMP code rests on a small set of functions and variables that are provided by the processor specification functionality. These are `smp_processor_id()` which returns the identity of the processor the call is executed upon. This call is assumed to be valid at all times. This may mean additional tests are needed during initialization. The variable `smp_num_cpus` holds the number of processors in the system. The function `smp_message_pass()` passes messages between processors.

7.3.3 Architecture Specific Code for the Intel MP Port

The architecture specific code for the Intel MP Port splits fairly cleanly into three sections. Firstly the initialization code used to boot the system, secondly the message handling and support code, and finally the extension to standard kernel facilities to cope with multiple processors.

Initialization

The Intel MP architecture captures all the processors except for a single processor known as the *boot processor* in the BIOS at boot time. This single processor enters the kernel bootup code. The first processor executes the bootstrap code, load and uncompresses the kernel. Having unpacked the kernel it sets up the paging and control registers then enters the C kernel startup.

Further processors are started up in `smp_boot_cpus()` by programming the APIC controller registers and sending an inter-processor interrupt (IPI) to the processor. This message causes the target processor to begin executing code at the start of any page of memory within the lowest 1MB, in 16-bit real mode. The kernel uses the single page it allocated for each processor to use as a stack. On entering the kernel the processor initializes its trap and interrupt handlers before entering `smp_callin()`, where it reports its status and sets a flag that causes the boot processor to continue and look for further processors. The processor then spins until `smp_commence()` is invoked.

Having started each processor up the `smp_commence()` function flips a flag. Each processor spinning in `smp_callin()` then loads the task register with the task state segment (TSS) of its idle thread as is needed for task switching.

Message Handling and Support Code

The architecture specific code implements the `smp_processor_id()` function by querying the APIC logical identity register. Message passing is accomplished using a pair of IPIs on interrupt 13 (unused by the 80486 FPUs in SMP mode) and interrupt 16. IRQ 13 is a fast IRQ handler that does not obtain the locks, and cannot cause a reschedule, while IRQ 16 is a slow IRQ that must acquire the kernel spin locks and can cause a reschedule. This interrupt is used for passing on slave timer messages from the processor that receives the timer interrupt to the rest of the processors, so that they can reschedule running tasks.

Extension to Standard Facilities

The kernel maintains a set of per processor control information such as the speed of the processor for delay loops. reloads.

The highly useful atomic bit operations are prefixed with the 'lock' prefix in the SMP kernel to maintain their atomic properties when used outside of (and by) the spinlock and message code.

The `/proc` file system support is changed so that the `/proc/cpuinfo` file contains a column for each processor present. This information is extracted from the data structure saved by the function `smp_store_cpu_infor()`.

Part III

TOPSY SMP

8 The Topsy Operating System

Topsy is a small operating system which has been designed for teaching purpose at the Department of Electrical Engineering at ETH Zürich. It constitutes the framework for the practical exercises related to the course Computer Engineering II. This chapter briefly describes the design and implementation of Topsy [FCZP97].

8.1 System Overview

Topsy has a strict modular structure. The kernel contains three main modules reflecting the basic OS tasks; the memory manager, the thread manager and the I/O subsystem. All kernel modules are independently implemented as threads and therefore preemptable (i.e. they can be interrupted like user threads.) The module structure is depicted in Figure 8.1. Big boxes represent the main modules, inner boxes stand for submodules (or subsubmodules). The dotted lines indicate three different layers: the hardware abstraction layer (HAL), the hardware independent kernel components and at the top the modules running in user space.

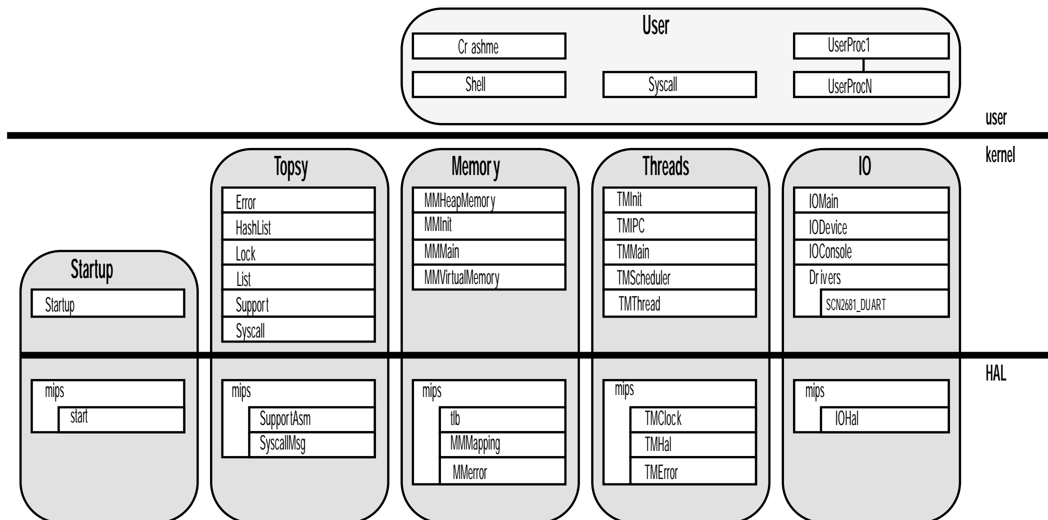


Figure 8.1: Modular structure of Topsy

The hardware abstraction layer is an important element of a portable design. Processor specific parts of the code like exception handling or context switching are embedded in this layer. Consequently, HAL contains a few lines of assembler code. The two upper layers are completely written in C.

8.2 Thread Management

Topsy is a multi-threaded operating system, i.e. multiple threads are able to run quasi-parallel. In Topsy, there exist exactly two processes: the user process and the kernel process (operating system). Threads, however, share their resources, in particular they run in the same address space. Synchronization of shared memory is accomplished via *messages*. The private area of a thread is its stack which is not protected against (faulty) accesses from other threads. However, a simple stack checking mechanism has been incorporated to terminate threads on returning from their main function (stack underflow).

System Calls

In Topsy all system calls are based on the exchange of messages between kernel and user threads. For most of them, a reply is expected from the kernel. Each system call causes a message to be created and sent to the kernel via a software interrupt mechanism. Each system call is recognized via an identifier in the message structure.

IPC

There is a single IPC mechanism in Topsy, namely the sending/receiving of messages. Messages can be sent either between user and kernel threads, between kernel threads, or between user threads. Both kernel and user threads have a unique fixed sized queue for storing incoming messages. A mix of FIFO and priority queuing was implemented; a thread has the possibility to request a particular message of a given type coming from a specific thread.

Scheduler

The Topsy scheduler uses a multilevel queuing system with three levels, corresponding to kernel, user, and idle threads. Within each level, a round-robin policy is used. The highest priority is devoted to kernel threads, i.e. no user thread may be scheduled if there is a single runnable kernel thread. Idle threads have lowest priority. Each thread may be in one of the following states:

- RUNNING: the thread is currently running (at any time, only a single thread may be in the RUNNING state).
- READY: the thread is ready to be scheduled and then dispatched.
- BLOCKED: the thread is blocked, e.g., waiting to receive a message.

Both kernel and user threads are preemptive. A new scheduling decision (schedule and dispatch of a new thread) may be performed in the following cases:

- Time quantum elapsed for the thread (if time-sharing mode).
- An exception is processed in the kernel.
- A running thread is put to sleep waiting for a message.
- After sending a message.

8.3 Memory Management

Topsy divides the memory into two address spaces: one for user threads and the other for the OS kernel (separation of user and kernel process). This has the advantage of better fault recognition facilities, and a stable and consistent behavior of the kernel (user threads are not able to crash the system by modifying kernel memory). The memory is organized in paged manner, i.e. the whole address space is split up into blocks of predefined size. Furthermore, the two address spaces are embedded in one virtual address space, although no swapping of pages to secondary memory is supported (see Figure 8.2).

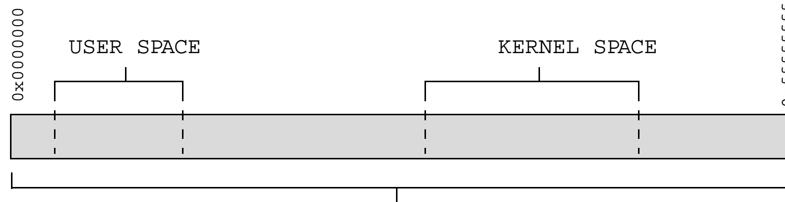


Figure 8.2: User and Kernel Address Space

To understand the mechanism how virtual and physical addresses are connected, let us have a look at Figure 8.3. The picture represents the address mapping supported by the MIPS R3000A processor. The virtual address space consists of four segments: three segments can only be accessed in kernel mode (kseg0, kseg1, kseg2), one is accessible in user and kernel mode (kuseg). The segments kseg0 and kseg1 are directly mapped to the first 512 Mbytes in physical memory. The segments kuseg and kseg2, on the other hand, can be mapped anywhere in the physical address space.

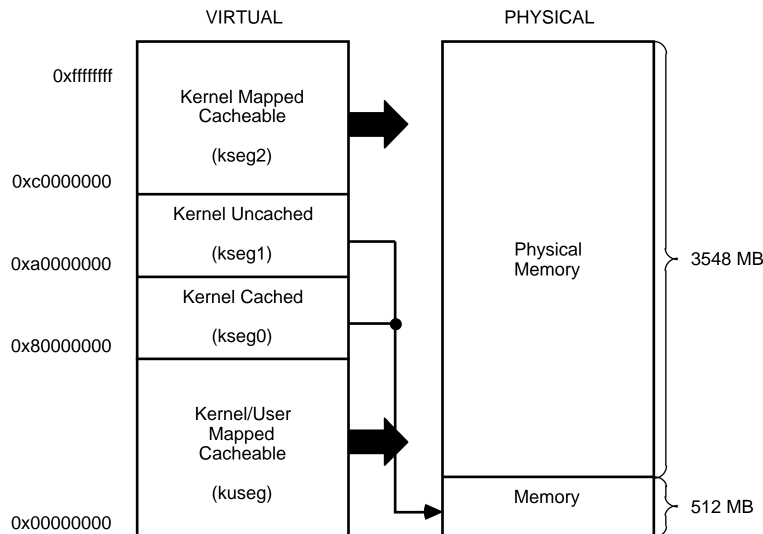


Figure 8.3: Virtual to Physical Address Mapping of the MIPS R3000A

Topsy itself comes with a small footprint. It is able to run with a few 10 Kbytes of memory which is managed in a dynamic fashion. This ensures good utilization of memory. Threads can allocate

memory by reserving a certain, connected piece of the virtual address space called *virtual memory regions*. Every virtual memory region is assigned an appropriate number of physical pages.

The memory manager is responsible to execute the memory demands of kernel or user threads: allocating and deallocating virtual memory regions.

8.4 I/O

The input/output (I/O) subsystem of Topsy is divided into a management part and the actual drivers. These units are running separately as independent threads, i.e. there is one thread (I/O thread) responsible for driver independent functions and further threads (driver threads) implementing the interface to specific hardware devices. User or kernel threads may ask the I/O thread for a certain device to be non-exclusive opened which returns the thread id of the driver thread (if it exists). This thread id allows another thread to communicate with the specific driver.

8.5 User Programs

Topsy comes with a small command line shell enabling the user to start threads, kill threads and get information about threads. The commands available at present are `start`, `exit`, `ps` and `kill`.

9 An SMP Machine for Topsy

This chapter describes what underlying simulation environment was chosen and what consequences this decision had on the existing Topsy kernel.

9.1 Choosing SimOS as a Simulation Environment

Recall from Section 1.1 that one of the goals of this thesis was the development resp. evaluation of a suitable simulation environment. The existing MipsSimulator written in Java provided only a single CPU, therefore a considerable amount of work would have been necessary to change the uniprocessor MipsSimulator into a multiprocessor simulation environment. Besides, the Java interpreter was much too slow even when using a just-in-time compiler. Therefore, the decision was taken to use the SimOS simulation environment from Stanford University[1]. The following list gives an overview over the advantages of the SimOS environment:

- Fast cycle-accurate simulation environment written in C
- Support for the MIPS R4000 processor family
- Build-in multiprocessor support
- Configurable hardware models (CPU, Memory, Bus, Cache)
- Runtime data collection through flexible scripts

Porting Topsy to SimOS required two separate phases. First, the hardware dependent parts of Topsy (which are coded in assembler) had to be ported from the MIPS R3000 (R3k) processor family to the MIPS R4000 (R4k) processor family. Second, the device drivers provided by Topsy had to be adjusted to the device drivers supported by the SimOS environment. The following sections cover the porting phases without mainly focusing on a specific kernel. This should help to use this chapter as an overall guide to port any kernel from MIPS R3k to a combination of R4k and SimOS. However, whenever you see this sign TOPSY at the border, you should be aware that the information you are reading is related especially to the Topsy or TopsySMP kernel.

[1] http://simos.stanford.edu

9.2 Porting Topsy to MIPS R4000

The SimOS environment currently supports the MIPS R4000 and R10000 processor family. Since Topsy was originally written for the MIPS R3000 processor family, the hardware dependent parts of Topsy — mainly the hardware abstraction layer (HAL) — had to be rewritten to operate on the new target processor. Fortunately, the R4k processor family is backward compatible with the R3k architecture and differs only in minor implementation details, namely in memory management and exception handling. In addition to that, the instruction set was subject to a few changes and now offers facilities not available with the R3k. We assume that the reader is familiar with the MIPS R3000 processor family architecture. More information about the MIPS R4000 processor family can be found in Appendix A.

9.2.1 Memory Management

Recall from Appendix A that the MIPS R4000 processor has an on-chip TLB with 48 entries, each of which maps a pair of variable-sized pages ranging from 4 Kbytes to 16 Mbytes, in multiples of four. This differs from the R3000, which had a fixed page size of 4 Kbytes. Therefore, the virtual page number (VPN) and the physical frame number (PFN) are fixed to 20 bit. The R3k TLB consists of 64 entries (each 64 bits wide) to provide a mapping of 64 pages. Figure 9.1 illustrates the format of a R3k TLB entry.

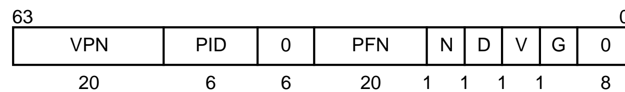


Figure 9.1: Format of a R3k TLB Entry

Like the R4k, the R3k has a set of coprocessor registers to provide the data path for operations which read, write or probe the TLB. The format of these registers is the same as the format of a TLB entry. The obvious differences between the TLBs and the relevant CP0 register sets provided by the two MIPS processor families are summarized in Table 9.1.

	MIPS R3k	**MIPS R4k**
TLB:		
Total number of entries	64	48
Number of fixed entries	8	≤ 48
Size of entry	64 bit	128 bit
CP0 Register Set:		
VPN size	20 bit	19 bit
PFN size	20 bit	24 bit

Table 9.1: Differences between R3k and R4k

From the perspective of an kernel designer this has two immediate consequences: First, the definitions (position, length, meaning) of the CP0 register fields have to be changed. Second, the routines which read, write or probe the TLB file have to be rewritten.

CP0 Register Field Definitions

The definitions of the MIPS R4k CP0 register fields can be found in the file `cpu.h`. Note that the MIPS R4k has additional registers that are not present in the R3k.

TOPSY

TLB Initialization

Recall from Appendix A that each R4k TLB entry maps a pair of pages, rather than a single page. Therefore, the R4k makes distinction between an *EntryLo0* and an *EntryLo1* register, which contain the page frame numbers of pages with even and odd addresses. To adopt a given R3k TLB mapping, the *EntryLo0*, *EntryLo1*, and *EntryHi* registers have to be filled appropriately to build a pair of corresponding pages. To illustrate this, the TLB file from Topsy will be shown next.

Topsy maps the user space by initializing the TLB with a fixed mapping. A contiguous region of 256 kBytes at the beginning (address 0x1000) of the segment kuseg is mapped to physical memory. The reason to start the user address space at 4k and not at address zero is to catch null pointer exceptions. Thus, the TLB entries using a MIPS R3k are shown in Table 9.2 (note that all entries have the appropriate bits *D*, *V*, and *G* set to 1).

TOPSY

Entry	EntryHi	VPN	EntryLo	PFN
0	0x00001000	0x01	0x000c0700	0xc0
1	0x00002000	0x02	0x000c1700	0xc1
2	0x00003000	0x03	0x000c2700	0xc2
3	0x00004000	0x04	0x000c3700	0xc3
⋮	⋮	⋮	⋮	⋮
60	0x0003d000	0x3d	0x000fc700	0xfc
61	0x0003e000	0x3e	0x000fd700	0xfd
62	0x0003f000	0x3f	0x000fe700	0xfe
63	0x00040000	0x40	0x000ff700	0xff

Table 9.2: TLB entries for Topsy on R3k

Translating this to the TLB scheme of the MIPS R4k results in the entries shown in Table 9.3.

Entry	EntryHi	VPN	EntryLo0	PFN	EntryLo1	PFN
0	0x00001000	0x00	0x00000001	0x00	0x0000302f	0xc0
1	0x00003000	0x01	0x0000306f	0xc1	0x000030af	0xc2
⋮	⋮	⋮	⋮	⋮	⋮	⋮
31	0x0003f000	0x1f	0x00003f6f	0xfd	0x00003faf	0xfe
32	0x00041000	0x20	0x00003f3f	0xff	0x00000001	0x00

Table 9.3: TLB entries for Topsy on R4k

Because the user address space of Topsy starts at address 0x1000, the first PFN of entry 0 in Table 9.3 (corresponding to the page with addresses starting at 0x0000) has to be set invalid. The same applies to the second PFN of entry 32. Recall, that Topsy maps a region of 256 kBytes and therefore, the first PFN of entry 32 (R4k) corresponds to entry 63 of the R3k and the second PFN is unused.

TOPSY The initialization of the TLB takes place in the function `mmInitMemoryMapping()` (file `MMDirectMapping.c`). Because of the extended CP0 register set used for TLB operations, the function to write a TLB entry had to be replaced by `setR4kTLBEntry` (file `tlb.S`):

```
void setR4kTLBEntry(Register TLBEntryLow0,
   Register TLBEntryLo1,
   Register TLBEntryHigh,
   Register Index);
```

Additionally, the range of wired TLB entries is set using the function `setTLBWired` (file `tlb.S`):

```
void setTLBWired(Register TLBWired);
```

All the above initialization steps were combined into a function called `mmInitTLB()` (file `MMDirectMapping.c`) which is also called by the additional CPUs during the SMP boot phase.

9.2.2 Exception Handling

Whenever a common exception occurs, the MIPS processor jumps to a predefined address according to the kind of exception. If a Reset happens, it jumps to the address stored in `0xbfc00000`. An UTLBMiss exception causes a branch to the address specified in address `0x80000000`. All other exceptions are treated by one handler (`generalExceptionHandler`). The latter address has changed from the MIPS R3k to the MIPS R4k and is now located at `0x80000180`. The steps performed in `generalExceptionHandler` are:

1. Save registers to be modified by the exception handler.
2. Set stack pointer and frame pointer on the exception stack.
3. Save context of the current thread.
4. Lookup the exception handler table to get the address of the specific exception handler.
5. Call the specific exception handler.
6. Restore context of the current thread.

TOPSY Two files in Topsy are affected by the displacement of the general exception handler base address: `cpu.h` and `TMHal.c`. Latter contains the function `tmInstallExceptionCode` which installs the exception handler in memory.

A basic instruction used in the context switching part of every exception handler routine — the return from exception (`RFE`) instruction — has been replaced in the MIPS R4k instruction set by `ERET`. They slightly differ in that `ERET` doesn't allow any instruction to be placed in its branch delay slot.

TOPSY The only function affected by the `ERET` instruction is `restoreContext` (file `TMHalAsm.S`).

9.2.3 MIPS R4000 Synchronization Primitives

Recall from Chapter 5 that it is essential in an MP system to have a way in which two or more processors working on a common task can execute programs without corrupting the other's subtask. Synchronization, an operation that guarantees an orderly access to shared memory, must be implemented for a properly functioning MP system.

The so called MIPS III[1] *Load Linked* (`LL`) and *Store Conditional* (`SC`) in conjunction with the cache coherency mechanism and protocol, provide synchronization support for R4000 processors. The two instructions work very much like their simple counterpart load and store. The `LL` instruction, in addition to doing a simple load, has the side effect of setting a user transparent bit called the *load link bit (LLbit)*. The LLbit forms a breakable line between the `LL` instruction and a subsequent `SC` instruction. The `SC` performs a simple store if and only if the LLbit is set when the store is executed. If the LLbit is not set, then the store will fail to execute. The success or failure of the `SC` is indicated in the target register of the store after the execution of the instruction. The target register is loaded with 1 in case of a successful store or it is loaded with 0 if the store was unsuccessful. The LLbit is reset upon occurrence of any event that even has potential to modify the lock variable while the sequence of code between `LL` and `SC` is executed. The most obvious case where the link will be broken is when an invalidate occurs to the cache line which was the subject of the load. In this case, some other processor successfully completed a store to that shared line. In general, the link will be broken if following events occur while the sequence of code between `LL` and `SC` is being executed:

1. External update to the cache line containing the lock variable.
2. External invalidate to the cache line containing the lock variable.
3. Intervention or snoop invalidating the cache line containing the lock variable.
4. Upon completion of an `ERET` (return from exception).

The most important features of the `LL` and `SC` primitives are:

- They provide a mechanism for generating all of the common synchronization primitives including test-and-set, counters, sequencers, etc. with no additional overhead.
- They operate in a fashion so that bus traffic is generated only when the state of the cache line changes; locked words stay in the cache until another processor takes ownership of that cache line.

Figure 9.2 shows how the `LL` and `SC` instructions can be used to implement a simple test-and-set function. The flowchart show general methodology and an example of implementation code is listed next to the corresponding flow symbol, with comments next to the code line. The lock variable is located in the cache at address `r1`. If unlocked the least significant bit (LSB) is zero, if locked the LSB is 1.

[1] MIPS III is the instruction set architecture (ISA) implemented in the R4000 processor family. It includes the MIPS I ISA implemented in the R3000.

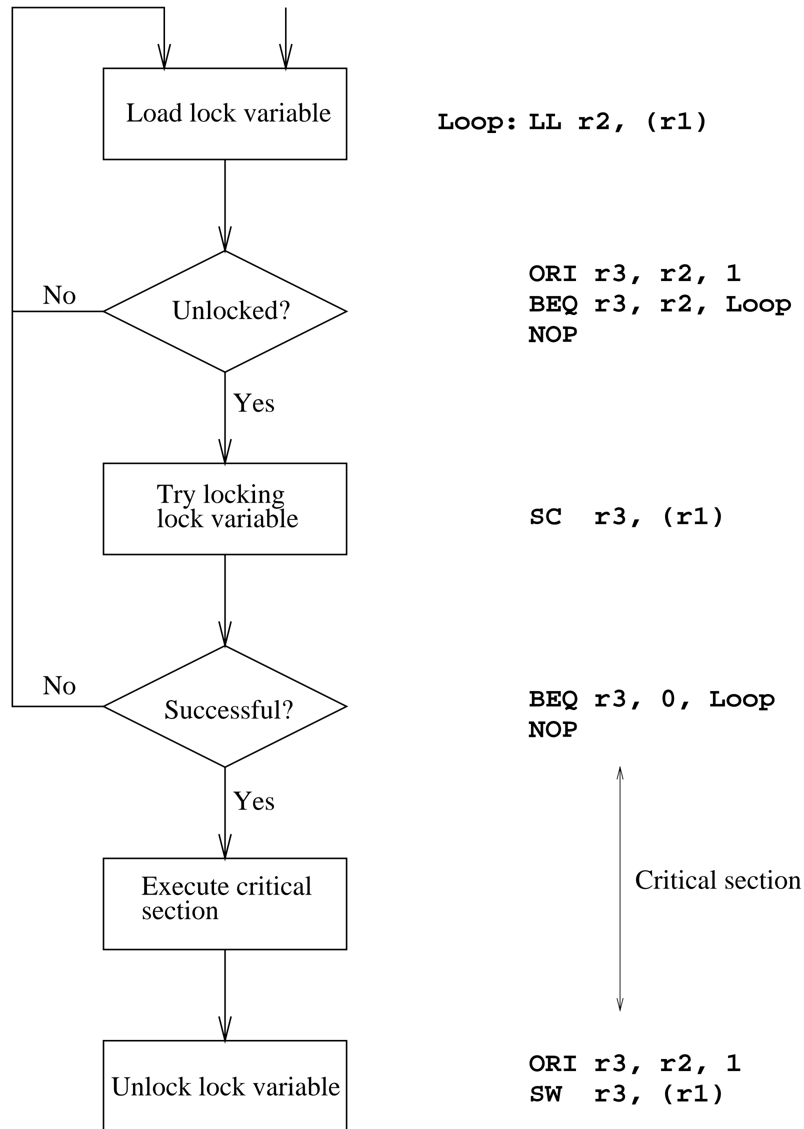


Figure 9.2: Test-and-Set using LL and SC

For this synchronization mechanism to work properly, the cache line which contains the word addresses by (`r1`) in the `LL` instruction, must not be uncached or noncoherent. For additional information regarding `LL` and `SC` instructions, cache-coherency mechanism and protocols, refer to Appendix A or [Hei94].

The TopsySMP lock and unlock functions

TOPSY

In contrast to the test-and-set mechanism shown in Figure 9.2, the locking mechanism used in Topsy and TopsySMP is divided into two separate functions called `lock()` and `unlock()`. The reason for this partitioning is that most of the critical sections in Topsy are written in C rather than Assembler and the distinction makes them more flexible to use.

Figure 9.3 shows the `lock()` function used in Topsy and TopsySMP.

```
void lock(Lock lock)
{
    testAndSet(&(lock->lockVariable));
}
```

Figure 9.3: lock function in Topsy

It uses an atomic test-and-set function based on the code shown in Figure 9.2, with the difference that it only returns on successful locking. The function `testAndSet()` is written in Assembler and uses the R4000 `LL` and `SC` instructions (see Figure 9.4).

```
 1         mfc0    t0, c0_status
 2         and     t0, t0, SR_IE_MASK
 3         mtc0    t0, c0_status
 4 loop:   ll      t1, 0(a0)
 5         ori     t2, t1, 1
 6         beq     t2, t1, loop
 7         nop
 8         sc      t2, 0(a0)
 9         beq     t2, 0, loop
10         nop
```

Figure 9.4: testAndSet function in TopsySMP

The `lock()` function first disables interrupts (lines 1–3) by clearing the IE bit in the status register. It then loads the value of the lock variable (whose address is located in register `a0`). If the lock is held by another processor (`t1` = 1), the processor calling the `lock()` function spins until the lock gets free (lines 4–7). At that moment, it tries to save the value of the lock variable (e.g., locking it) with the `SC` instruction (line 8). The store was successful if and only if the value of the register `t2` is 1. In that case the lock is successfully acquired and the function returns. Otherwise, the processor goes back to the `LL` instruction and tries it again.

A short note on using the GNU C cross-compiler. The `LL`, `SC` and `ERET` instructions are part of the MIPS III instruction set architecture (ISA). In order to produce the correct opcodes, the complier (actually the assembler) has to toggle between MIPS I and MIPS III mode. Thus, whenever a MIPS III instruction is used it has to be embedded with assembler directives.

The following example shows line 4 from Figure 9.4 embedded in assembler directives:

```
loop:
        .set mips3
        ll      t1, 0(a0)
        .set mips0
```

Note that the interrupts remain disabled until the subsequent `unlock()` (see Figure 9.5) reenables them. This is done by setting the IE bit in the status register. The reason for disabling interrupts while the lock is held, is that otherwise an interrupt will further delay other processors waiting for that lock and might result in a deadlock.

```
void unlock(Lock lock)
{
    lock->lockVariable = FALSE;
    enableInterrupts();
}
```

Figure 9.5: unlock function in Topsy

9.3 Porting Device Drivers to SimOS

Two devices are vital to Topsy and most other operating systems: the clock and the console device. The clock device is responsible of initiating clock interrupts in a periodical manner, allowing the implementation of a time-sliced scheduling algorithm. The console device allows text to be written on the screen and keyboard input to be collected. The console device is presented first because of its simplicity. All other devices, including the ones added to SimOS, are similar to the console device, which can therefore be viewed as an example device.

9.3.1 Console Device Driver

SimOS supports a very simple console device with a status and a data register. The definition of the console device can be found in the file `machine_defs.h`:

```
#define DEV_CNSLE_TX_INTR 0x01 /* intr enable / state bits */
#define DEV_CNSLE_RX_INTR 0x02

typedef struct DevConsoleRegisters {
    DevRegister intr_status; /* r: intr state / w: intr enable */
    DevRegister data;        /* r: current char / w: send char */
} DevConsoleRegisters;
```

These registers are mapped into virtual memory space depending on the number of additional devices added to SimOS. Each register has a size of 4 kByte.

In our case they were mapped from address 0xa0e01000 to 0xa0e01008. In order to use the console device SimOS together with the UART model of Topsy, these addresses were added to the files IOHal.h and SCN2681_DUART.h:

TOPSY

```
#define UART_A_BASE 0xa0e01000
#define STATUS_REGISTER 0x0
#define TX_REGISTER 0x4
#define CONS_INT_TX 0x01
#define CONS_INT_RX 0x02
```

In addition to this, all register accesses in the UART routines had to be changed from char (UART) to int (SimOS console).

The appropriate device driver in SimOS is located in the file simmagic.c and consists of a service routine (called a *handler* in SimOS' terminology) used to access the registers, and an interrupt handler routine.

Register Service Routine

The service routine console_handler handles all accesses to the memory-mapped device registers. The major task of the routine is to distinguish which register is about to be accessed and if the operation on the register is a read or a write. The code section shown in Figure 9.6 illustrates this.

```
switch (offs) {
case offsetof(DevConsoleRegisters, intr_status):
   if (BDOOR_IS_LOAD(type)) {
      *datap = (DevRegister) sim_console_int_status(console);
   } else {
      sim_console_int_set(console, (int)*datap);
   }
   break;
case offsetof(DevConsoleRegisters, data):
   if (BDOOR_IS_LOAD(type)) {
      *datap = (DevRegister) sim_console_in(console);
   } else {
      sim_console_out(console, (char)*datap);
   }
   break;
}
```

Figure 9.6: Device Driver Service Routine

The parameter offs specifies the register and the macro BDOOR_IS_LOAD returns true if the operation on the register is a read. In the case of the console device all further actions are handled by internal SimOS functions.

Device Interrupt Handler

The interrupt handler routine console_interrupt first specifies the CPU to which the interrupt is delivered, and then raises the interrupt by calling the SimOS function RaiseSlot with the CPU number and the kind of interrupt as parameter. This function then calls RaiseIBit which sets the appropriate interrupt bit in the status register of the CPU to which the interrupt is delivered.

TOPSY In order not to change the interrupt scheme used by Topsy, the function `RaiseIBit` had to be extended to map the SimOS interrupt ids to the ones used in Topsy as shown in Table 9.4.

Interrupt	Interrupt Id used by SimOS	Interrupt Id used by IDT-Board
Console	0x10	0x05
Clock	0x18	0x00

Table 9.4: Mapping of the Interrupt Ids between SimOS and IDT-Board

9.3.2 Clock Device Driver

Strange enough, SimOS did not deliver clock interrupts by itself. Timers were installed and started using the function `InstallTimers` (file `simmagic.c`), but the correct value for the timer interval was never set! In order to receive timer interrupts, the values had to be set somewhere in the initialization routines of SimOS. We choose to set these values in the function `SimulatorEnter` (file `simmisc.c`) which is called in an early phase of the initialization:

```
SBase[cpu].clockStarted  = 1;
SBase[cpu].clockInterval = 250000;
SBase[cpu].clockTimeLeft = 250000;
```

The values are interpreted as CPU cycles, those 250 000 cycles give a timer frequency of 100 Hz using a clock speed of 25 MHz.

After this, the function `InstallTimers` installed a timer using a so called *event callback handler*, which is triggered by the cycle time of the next timer interrupt event. When the event occurs, the function `TimerCallback` is called which delivers the interrupt and enqueues the callback handler for the next timer interrupt event.

In order to reset a timer interrupt, a special *reset* register was added to the clock device. Therefore, the service routine of the clock handler had to be enlarged to cover access to the reset register, as shown in Figure 9.7.

```
case offsetof(DevClockRegisters, reset):
  if (BDOOR_IS_LOAD(type)) {
    ClearIBit(cpuNum, DEV_IEC_CLOCK);
  } else {
    ASSERT(0); /* for debugging */
    return 1;  /* reset register not writable */
  }
  break;
```

Figure 9.7: Handling Clock Reset

The reset register is a read-only register with the side effect of resetting the current clock interrupt by clearing the appropriate bits in the status register.

The same clock device can be used as a very primitive CMOS clock. The `ctime` register contains the time since January 1, 1970 (same as the Unix `gettimeofday` result).

TOPSY

In Topsy the clock device registers were mapped from address `0xa0e00000` to `0xa0e00008`. In order to use the clock device from SimOS together with the clock model of Topsy, these addresses were added to the file `TMClock.h`:

```
#define TIMERBASE  0xa0e00000
#define COUNTER0   (TIMERBASE+0x0)
#define RESETCINT0 (TIMERBASE+0x4)
```

9.4 Adding additional Devices to SimOS

In order to support symmetric multiprocessing and interprocessor communication in Topsy, a simple device called *InterProcessor Interrupt Controller (IPIC)* has been added to the devices of SimOS. This device can be used as an example to demonstrate how to add additional devices to SimOS, as illustrated by the following sections.

9.4.1 Definition of the Device Register Set

First, the device register set has to be defined in the file `machine_defs.h`. In the case of the IPIC device the following registers were defined:

```
typedef struct DevIpicRegisters {
  DevRegister total_cpus;   /* total number of CPUs */
  DevRegister cpu;          /* hw CPU number */
  DevRegister ipi;          /* inter processor interrupt */
  DevRegister bootaddr;     /* boot address for non-bootup CPUs */
  DevRegister bootstack;    /* boot stack for non-bootup CPUs */
} DevIpicRegisters;
```

The first register (`total_cpus`) provides the total number of CPUs available in the system. The second register (`cpu`) contains the hardware number of the current CPU, ranging from zero to `total_cpus` $- 1$. The third register (`ipi`) is used for interprocessor interrupts and is explained later in this chapter. The fourth register (`bootaddr`) contains the address of the kernel entry point routine called by each non-bootup CPU. The fifth register (`bootstack`) contains the address of the boot stack needed by every non-bootup CPU.

The next thing to do is to reserve a memory region used by SimOS for the simulated devices. We choose an offset between the memory block used by the ethernet device and the one used by the disk device. The macro definition shown in Figure 9.8 is for internal use and needs to know only the name of the register structure and the device offset.

```
#define __MAGIC_BDOOR_IPIC_OFFS 0x00003000 /* ipic controller */
#define DEV_IPIC_REGISTERS(node, nbits)                       \
   ( ((volatile DevIpicRegisters*)                            \
        (__MAGIC_ZONE(node, nbits, MAGIC_ZONE_BDOOR_DEV) + \
         __MAGIC_BDOOR_IPIC_OFFS)) )
```

Figure 9.8: Macro Definition for a SimOS Device

After this is done, we move to file `simmagic.c` and include the necessary lines to add our device to the simulated devices of SimOS:

```
sprintf(name, "IPIC %d", n);
RegistryAddRange((VA)(__MAGIC_ZONE(n, 0, MAGIC_ZONE_BDOOR_DEV) +
                      __MAGIC_BDOOR_IPIC_OFFS), sizeof(DevIpicRegisters),
                      REG_FUNC, (void*)BDOOR_ipic_access, name);
```

and bind it to a service routine:

```
static int
BDOOR_ipic_access(int cpuNum, uint VA, int type, void* buff)
{
  int n    = VA_NODE(VA);
  int offs = VA_OFFS(VA) - __MAGIC_BDOOR_IPIC_OFFS;

  ASSERT(VA_ZONE(VA) == MAGIC_ZONE_BDOOR_DEV);
  ASSERT(0 <= offs && offs < sizeof(DevIpicRegisters));

  return ipic_handler(cpuNum, n, offs, type, buff);
}
```

The service routine has the same structure as the one used for the console and clock devices. Its main task is to distinguish between the different registers and to perform the desired actions. Figure 9.9 shows the section of code which handles the access to the `ipi` register. The remaining registers are read-only or write-only, respectively, and therefore handled similar to the one shown in Figure 9.7.

Figure 9.10 shows the routine `ipic_ipi` called by the handler. The only function currently available is used by the boot CPU to startup the additional CPUs in the system. This function is based on an internal routine provided by SimOS called `LaunchSlave` (file `simmisc.c`).

The only thing left to do now, is to port your specific OS to the new devices. . .

```
static int
ipic_handler(int cpuNum, int n, int offs, int type, void* buff)
{
  DevRegister* datap = (DevRegister*)buff;
  static VA boot_addr, boot_stack;

  ASSERT (BDOOR_SIZE(type) == sizeof(DevRegister));

  switch (offs) {
  ...
  case offsetof(DevIpicRegisters, ipi):
    if (BDOOR_IS_LOAD(type)) {
      ASSERT(0); /* for debugging */
      return 1;  /* ipic register not readable */
    } else {
      ipic_ipi(cpuNum, (int)*datap, boot_addr, boot_stack);
    }
    break;
  ...
  }
  return 0;
}
```

Figure 9.9: Service Routine for the IPIC Device

```
void
ipic_ipi(int cpuNum, int ipi_reg, VA boot_addr, VA boot_stack)
{
  int i;
  int target = (ipi_reg & IPIC_TARGET_MASK);
  int type = (ipi_reg & IPIC_IPITYPE_MASK);

  switch (type) {
  case IPIC_STARTUP:
    if ( target == IPIC_TARGET_ALL ) {
      for (i=0; i<NUM_CPUS(0); i++) {
        LaunchSlave(i, boot_addr, (Reg)boot_stack, 0, 0, 0);
      }
    }
    else {
      LaunchSlave(target, boot_addr, (Reg)boot_stack, 0, 0, 0);
    }
    break;
  default:
      ASSERT(0); /* for debugging */
      return;
  }
}
```

Figure 9.10: Interprocessor Communication using the IPIC Device

10 Design of TopsySMP

This chapter covers the design of an SMP port of the Topsy operating system. It starts with the principal design goals of TopsySMP, followed by a detailed discussion of relevant design issues.

10.1 Principal Design Goals of TopsySMP

The design of a multiprocessor OS is complicated because it must fulfill the following requirements: A multiprocessor OS must be able to support concurrent task execution, it should be able to exploit the power of multiple processors, it should fail gracefully, and it should work correctly despite physical concurrency in the execution of processes. The principal design goals of TopsySMP were defined as follows:

- **Simplicity.** The simple structure of the Topsy OS should not be complicated by an over-sized SMP mechanism.
- **Multithreading.** The multithreaded architecture of Topsy should not be changed.
- **High degree of parallel Kernel Activity.** The kernel should scale well running applications with a realistic job mix.
- **Parallel Thread Scheduling.** Each CPU should run an instance of the thread scheduler.
- **Efficient Synchronization Primitives.** Spin lock times should be reasonably short in order to prevent CPUs from spinning idle for too long.
- **Uniprocessor API.** The system call API of Topsy should not be changed.
- **Scalability.** Scalability means, that additional CPUs can be added to (or removed from) the system without recompiling or even reconfiguring the kernel.

10.2 SMP Operating System Design Issues

This section discuses the principal design goals of TopsySMP as well as additional relevant design issues of a multiprocessor operating system.

10.2.1 Simplicity

The original Topsy was designed for teaching purposes. Therefore, the main goal was to create a small, easy to understand, well structures, but yet realistic system. These ideas did affect the design and implementation of TopsySMP as well. The extension to Topsy should be simple but yet efficient in order to be used as a lecture example of an SMP kernel. The changes to the original code should be minimal while retaining the readability. Therefore, and because of other reasons explained in subsequent sections, TopsySMP was designed primary as an operating system for small-scaled multiprocessor systems with up to eight CPUs with a shared-memory architecture.

10.2.2 Multithreading

The effectiveness of parallel computing depends greatly on the primitives that are used to express and control parallelism within an application. It has been recognized that traditional processes impose too much overhead for context switching. Therefore, threads have been widely utilized in recent systems to run applications concurrently on many processors.

Recall from Chapter 5 that one way to make SMP systems cost effective for highly interactive applications is to allow multiple threads of kernel activity to be in progress at once. This is referred to as a *multithreaded* kernel. To multithread an operating system, all critical regions must be identified and protected in some way.

As the original Topsy is already multithreaded, the current version of TopsySMP adopted the kernel partitioning without changes.

10.2.3 Kernel Design

Recall from Chapter 5 that a master-slave kernel is a poor choice for highly interactive (or otherwise I/O intensive) application environments because of the high system call and I/O activity of these applications. Nevertheless, the uniprocessor Topsy could be easily modified to become a master-slave kernel. Recall, that the slave processor may execute only user code. If we manage to schedule only user threads on the slave processor (or the idle thread if there is nothing to do), were almost done. The priority-based scheduling algorithm of Topsy is of great help to achieve this. Figure 10.1 shows that the inner loop of the scheduler always starts with the kernel priority run queue. If no suitable thread is found, the scheduler moves on to the user priority run queue and finally to the idle priority run queue.

```
for (priority = KERNEL_PRIORITY; priority < NBPRIORITYLEVELS; priority++) {
...
}
```

Figure 10.1: Inner Loop of the Scheduler in Topsy.

If the start value of the loop variable `priority` is set to `USER_PRIORITY` instead, the scheduler only picks user (or idle) threads and thus can be used as a scheduler for a slave processor. In order not to provide two otherwise unchanged versions of the function `schedule()`, a simple test can

be put in front of the for-loop. If we assume that the function `isSlaveCPU()` returns true on the slave processor, then the modified master-slave scheduler could look like the one below:

```
ThreadPriority priority = KERNEL_PRIORITY;
...
if ( isSlaveCPU() ) start_priority = USER_PRIORITY;
for (priority = start_priority; priority < NBPRIORITYLEVELS; priority++) {
...
}
```

However, if the scheduler runs in parallel on both processors, some data structures have to be duplicated as well. As this is true for every kernel with parallel thread scheduling, this topic is discussed in a subsequent section.

Because of the multithreaded structure of the original Topsy kernel, the SMP kernel was chosen to be either a spin-locked or a semaphored kernel.

10.2.4 High degree of parallel Kernel Activity

The amount of concurrent kernel activity that is possible across all the processors is partially determined by the degree of multithreading. Topsy used at least three kernel threads: `tmThread` (thread manager), `mmThread` (memory manager), `ioThread` (I/O manager), an idle thread, and a number of device driver threads. Quasi-parallel requests to the kernel threads get serialized by the system call interface which maps system calls to messages sent to the appropriate kernel module. The kernel threads reside in a loop waiting for a message to arrive. Upon reception, the corresponding system call is performed, a response message is sent back to the originator of the message, and the kernel thread goes back to sleep waiting for another message.

If the design of the kernel modules remain unchanged, the maximum parallelism of kernel activity is limited by the number of kernel threads. Each kernel thread can run simultaneously on a separate CPU, but one specific thread can only run on a single CPU at a time. Therefore, two system calls provided by the same kernel module can never run in parallel, even if they operate of distinct data structures.

In order to support simultaneous system calls from different threads running on different processors, the design of the kernel modules has to be changed. One possible solution would be to duplicate the control thread of the kernel module, allowing two system calls (provided by the duplicated module) to be handled simultaneously. As a result, the interface of these system calls has to be changed, otherwise all messages would be sent to the original control thread. The system call interface should distribute the messages fairly upon the two message queues, e.g., using a round-robin policy.

The distribution of the messages among the multiple control threads could be performed at three different places:

- Inside the system call library,
- by one of the control threads, or
- inside the kernel message dispatching routine.

Figure 10.2 shows a single function (`vmAlloc`) of the system call library. We assume that the thread ids of the two control threads are `MMTHREADID0` and `MMTHREADID1` respectively. The receiver of the message is swapped right before the message is sent, resulting in a round-robin policy. The drawback of this simple approach is that the necessary changes affect every single system call, and that the round-robin policy is on a per-system-call basis rather than affecting all system calls.

```
SyscallError vmAlloc(Address *addressPtr, unsigned long int size)
{
    Message message, reply;
    static ThreadId receiver = MMTHREADID0;

    message.id = VM_ALLOC;
    message.msg.vmAlloc.size = size;
    reply.id = VM_ALLOCREPLY;
    receiver = ( receiver == MMTHREADID0 ) ? MMTHREADID1 : MMTHREADID0;

    if (genericSyscall(receiver, &message, &reply) == TM_MSGSENDFAILED)
        return VM_ALLOCFAILED;

    *addressPtr = reply.msg.vmAllocReply.address;
    return reply.msg.vmAllocReply.errorCode;
}
```

Figure 10.2: Modified System Call Interface.

Because the two message queues are distinct, they do not have be protected by synchronization primitives. However, all data structures (belonging to the control thread) that can be accessed simultaneously have to be protected in order to guarantee system integrity.

A better solution would be to handle the message distribution inside a special server (e.g. one of the control threads), or inside the kernel message dispatching routine. The modifications would be restricted to one place in the kernel and the dispatching policy could affect all system calls.

Another solution to the problem of message distribution is the use of an *anycast* protocol. Anycast refers to communication between a single sender and the nearest of several receivers in a group. The term “nearest” can be defined in a wider scope as the first receiver to respond or the first receiver actually getting the message. If we consider the control threads of our example to be members of a group representing the kernel module, anycast means, that the system call interface would address the group rather than an individual thread. The first thread to whom the message could be delivered will perform the system call. Anycast requires group addressing which could be easily added to the Topsy kernel.

Section 11.1.9 shows a possible implementation of a duplicated kernel module control thread.

10.2.5 Parallel Thread Scheduling

To ensure the efficient use of its hardware, a multiprocessor OS must be able to utilize the processors effectively in executing the tasks.

In order to provide a parallel thread scheduling, all critical regions of the scheduler must be identified and protected in some way. Since the scheduler manipulates the priority queues and the scheduler specific data of the thread, these two structures need further considerations. The priority queues are global to all schedulers and accessed and modified simultaneously. Thus, they have to be protected in order to guarantee system integrity. The scheduler specific data (status of the thread) is private to each thread and therefore not subject to race conditions. The locking strategy for the scheduler can either consist of a single lock protecting the complete scheduling algorithm or a set of locks protecting the priority queues.

Since the uniprocessor kernel only allows one running thread at any time, some data structures, representing the context of the running thread, have to be replicated. This is subject to a discussion in Section 11.1.2.

A multiprocessor system must be able to degrade gracefully in the event of failure. Thus, a multiprocessor OS must provide reconfiguration schemes to restructure the system in case of failures to ensure graceful degradation. The parallel execution of the thread scheduler provides a degree of fault tolerance as well. Consider a system with a central thread scheduler for all processors. If this CPU fails, the entire system fails, because no more thread scheduling takes place. Contrarily, a system with parallel thread scheduling would likely continue to work with a reduced number of CPUs.

10.2.6 Efficient Synchronization Primitives

In a multiprocessor operating system, disabling interrupts is not sufficient to synchronize concurrent access to shared data. A more elaborate mechanism that is based on shared variables is needed. Moreover, a synchronization mechanism must be carefully designed so that it is efficient, otherwise, it could result in significant performance penalty.

Short critical sections can be protected by spin locks. They should not be used as a long-term mutual exclusion technique, because processors waiting for the lock do not perform any useful work while spinning. Overall system performance will be lowered if the processors spend too much time waiting to acquire locks. This can also happen if too many processors frequently contend for the same lock.

Topsy uses three spin locks for the entire kernel: `hmLock`, `threadLock`, and `schedLock`. The following list shows all modules and functions with critical sections:

- `MMHeapMemory.c`
  - `hmAlloc()`
  - `hmFree()`
- `TMThread.c`
  - `threadStart()`

  - `threadDestroy()`
- `TMScheduler.c`
  - `schedulerInsert()`
  - `schedulerRemove()`
  - `schedulerSetReady()`
  - `schedulerSetBlocked()`
  - `schedule()`

The first two modules only use their corresponding spin lock, i.e., `hmAlloc()` and `hmFree()` use `hmLock`, and `threadStart()` and `threadDestroy()` use `threadLock` respectively. The third module is different, because the function `schedule()` uses not only its corresponding spin lock `schedLock` but also `threadLock`. This is important because the scheduler may be called in a clock interrupt and interfere with the thread manager previously run. By checking the `threadLock`, the scheduler makes sure that the thread manager was not holding the lock.

If we build a matrix showing which spin lock is used in which kernel thread, we get the following picture:

	hmLock	**threadLock**	**schedLock**
tmThread	•	•	
mmThread	•		
ioThread	•		
Drivers	•	(•)	
Exception Context		•	•

From this point of view, the `hmLock` should be object to the highest lock contention, although the lock only protects two system calls. However, this highly depends on the workload.

For a complete analysis of the critical sections of Topsy, we have to take a look at the global data structures protected by the locks. The following list shows all global data structures within the three modules listed above:

- `MMHeapMemory.c`
  - `HmEntry start;`
  - `HmEntry end;`
- `TMThread.c`
  - `HashList threadHashList;`
  - `List threadList;`
- `TMScheduler.c`
  - `Scheduler scheduler;`

The two pointers to the start and end of the head memory list are only accessed through the functions `hmAlloc()` and `hmFree()`. As both functions acquire the heap memory lock before modifying the memory list (and releasing it afterwards), no further protection is necessary for the SMP version of the kernel.

The `threadHashList` is accessed not only in `TMThread.c` but also in `TMIPC.c`, which handles all interprocess (resp. inter-thread) communication. The same is true for the `threadList`. Because all Topsy system calls are mapped to messages, these two lists are accessed during each system call. An SMP kernel has to make appropriate measures to protect these data structures in order to allow multiple messages (from multiple threads running on multiple processors) to be dispatched simultaneously.

The `scheduler` data structure is accessed only within the module `TMScheduler.c`. Put since the scheduler of our SMP kernel should run in parallel on every CPU, the data structure is accessed in parallel and has to be protected from corruption.

10.2.7 Uniprocessor API

Recall from Section 3.2 that a shared-memory architecture has the advantage of a simple programming model, which is an extension of the uniprocessor model. In this model, the data is directly accessible to every processor, and explicit communication code in only needed to coordinate access to shared variables. Therefore, no additional system calls are needed to write multithreaded programs runnable on a shared-memory architecture. The uniprocessor system call API of Topsy can be adopted without changes.

As a side effect, all programs written for the original (uniprocessor) Topsy will run under TopsySMP without modification. The additional CPUs are totally transparent to kernel and user threads.

However, one system call should be added, in order to provide CPU-pinning of threads. CPU-pinning means, that threads can be assigned to certain CPUs and are not allowed to run on other CPUs. This is very useful for device drivers which normally receive interrupts only from the CPU to which the device is attached.

10.2.8 Scalability

SimOS provides an elegant way to configure the simulated hardware, including the total number of available CPUs. This is done by modifying an ASCII file which is processes whenever the SimOS environment is started. In order to run the same multiprocessor kernel on top of a variety of hardware configurations, the kernel should not have to be rebuild if the number of CPUs changes.

There are two ways to achieve this: First, the kernel could be statically configured so that the actual number of CPUs must be less or equal to a given maximum number of CPUs. Second, the kernel could be configured dynamically to the effective number of available CPUs. In the first case, memory always has to be allocated for the maximum number of CPUs, even if they are not available. In the second case, the kernel provides memory only for those CPUs actually present in the system.

Therefore, all kernel resources needed by each CPU should be allocated dynamically from the kernel heap memory rather than statically. This limits the maximum number of processors since the original

memory layout of Topsy was preserved. However, the great advantage of dynamic memory allocation is that the kernel remains small in size even for a multiprocessor system with up to 16 processors.

11 Implementation of TopsySMP

This chapter covers the implementation of an SMP port of the Topsy operating system discussed in the previous chapter.

11.1 Implementation Steps

The following steps were performed in order to make Topsy runnable on an SMP system:

1. Make the SMP system configuration available to the kernel,
2. Replicate the essential data structures for additional CPUs,
3. Adapt the bootstrapping phase,
4. Adapt the scheduler and scheduling algorithm,
5. Adapt the exception handler routines.

The following sections give a detailed overview over the implementation phases.

11.1.1 SMP System Configuration

The hardware configuration of the SMP system can be determined by the kernel by reading a special memory-mapped device similar to the combination of BIOS and APIC used in Intel's MP specification. This device called IPIC has been added to SimOS and is described in Section 9.4. Beside the total number of available CPUs, the hardware number of each CPU can be read.

The total number of available CPUs in the system can be accessed within the kernel through the global (constant) variable `smp_ncpus`, which is set during the bootstrapping phase. The hardware number of each CPU can be accessed through the function `smpGetCurrentCPU()`. They all can be found in the file `SMP.c` and are declared in the corresponding header file `SMP.h`:

```
unsigned int smp_ncpus;
unsigned int smpGetCurrentCPU();
```

In addition to these SMP-specific functions, the content of the processor revision identifier register (PRId) can be read through the function `getPRID` which is located in the file `SupportAsm.S`. The structure of the PRId register is described in Section A.3.3 of Appendix A.

```
Register getPRID();
```

11.1.2 Data Structures

Running Thread

The uniprocessor version of Topsy contains a data structure which represent the currently running thread. In a multiprocessor environment with *n* processors, these data structure has to be duplicated by the number of processors. Thus, the pointer to the data structure of the currently running thread `Scheduler.running` now becomes an array of pointers, indexed by the hardware number of the CPU. To access the currently running thread, the function `smpGetCurrentCPU()` can be used as shown in the SMP implementation of the function `schedulerRunning`:

```
/* return the current thread */
Thread* schedulerRunning()
{
    return scheduler.running[smpGetCurrentCPU()];
}
```

Idle Thread

If the scheduler of a uniprocessor kernel finds no ready-to-run threads, he spins idle in the context of the last thread running until another thread becomes ready. Unfortunately, this is not possible in a multiprocessor kernel, because the kernel runs simultaneously on every CPU. A spinning scheduler would prevent every other CPU from selecting a ready-to-run thread. Therefore, a special *idle thread* is introduced, which is always ready-to-run and can be scheduled like any other thread. Thus, the scheduler is guaranteed to always find a suitable thread.

TopsySMP creates an idle thread for every CPU in the system. Idle threads are user threads running at lowest priority, consisting of a infinite loop. Figure 11.1 show how the idle threads are created and started in the function `tmMain` (file `TMMain.c`):

```
/* Starting idle thread(s) */
for (i=0; i<smp_ncpus; i++) {
   if ((idleThreadId[i] = threadStart((ThreadMainFunction)tmIdleMain,
                      NULL, KERNEL, "idleThread", TMTHREADID, NO_CPU, TRUE))
                      == TM_THREADSTARTFAILED) {
      PANIC("idleThread␣could␣not␣be␣started");
    }
}
```

Figure 11.1: Starting idle thread(s) in tmMain().

Exception Context

On occurrence of an exception, the kernel has to save the context of the currently running thread before calling the exception handling routine. Topsy uses an exception context data structure (`exceptionContext`) to pass parameters to the appropriate handlers. In TopsySMP, this data structure becomes an array indexed by the hardware number of the CPU on which the exception occurs. The memory for the exception context is allocated dynamically in function `tmInit()` (file `TMInit.c`):

```
extern Register *exceptionContext;

/* Allocate memory for the exception context */
if (hmAlloc((Address*)&exceptionContext, smp_ncpus*4*sizeof(Register))
            == HM_ALLOCFAILED) {
   PANIC("couldn't create exception context");
}
```

Boot Stack

During the bootstrapping phase every CPU uses a distinct boot stack, allowing them to boot simultaneously without mutual interference. After booting, the boot stack is used as an exception stack needed during exception handling.

```
/** boot/exception stack(s) */
bsbottom = BOOTSTACKBOTTOM;
bstop = BOOTSTACKTOP;
for (i=0; i<smp_ncpus; i++) {
    vmInitRegion(space->regionList, (Address)bsbottom, BOOTSTACKSIZE,
                 VM_ALLOCATED, READ_WRITE_REGION, 0);
    bsbottom = bstop;
    bstop = (bsbottom+BOOTSTACKSIZE-4);
}
```

All SMP data structures are allocated dynamically rather that statically. This has the advantage of not wasting memory for CPUs not available to the system. Furthermore, no upper limit is coded into the kernel, which leads to a scalable system. The kernel has not to be reconfigured or recompiled if additional CPUs are added.

11.1.3 Bootstrapping

While all processors in an SMP system are functionally identical, there classified into two types: the boot processor (BP) and the non-boot processors (NBP). Which processor is the BP is determined by the hardware. This differentiation is for convenience and is in effect only during the initialization process. The BP is responsible for initializing the system and booting the operating system; NBPs are activated only after the operating system is up and running. CPU0 is designated as the BP. CPU1, CPU2, and so on, are designated as the NBPs.

The bootstrapping phase of the BP is identical to the bootstrapping phase of a single processor in the conventional Topsy OS, with one exception: the BP is responsible for initializing the additional NBPs. The SimOS environment initializes all processors in the system to their reset state, meaning that the registers are set to a default value. The NBPs are then halted and the BP starts its bootstrapping phase by loading the kernel into memory. After initializing the kernel subsystems (thread management, memory management and I/O devices) the BP starts the additional NBPs, by calling `smpBootCPUs` (see Figure 11.2) in the file `TMMain.c` after having started all idle threads.

This function calculates the address of the boot stack for each NBP and starts them through the SimOS IPIC device. The address of the initial startup routine (`SMP_BOOT_ADDR`, defined in `SMP.h`) is passed to the IPIC device as an additional parameter.

```
void smpBootCPUs()
{
  int i, cmd, bootstacktop, bootstackbottom;
  int *boot_addr = (int*)((unsigned long)(IPIC_BASE) + BA_REGISTER);
  int *boot_stack = (int*)((unsigned long)(IPIC_BASE) + BS_REGISTER);
  int *ipi_reg = (int*)((unsigned long)(IPIC_BASE) + IPI_REGISTER);

  bootstackbottom = BOOTSTACKTOP;
  for (i=1; i<smp_ncpus; i++) {
    *boot_addr = (int)SMP_BOOT_ADDR;
    bootstacktop = (bootstackbottom+BOOTSTACKSIZE-4);
    bootstackbottom = bootstacktop;
    *boot_stack = bootstacktop;
    cmd = i | IPIC_STARTUP;
    *ipi_reg = cmd;
    delayAtLeastCycles(1000);
  }
}
```

Figure 11.2: Function smpBootCPUs.

Upon startup, NBPs are calling the initial startup routine `__startSMP` (file `start.S`) which takes the address of the boot stack as a parameter and initializes stack and frame pointers:

```
la      gp, 0x80000000      /*_gp, value of global pointer */
move    sp, a0              /* prepare bootstack */
move    fp, sp
subu    sp, sp, 32
sw      ra, 28(sp)
sw      fp, 24(sp)
mtc0    zero, c0_status     /* disable interrupts, kernel mode */
jal     smpStartupCPU
nop
```

This function then calls the kernel entry point `smpStartupCPU()` (file `SMP.c`):

```
void smpStartupCPU()
{
  mmInitTLB();          /* init TLB */
  scheduleIdle(CPU);    /* first thread is idleThread */

  /* Restoring context of first thread to be activated */
  restoreContext(scheduler.running[CPU]->contextPtr);
}
```

This function performs the processor specific initialization needed by every CPU. This includes initializing the TLB and calling the scheduler, which picks an idle thread. This is necessary because the BP first has to schedule the memory and io thread, which have to do basic initialization work before the system is fully functional. The NBPs are therefore forced to schedule an idle thread first. In order not to complicate the primary scheduler, a separate scheduler-like function called `scheduleIdle` has been added to file `TMScheduler.c`, which is called only once by every

NBP after initializing the TLB. After this, each NBP restores the context of his idle thread and keeps spinning in the idle loop until the first thread becomes ready to run.

In the meantime, the BP has scheduled all kernel threads, initialized all kernel data structures, and started the user shell.

11.1.4 Scheduler

Recall from previous sections that the scheduler in TopsySMP runs on every CPU. It searches the ready queues from higher to lower priority to find the next thread that is allowed to run. In a multiprocessor environment with *n* processors there are always *n* threads running concurrently, even if all those threads are idle threads. The scheduler therefore has to search the entire ready queue instead of picking the first one as in the uniprocessor version of Topsy. Furthermore, TopsySMP introduces the possibility to pin certain threads to a specific CPU, e.g. device drivers which deliver interrupts only to one CPU. Therefore, the scheduler has to check if a CPU is allowed to run a certain thread. The inner loop of the scheduler is shown in Figure 11.3.

```
for (priority = KERNEL_PRIORITY; priority < NBPRIORITYLEVELS; priority++) {
   listGetFirst(scheduler.prioList[priority].ready, (void**)&newRunning);
   while ( newRunning != NULL ) {
      if ( newRunning->schedInfo.status == READY ) {
         if ( (newRunning->schedInfo.pinnedTo == NO_CPU) ||
              (newRunning->schedInfo.pinnedTo == cpu) ) {
            newRunning->schedInfo.status = RUNNING;
            newRunning->schedInfo.cpu = cpu;
            ipcResetPendingFlag(newRunning);
            scheduler.running[cpu] = newRunning;
            break;
         }
      }
      listGetNext(scheduler.prioList[priority].ready, (void**)&newRunning);
   }
   if ( newRunning != NULL ) break;
}
```

Figure 11.3: Inner Scheduler Loop of TopsySMP

The inner loop starts by picking the first thread from the kernel priority ready list. After checking that the thread is indeed ready — it could be running on an other CPU — the scheduler inspects the `schedInfo` structure of the thread to see if this thread is pinned to a specific CPU. Therefore, the `schedInfo` structure in TopsySMP was extended by a field called `pinnedTo` which either contains the number of a CPU or the value `UNBOUND`. In the latter case, the thread can run on any CPU.

If the thread is pinned to the CPU on which the scheduler is currently running, or the thread is allowed run on any CPU, it is chosen to be the next thread to run and the inner loop is left. Otherwise, the next thread on the current ready queue is picked. If the end of the ready queue was reached, the next lower priority ready queue is selected and the loop starts again. Remember that the lowest priority queue holds the idle threads which are always ready to run.

The locking strategy of the uniprocessor scheduler had to be changed in order to guarantee data consistency in a multiprocessor environment. The scheduler in TopsySMP is protected by a spin lock to grant mutual exclusion.

11.1.5 System call API

In order to allow threads to be pinned to specific CPUs the two system calls `threadStart()` and `threadBuild()` (file `TMThread.c`) have been supplied with an additional parameter called `pinnedTo`. The parameter either holds the hardware number of a specific CPU or the value `UNPINNED`.

User programs can use the new system call `tmStartBound()` to pin user threads to a specific CPU:

```
SyscallError tmStartBound( ThreadId* id,
      ThreadMainFunction function,
      ThreadArg parameter,
      char *name,
      CPUId pinnedTo);
```

In order to implement the new system call, the `tmStart` message was extended to hold the additional parameter. The `tmStart()` system call, which uses the same message structure, sets this value to `UNPINNED`.

11.1.6 User Thread Exit

User threads in Topsy exit by either explicitly calling `tmExit()` as the last statement in a user program or by a implicit call to `automaticThreadExit()`. These two function send an exit message to the thread manager which then removes the thread from the run queues and frees all memory hold by the thread.

This thread will never run again, although (theoretically speaking) he would be able to do so unless the thread manager has finished handling the `tmExit()` system call. In a uniprocessor environment this could never happen because after sending the exit message the user thread will be preempted. The scheduler will probably pick the thread manager sometimes but certainly before the user thread (which is still in the ready queue). This can be guaranteed by two things: First, the thread manager has higher priority than any user thread, and second, threads are scheduled in a round-robin manner.

However, in a multiprocessor environment with n simultaneous threads running at a time, some measures have to be taken to prevent the user thread from being scheduled on any CPU before the thread manager is scheduled. Therefore, the handling of an exit message had to be changed in the following manner: The function `threadDestory()` (files `TMThread.c`), which is called by `threadExit()`, first tries to remove the thread from the ready queue by calling `schedulerRemove()` in Figure 11.4.

If the status of the thread is blocked or ready (i.e. not running), the thread is simply removed and the function returns `TM_OK`. Otherwise (i.e. the thread is running on a different CPU), the status of the

```
Error schedulerRemove( Thread* threadPtr) {
    ThreadPriority priority = threadPtr->schedInfo.priority;

    lock(schedLock); {
        if (threadPtr->schedInfo.status == BLOCKED) {
            listRemove(scheduler.prioList[priority].blocked, threadPtr,
                        threadPtr->schedInfo.hint);
        } else if ((threadPtr->schedInfo.status == READY) ||
                   (threadPtr->schedInfo.status == EXIT)) {
            listRemove(scheduler.prioList[priority].ready, threadPtr,
                       threadPtr->schedInfo.hint);
        } else {
            if (threadPtr->schedInfo.cpu != CPU ) {
                threadPtr->schedInfo.status = EXIT;
                unlock(schedLock);
                return TM_FAILED;
            }
        }
    }
    unlock(schedLock);
    return TM_OK;
}
```

Figure 11.4: Function schedulerRemove

thread is changed to `EXIT` and the function returns `TM_FAILED`. Back in `threadDestory()`, the return value decides whether the removal of the thread can continued or not. In the latter case, the function exits immediately and the thread keeps running until the next timer interrupt. The scheduler then recognizes the special exit status and prevents the thread from being scheduled again. The clock interrupt handler periodically calls the function `schedulerRemoveExit()` to remove all user threads with status `EXIT` (see Figure 11.5).

```
void schedulerRemoveExit() {
    Thread* t = NULL;
    listGetFirst(scheduler.prioList[USER_PRIORITY].ready, (void**)&t);
    while ( t != NULL ) {
        if ( (t->schedInfo.status == EXIT) && (t->schedInfo.cpu = NO_CPU) ) {
           threadExit(t->id);
        }
        listGetNext(scheduler.prioList[USER_PRIORITY].ready, (void**)&t);
    }
}
```

Figure 11.5: Function schedulerRemoveExit

This function can be optimized by a global counter variable holding the number of threads with status `EXIT`. This counter is incremented in `schedulerRemove()` whenever a thread changes his state to `EXIT`. The clock interrupt handler tests this counter and calls `schedulerRemoveExit()` only if the value is greater than zero.

Recall from above that all threads currently not running can be safely remove from the ready queue.

This includes all threads with status EXIT. After this, the allocated memory held by the thread can be freed and the global counter is decremented.

11.1.7 Exception Handler

Recall from Section 11.1.2 that TopsySMP allocates an exception context for every CPU in the system. In order to be able to handle exceptions concurrently, each handler routine has to select the appropriate exception context of the CPU it is running on. This is done throughout all handler routines by the lines of code shown in Figure 11.6.

```
    la      k0, exceptionContext      /* exceptionContext -> k0 */
    la      k1, IPIC_BASE             /* ipic register base -> k0 */
    lw      k1, CPU_OFFSET(k1)        /* cpu number -> k0 */
    beq     k1, 0, no_adjust
    nop
    sll     k1, k1, 4
    addu    k0, k0, k1
no_adjust:
```

Figure 11.6: Exception Handler Code

First, general register k0 is loaded with the start address of the exception context array. Then, k1 is loaded with the hardware number of the CPU. If this number is zero (i.e. the CPU is the boot CPU), k0 already points to the right array entry and the handler can continue with the instruction at label no_adjust. Otherwise, the CPU number is shifted left by four (i.e. multiplied by 16) and added to k0 to index the corresponding array entry. (Recall, that the exception context consists of four 4-byte registers, making a total of 16 bytes.)

Two more SMP-specific things have to be done by an exception handler. First, the stack pointer has to be loaded with the address of the exception stack belonging to the specific CPU. Second, the context of the running thread has to be saved first and restored later. This is done by calling the function saveContext and restoreContext respectively, which takes a pointer to the thread context as argument. Both, the address of the exception stack and the pointer to the thread context are CPU-specific and can only be determined through the CPU number. The code is similar to the one above and therefore not presented here.

11.1.8 Synchronization Primitives

The main synchronization primitive in TopsySMP is the spin lock. Critical sections can be protected by enclosing then with the function pair lock and unlock. They are based upon the MP-safe test-and-set function presented in Section 9.2.3. Beside the simple spin lock, two more powerful synchronization primitives were implemented in TopsySMP: *semaphores* and *reader-writer locks*.

Semaphores

Recall from Section 5.3 that a semaphore can implement either mutual exclusion or process synchronization and works correctly for any number of processors in the system, including the uniprocessor case.

Each semaphore requires a small data structure to maintain the current value and the queue of blocked threads. A single linked list will be used for the queue. A thread that blocks on a semaphore is added to the tail of the list, and threads unblocked by a *V* operation are removed from the head. A spin lock is added to provide mutual exclusion while the data structure is being updated.

```
typedef struct SemDesc_t {
    LockDesc lock;
    int count;
    List queue;
} SemDesc;
typedef SemDesc* Semaphore;
```

The semaphore data structure must be initialized before it is used. A semaphore can be initialized to any given value with the function `semInit` shown in Figure 11.7.

```
void semInit(Semaphore sem, int initial_cnt)
{
    lockInit(&sem->lock);
    sem->queue = listNew();
    sem->count = initial_cnt;
}
```

Figure 11.7: Function semInit

The semaphore *P* operation can be implemented as shown in Figure 11.8.

```
void semP(Semaphore sem)
{
    int CPU = smpGetCurrentCPU();
    lock(&sem->lock);
    sem->count--;

    if ( sem->count < 0 ) {
      listAddAtEnd(sem->queue, scheduler.running[CPU], NULL);
      schedulerSetBlocked(scheduler.running[CPU]);
      unlock(&sem->lock);
      tmYield();
      return;
    }
    unlock(&sem->lock);
}
```

Figure 11.8: Function semP

The semaphore *V* operation can be implemented as shown in Figure 11.9. If a thread had blocked during a previous *P* operation, the *V* operation removes the oldest thread from the queue and sets the thread ready. As such, this implementation favors overall fairness by awakening processes in FIFO order.

Multireader Locks

A multi-reader, single-writer lock (or simply a multireader lock for short) allows multiple processes

```
void semV(Semaphore sem)
{
    Thread* threadPtr = NULL;

    lock(&sem->lock);
    sem->count++;

    if ( sem->count <= 0 ) {
      listGetFirst(sem->queue, (void**)threadPtr);
      listRemove(sem->queue, threadPtr, NULL);
      unlock(&sem->lock);
      schedulerSetReady(threadPtr);
      return;
    }
    unlock(&sem->lock);
}
```

Figure 11.9: Function semV

to access a shared data structure at once, as long as none of them needs to modify it. Writers are granted mutually exclusive access so that the integrity of the data structures is maintained. Multi-reader locks can be easily implemented with semaphores as follows.

The data structure shown next will be used to record the state of a multireader lock. A spin lock is used to protect the counter fields. The data structure keeps track of both the number of threads currently in the critical section, as well as the number of threads waiting to enter. These counts are divided between readers and writers.

```
typedef struct MRLockDesc_t {
    LockDesc lock;
    int rdcnt;          /* # of readers in critical section */
    int wrcnt;          /* # of writers in critical section */
    int rdwcnt;         /* # of waiting readers */
    int wrwcnt;         /* # of waiting writers */
    SemDesc rdwait;     /* sync semaphore where readers wait */
    SemDesc wrwait;     /* sync semaphore where writers wait */
} MRLockDesc;

typedef MRLockDesc* MRLock;
```

Before using the lock, it must be initialized by calling the routine shown in Figure 11.10.

```
void mrlockInit(MRLock lock)
{
    lockInit(&lock->lock);
    initSem(&lock->rdwait, 0);
    initSem(&lock->wrwait, 0);
    lock->rdcnt = 0;
    lock->wrcnt = 0;
    lock->rdwcnt = 0;
    lock->wrwcnt = 0;
}
```

Figure 11.10: Function mrlockInit

The strategy for controlling access to the critical resource protected by the multireader lock is to allow readers to enter the critical section at any time, as long as no writers are waiting or currently in the critical section. Once a writer arrives, subsequent readers are blocked. This ensures that a continuous stream of new readers arriving at the lock does not starve out the writers forever. Threads that whish to acquire the multireader lock for reading use the routine shown in Figure 11.11.

```
void mrEnterReader(MRLock lock)
{
    lock(&lock->lock);

    /*
     * if a writer has the lock presently or there are
     * writers waiting, then we have to wait.
     */
    if ( lock->wrcnt || lock->wrwcnt ) {
      lock->rdwcnt++;
      unlock(&lock->lock);
      semP(&lock->rdwait);
      return;
    }
    lock->rdcnt++;
    unlock(&lock->lock);
}
```

Figure 11.11: Function mrEnterReader

A reader leaving a critical section protected by a multireader lock calls the routine shown in Figure 11.12. Once all the readers have left the critical section, a single writer is awakened, if any is waiting.

When a writer wishes to acquire the lock, it must wait for all processes using the lock to leave the critical section. If no processes are currently using it, the writer can acquire the lock immediately. Note the `wrcnt` field can never be greater than 1 by definition of the multireader, *single*-writer lock. (see Figure 11.13)

```
void mrExitReader(MRLock lock)
{
    lock(&lock->lock);
    lock->rdcnt--;

    /*
     * if we're the last reader, and a writer is waiting,
     * then let the writer go now.
     */
    if ( lock->wrwcnt && (lock->rdcnt == 0) ) {
      lock->wrcnt = 1;
      lock->wrwcnt--;
      unlock(&lock->lock);
      semV(&lock->wrwait);
      return;
    }
    unlock(&lock->lock);
}
```

Figure 11.12: Function mrExitReader

```
void mrEnterWriter(MRLock lock)
{
    lock(&lock->lock);

    /*
     * block if any threads are already using the lock.
     */
    if ( lock->wrcnt || lock->rdcnt ) {
      lock->wrwcnt++;
      unlock(&lock->lock);
      semP(&lock->wrwait);
      return;

    }
    lock->wrcnt = 1;
    unlock(&lock->lock);
}
```

Figure 11.13: Function mrEnterWriter

Releasing a multireader lock that is held by a writer is the most complex operation (see Figure 11.14). To ensure a degree of fairness, readers are awakened first when both readers and writers are waiting for the lock. This prevents a continuous stream of writer processes from arriving at the lock and starving out the readers. Since subsequent arriving readers are blocked when one or more writers are waiting, it is guaranteed that writers will not be blocked indefinitely either. In this way, the lock rotates between readers and writers when both types of threads are waiting. Since readers can use the critical section in parallel, all readers are awakened whenever a writer leaves.

```
void mrExitWriter(MRLock lock)
{
    int rdrs;

    lock(&lock->lock);

    /*
     * let readers go first if any are waiting
     */
    if ( lock->rdwcnt ) {
      lock->wrcnt = 0;

      /*
       * awaken all readers that are presently waiting.
       */
      rdrs = lock->rdwcnt;
      lock->rdcnt = rdrs;
      lock->rdwcnt = 0;
      unlock(&lock->lock);

      while ( rdrs-- )
        semV(&lock->rdwait);

      return;
    }

    /*
     * no readers waiting, let one writer go (if any).
     */
    if ( lock->wrwcnt ) {
      lock->wrwcnt--;
      unlock(&lock->lock);
      semV(&lock->wrwait);
      return;
    }

    /*
     * nobody waiting - release lock.
     */
    lock->wrcnt = 0;
    unlock(&lock->lock);
}
```

Figure 11.14: Function mrExitWriter

It is possible to implement multireader locks with spin locks as well. These are useful for protecting critical sections that are too short for semaphores. Together, spin locks, semaphores, and multireader locks provide a useful base set of primitives for resolving contention in mutithreaded kernels.

11.1.9 Enhancement of Parallel Kernel Activity

Recall from previous design sections, that the amount of parallel kernel activity can be raised by duplicating the control thread of a kernel module. This section shows an example implementation of a kernel with dual thread manager modules.

First, the tmThread's have to get distinct threads ids (file `Topsy.h`):

```
#define MMTHREADID   -1     /* Memory Manager Thread Id. */
#define TMTHREADID0  -2     /* Thread Manager 0 Thread Id. */
#define TMTHREADID1  -3     /* Thread Manager 1 Thread Id. */
#define IOTHREADID   -4     /* Input/Output Manager Thread Id. */
```

Second, we need to have an additional thread structure and thread context (file `TMInit.c`):

```
Thread tmThread0, tmThread1;
ProcContext tmContext0, tmContext1;
```

Then, both of them have to be started, added to the appropriate thread and hash lists, and put onto the ready run queue (function `tmInit()` in file `TMInit.c`):

```
threadBuild(TMTHREADID0, 0, "tmThread0", &tmContext0,
            tmStack0, TM_DEFAULTTHREADSTACKSIZE,
            tmMain, (ThreadArg)userInit, KERNEL, NO_CPU, FALSE, &tmThread0);
threadBuild(TMTHREADID1, 0, "tmThread1", &tmContext1,
            tmStack1, TM_DEFAULTTHREADSTACKSIZE,
            tmMain, (ThreadArg)userInit, KERNEL, NO_CPU, FALSE, &tmThread1);

hashListAdd(threadHashList, &tmThread0, tmThread0.id);
hashListAdd(threadHashList, &tmThread1, tmThread1.id);

listAddInFront(threadList, &tmThread0, NULL);
listAddInFront(threadList, &tmThread1, NULL);

schedulerSetReady(&tmThread0);
schedulerSetReady(&tmThread1);
```

Both threads need to have a distinct stack in order to execute independently. Therefore, the function `mmVmInit()` (file `MMVirtualMemory.c`) has to allocate two stacks instead of just one:

```
/* tmStack */
loc += STACKSIZE;
vmInitRegion(space->regionList, (Address)loc, STACKSIZE,
             VM_ALLOCATED, READ_WRITE_REGION, 0);
*tmStackPtr0 = (Address)loc;
loc += STACKSIZE;
vmInitRegion(space->regionList, (Address)loc, STACKSIZE,
             VM_ALLOCATED, READ_WRITE_REGION, 0);
*tmStackPtr1 = (Address)loc;
```

Furthermore, the argument list of all functions handling with the initialization of thread manager data structures have to be adopted to support the duality.

In order to distribute all thread management system calls among the two thread managers, the system call interface has to implement a distribution policy. Our approach uses a simple round-robin policy, i.e., the receiver of the corresponding syscall message is swapped after every system call. In the original Topsy kernel, the constant `TMTHREADID` is used in every thread management system call as the receiver of the corresponding message. In order to retain the system call interface, the constant was replaced by a macro which implements the round-robin policy:

```
#define TMTHREADID   (ThreadId)((targetTmId+1)%2)
```

At this point, two system calls provided by the thread manager can be executed simultaneously on two distinct processors. Therefore, the system calls have to be analyzed to identity critical section and to protect them somehow.

Global Values

There are four global values of interest in the thread manager. The (linear) thread list and the hash list of all threads are both protected by a spin lock and therefore not susceptible to race conditions. The next two values represent the next valid kernel or user thread id respectively. They are used in function `getThreadId()` which is analyzed next.

getThreadId()

The function `getThreadId()` returns an unique id for a new kernel or user thread. It does this by reading the global variable `nextKernelThreadId` (`nextUserThreadId`) and checking the hash list for an existing thread with this id. If no thread is found the id is returned and the global variable is decremented (incremented) by one. The access to the global variables is not synchronized and therefore susceptible to race conditions. However, because the function is only called in `threadStart()` it can be protected by adding the function call to the critical section of `threadStart()`.

threadStart()

The call to the function `getThreadId()` can be added to the critical section:

```
lock(threadLock); {
   threadPtr->id = getThreadId(space);
   if ((threadPtr->id == 0) ||
   ...
}
```

All other functions from `TMThread.c` are not susceptible to race conditions.

12 Performance Analysis of TopsySMP

This chapter presents the simulation results for the symmetric multiprocessor version of Topsy running on top of the SimOS environment.

12.1 Introduction

Ideally, the overall system throughput of an SMP system will increase linearly as more processors are added. Thus a two-processor system should be able to handle twice the throughput of an UP. How close an MP implementation can approach this ideal depends on three main factors: the hardware architecture, the application job mix, and the kernel implementation.

If the hardware design is not suitable for an SMP system, then no amount of software tuning will allow an implementation to approach the ideal goal of linear performance increase as additional processors are added. The application job mix refers to the number and type of applications that are run on the system. Is is important to understand the application job mix of any benchmark in order to interpret the results correctly.

We studied two aspects of the performance of TopsySMP: overall speedup for benchmark applications, and the time needed for internal kernel operations such as context switching or exception handling.

12.2 Simulation Environment

All performance measurements were made running TopsySMP on top of the SimOS environment configured as a small-scale multiprocessor system. The particular hardware configuration used in our experiments consists of up to 64 MIPS R4000 CPUs running at 25 MHz and a variable sized shared memory connected over a single bus. The memory model (BusUma) uses uniform memory access time with bus contention, snoopy caches, and writeback buffers.

The system was running with 32 kBytes of first-level instruction and data cache and 1024 kBytes of unified second-level cache. All caches are 2-way associative with a line size of 64 Bytes (1st-level) and 128 Bytes (2nd-level) respectively. The time for a second-level cache hit was set to 50 ns. The maximum bus bandwidth was limited (by configuration) to 1200 MB/s. The total time to fetch a cache line from memory in an unloaded system was set to 500 ns. The time to get data out of another CPU's cache was set to 250 ns. And finally the total time to issue an upgrade in an unloaded system was set to 400 ns.

12.3 Benchmarks

In order to show how the overall performance vary with added processors we used three different benchmark application:

- **Sum**. A compute bound benchmark forming the sum of M numbers where M is much larger that the number of processors N.
- **Reserve**. A I/O bound benchmark simulating a number of travel agency's trying to reserve a fixed number of seats in a airplane.
- **Syscall**. A synthetic syscall-bound benchmark based on a test program called *Crashme* distributed with Topsy.

The following sections describe the benchmarks in more details.

12.3.1 Sum

Consider the simple problem of forming the sum of M numbers where M is much larger that the number of processors N. In this case, N partial sums can be formed in parallel by breaking the list into N lists, each with M/N numbers in it.

The benchmark application *Sum* uses N threads to form the partial sums in parallel but uses only a single thread to add the partial sums. The range of numbers to add and the resulting partial sum are exchanged among the threads using shared-memory. The parent threads waits for all child threads to terminate before adding the partial sums.

On a single processor the benchmark spends 97% of its execution time in user mode and only 3% in kernel mode.

12.3.2 Reserve

Consider the problem of several travel agency's trying to reserve a given (fixed) number of seats in an airplane.

The benchmark *Reserve* uses n threads to simulate the travel agency's which concurrently try to increase the number of reserved seats until a given maximum is reached. The access to the global count of reserved seats is not synchronized. Therefore, it is possible that more seats are reserved than are actually available. The parent thread waits for all child threads to terminate before adding the number of seats reserved by every child.

On a single processor the benchmark spends 34% of its execution time in user mode and 66% in kernel mode, mainly doing console output (half of the execution time).

12.3.3 Syscall

The benchmark *Syscall* uses a mix of different system calls to stress the system. These system calls include basic memory management and thread management functions.

This benchmark was chosen because his job mix can be easily adapted to lie in between the compute bound *Sum* and the I/O bound *Reserve*. Furthermore, the system calls can be assorted to measure the throughput of a specific kernel module.

The variant of *Syscall* used to collect the results shown in Figure 12.1, the benchmark was configured to spend 57% of its execution time in user mode and 43% in kernel mode. The kernel time was further separated into 14% thread management, 69% memory management and 17% I/O.

12.4 Benchmark Results

Figure 12.1 shows the results of the overall speedup measurements using benchmark applications.

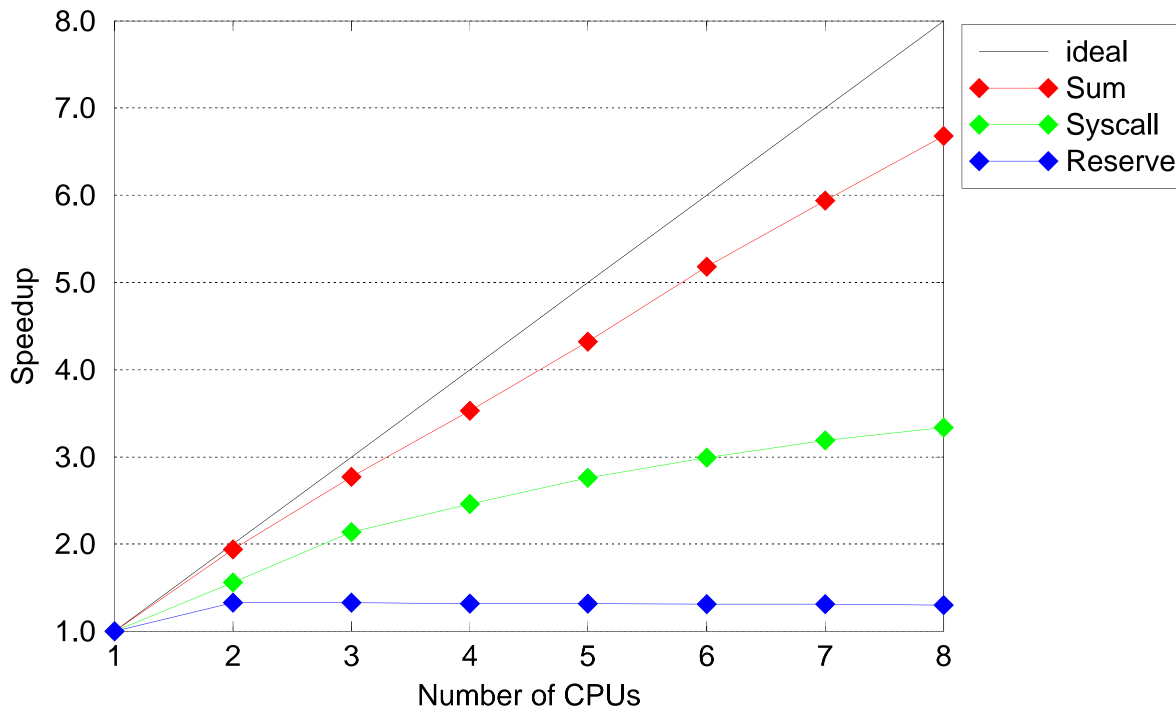


Figure 12.1: Benchmark Results

The results show that TopsySMP scales well for compute bound benchmarks like *Sum*. On the other extreme lies the I/O bound benchmark *Reserve* which shows no performance improvement over an UP system, regardless of the number of processors present in the system. This is due to the fact, that the `ttya` device driver handling the console output is pinned to CPU 0. Thus, adding additional CPUs will not increase performance any further because they can do nothing to reduce the 50% of the

time the benchmark spends doing I/O. The results even show, that additional CPUs are obstructive because of the increasing synchronization overhead.

In between lies the synthetic *Syscall* benchmark, whose 57% of user mode execution can lead to performance improvement. At the same time we see that the remaining 43% of kernel mode execution prevents a better speedup. This is the result of the fact, that the system calls provided by one kernel module cannot be handled in parallel. So if the application spends almost 70% of its kernel mode execution time within a single module, we have a great amount of serial code which cannot be parallelized.

12.5 Kernel Locking

12.5.1 Spinlock Times vs. Context Switch

In order to make a statement on the effectiveness of the synchronization primitives used in TopsySMP, the average time of a spin lock acquirement has to be determined. This time is then compared to the average context switch time. Both times were collected using the statistics mechanism from SimOS.

The time needed to acquire a kernel lock varies from 21 cycles up to 2500 cycles. The lower bound is the minimum time needed by the function `testAndSet()` without spinning. The function consists of a dozen assembler instructions if we take the `nop` instructions into count. The upper bound results from a lock contention in which the calling thread has to spin several times before acquiring the lock. Despite the wide time range, the average time spend to acquire a lock is never more than 40 cycles, normally it is about 25 cycles.

If we define the context switch time to be sum of the time needed calling the function `saveContext()`, scheduling a suitable thread, and calling the function `restoreContext()`, then we measure the following values (processor cycles):

Value	**Min**	**Max**	**Average**
saveContext	58	147	69.6
schedule	116	1859	440.1
restoreContext	46	101	50.0
context switch	220	2107	559.7

Although `saveContext` and `restoreContext` do a constant amount of work (the context has a fixed size), the gap between the minimum and maximum cycle time is quite big. This results from cache misses during the read/write operations.

So if we compare the two values measured above, we see, that the average time to acquire a spin lock is more than a factor 10 smaller than the time for a context switch.

12.5.2 Complex locking strategies

Our performance tests have clearly shown, that the time spent inside the `lock` system call is short enough to allow the use of simple spin locks within the entire kernel instead of more complex synchronization primitives.

12.5.3 Detailed Spin Lock Times

The spin lock times presented above were further separated into the times spent acquiring and holding the individual kernel locks. Recall, that TopsySMP uses three spin locks to guarantee system integrity. All values in the following table are processor cycles.

	acquire lock			**hold lock**		
Spin Lock	**Min**	**Max**	**Average**	**Min**	**Max**	**Average**
schedLock	21	2009	24.9	87	4491	149.4
hmLock	21	2503	51.2	76	5100	1561.4
threadLock	21	31	23.4	1104	8515	3737.9

Recall from Section 10.2.6 that the `hmLock` should be object to the highest lock contention. This is confirmed by our measurements, as the average time to acquire the `hmLock` is more than twice as much as the time of every other lock.

12.6 Internal Kernel Operations

12.6.1 Exception Handling

If we define the time to handle an exception as the time spent executing inside the general exception handler, than the kernel spends between 200 and 2500 cycles handling exceptions. The average values is 830 cycles.

12.6.2 System Call

Recall from Chapter 8 that all system calls in Topsy are based on the exchange of messages between kernel and user threads. Therefore, the time needed to execute a system call is the sum of the time needed to send a message and the time waiting for a reply (which of course includes the time actually performing the desired action by the corresponding kernel module). These values vary from 2500 cycles up to 14 000 cycles. The lower bound results from the simplest system call which is `tmGetInfo` with parameter `SELF`, returning the thread id of the calling thread. The average time for a system call amounts to 6500–7000 cycles.

12.6.3 SMP Management Overhead

In order to make a statement on how much system administration overhead is originated by the SMP version, we compared the uniprocessor version of Topsy with the SMP version running on a single CPU. Beside the overall performance, we measured spin lock times, context switching times, and scheduling times. One would expect, that these times only differ in case of more complex algorithms used in the SMP version.

If we compare the overall performance of the two systems, we find only irrelevant differences in the execution times. If we take for example the *Sum* benchmark, the execution times vary less than a hundredth percent. However, if we compare the spin lock times and the times used for context switching, we find some differences illustrated in Table 12.1 (all values in processor cycles).

	UP Version			MP Version		
Value	**Min**	**Max**	**Average**	**Min**	**Max**	**Average**
Generic Syscall	2933	12596	6323.6	3208	12965	6753.4
Exception Handling	254	802	564.7	230	1270	586.6
saveContext	52	65	52.1	58	72	58.2
schedule	108	206	133.4	116	232	150.2
restoreContext	46	61	46.1	46	71	46.3
schedLock acquire	23	25	23.1	21	23	21.1
schedLock hold	25	947	53.8	87	954	135.4
hmLock acquire	23	37	23.4	21	33	21.4
hmLock hold	78	844	483.1	76	867	481.7
threadLock acquire	23	23	23	21	21	21
threadLock hold	25	1831	36.1	1019	1842	1391.2

Table 12.1: UP vs. MP with one Processor

If we take a look at the scheduling times, for instance, we see that the simpler uniprocessor scheduler needs less time to scheduler a suitable thread. The multiprocessor scheduler is more complex and includes some overhead in the case of a single CPU, mainly from the additional tests needed to implement CPU-pinning. However, with an average overhead of 20–30 cycles the MP scheduler is still an efficient implementation.

The values for exception handling show the same characteristics. The exception handler of the MP kernel is not much more complicated than its UP counterpart. However, this is only true for a single CPU since for accessing the exception context of CPU0 no index calculation is necessary. The overhead in the MP exception handling therefore merely comes from the additional compare instructions.

Next thing to observe is the wide difference between the times the schedLock is hold. This results from the fact, that the uniprocessor `schedule()` function does not actually hold the lock, but merely checks its state using the following code:

```
if (!lockTry(schedLock)) return;
else unlock(schedLock);
```

The same is true for the threadLock inside the scheduler. However, both locks are used in other functions, therefore the hold time is always bigger than the time used to acquire it.

13 Conclusions

In this thesis, we have presented a simple SMP kernel architecture for small-scaled shared-memory multiprocessor systems. This architecture consists of a number of concurrent threads that together provide high performance and scalability.

We have achieved the following goals:

- **Simplicity.** The simple structure of the Topsy OS has not been complicated by an over-sized SMP mechanism. Instead, TopsySMP uses the same amount of locks and adds only a few additional SMP data structures and functions to the kernel.
- **High degree of parallel Kernel Activity.** The kernel scales nearly linear on compute bound application job mixes. Extreme I/O bound benchmarks have shown no performance improvement because the I/O threads are pinned to a specific CPU and therefore become a system performance bottleneck. The maximum amount of parallel kernel activity is given by the multithreading of the kernel. Thus, the more time an application spends doing system calls from a specific kernel module, the more it gets delayed by the serial handling of the system calls inside the kernel module.
- **Uniprocessor API.** The system call API of the original Topsy OS was adopted completely. Therefore, all programs written for Topsy are can to run on the SMP port without modifications. A single system call has been added to the API, in order to provide CPU-pinning.
- **Scalability.** TopsySMP is scalable from a single CPU up to 64 CPUs (limited by SimOS). This means, that additional CPUs can be added to (or removed from) the system without recompiling or even reconfiguring the kernel. This was achieved by determining the number of available CPUs in the system upon startup and dynamically allocating the necessary kernel data structures from the heap memory. However, we found, that the efficiency of the kernel is obtained on a small-scaled system with up to eight processors.
- **Multithreading.** The multithreaded architecture of the original Topsy kernel was adopted unchanged. This led to a straight forward implementation of an SMP kernel using the same locking strategy as Topsy. However, as mentioned above, the maximum amount of parallel kernel activity is given by the multithreading of the kernel. Thus, to improve the parallel kernel activity, the kernel threads have to be further parallelized.
- **Parallel Thread Scheduling.** Each CPU runs an instance of the scheduler. This provides a degree of fault tolerance because the CPUs are not dependent of a central scheduler.

- **Efficient Synchronization Primitives.** A hybrid coarse-grained/fine-grained locking strategy was used that has the low latency and space overhead of a coarse-grained locking, while having the high concurrency of a fine-grained locking strategy. The coarse-grain locks protect entire kernel threads and therefore large amounts of data, while the fine-grained locks protect single kernel data structures. The results of our performance experiments clearly demonstrated the effectiveness of our locking strategy, at least on the simulated hardware. The time spent inside the lock system call is short enough (compared with the time of a context switch) to allow the use of simple spin locks within the entire kernel instead of more complex synchronization primitives.
- **Powerful Simulation Environment.** This thesis has shown that SimOS provides a powerful simulation environment for running and profiling different kernel implementations. SimOS simulates the hardware of a computer system in enough detail to boot an operating system and run realistic workloads on top of it. Our experiences in porting Topsy to SimOS can serve others in trying to develop a new operating system or port an existing one to the SimOS environment.

Overall, we have found that the implementation of an SMP kernel based on a multithreaded uniprocessor kernel is straightforward and results in a well structured and clear design. The overhead caused by the integration of SMP functionality was kept to a minimum, resulting in a small and efficient implementation of a multithreaded microkernel for symmetrical multiprocessing hardware architecture.

13.1 Future work

The hardware implementation of a small-scale SMP system with up to four CPUs and TopsySMP as operating system would be an interesting and challenging task. A possible hardware target could be based on multiple MIPS processors like, for example, a workstation from Silicon Graphics. Since Topsy was already ported to the Intel processor architecture ([Ruf98]), another interesting target hardware would consist of a dual Pentium board available from many vendors these days.

Section 11.1.9 has suggested a way to further improve the performance of a multithreaded SMP kernel. The idea of using multiple instances of a kernel module control thread, could be further investigated using TopsySMP as basic kernel design and SimOS as simulation environment.

Another multiprocessor system architecture which could be investigated with SimOS, would be that of a distributed system consisting of multiple SMP clusters connected by a network.

Part IV

APPENDIX

A MIPS R4000 Architecture

This chapter describes the MIPS R4000 processor family (also referred to in this thesis by R4k).

A.1 Introduction

The MIPS R4000 processor provides complete application software compatibility with the MIPS R2000, R3000, and R6000 processors. Although the MIPS processor architecture has evolved in response to a compromise between software and hardware resources in the computer system, the R4000 processor implements the MIPS ISA for user-mode programs. This guarantees that user programs conforming to the ISA execute on any MIPS hardware implementation.

A.2 Processor General Features

This section briefly describes the programming model, the memory management unit (MMU), and the caches in the R4000 processor. Figure A.1 shows a block diagram of the MIPS R4000 processor.

- **Full 32-bit and 64-bit Operations.** The R4000 processor contains 32 general purpose 64-bit registers. (When operating as a 32-bit processor, the general purpose registers are 32-bits wide.) All instructions are 32 bit wide.
- **Efficient Pipeline.** The superpipeline design of the processor results in an execution rate approaching one instruction per cycle. Pipeline stalls and exceptional event are handled precisely and efficiently.
- **MMU.** The R4000 processor uses an on-chip TLB that provides rapid virtual-to-physical address translation.
- **Cache Control.** The R4000 primary instruction and data caches reside on-chip, and can each hold 8 Kbytes. All processor cache control logic, including the secondary cache control logic, is on-chip.
- **Floating-Point Unit.** The FPU is located on-chip and implements the ANSI/IEEE standard 754-1985.
- **Operating Modes.** The R4000 processor has three operating modes: User mode, Supervisor mode, and Kernel mode. The manner in which memory addresses are translated or mapped depends on the operating mode of the CPU.

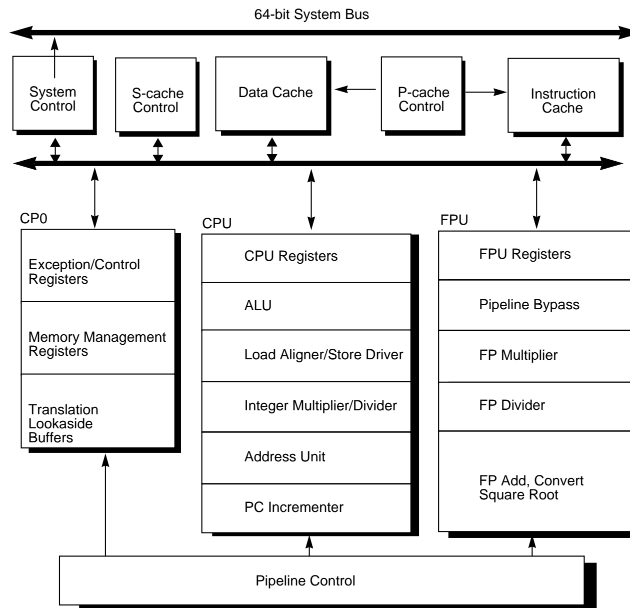


Figure A.1: Block Diagram of the MIPS R4000

The simulated MIPS R4000 processor in SimOS runs in 32-bit mode. Therefore, we will focus on 32-bit mode for the rest of this chapter.

A.3 Memory Management

The MIPS R4000 processor provides a full-featured memory management unit (MMU) which uses an on-chip translation lookaside buffer (TLB) to translate virtual addresses into physical addresses.

A.3.1 System Control Coprocessor, CP0

The System Control Coprocessor (CP0) is implemented as an integral part of the CPU, and supports memory management, address translation, exception handling, and other privileged operations. CP0 contains the registers shown in Figure A.2 plus a 48-entry TLB. The section that follow describe how the processor uses the memory management-related registers. Each CP0 register has a unique number that identifies it; this number is referred to as the *register number*. For instance, the *PageMask* register is register number 5.

A.3.2 Format of a TLB Entry

Figure A.3 shows the TLB entry formats for 32-bit mode. Each entry has a set of corresponding fields in the *EntryHi*, *EntryLo0*, *EntryLo1*, or *PageMask* registers, as shown in Figures A.4 and A.5; for example the *Mask* field of the TLB entry is also held in the *PageMask* register.

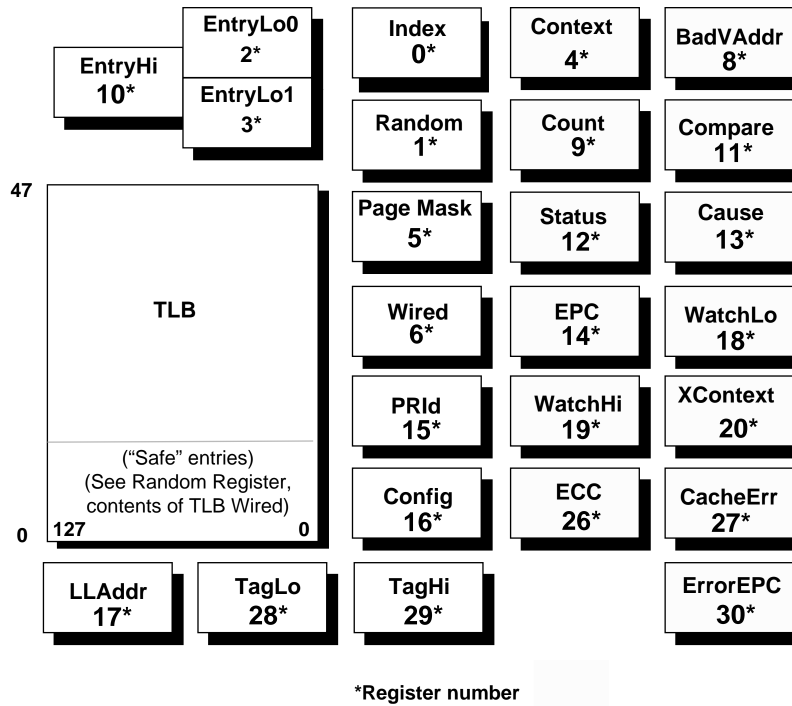


Figure A.2: CP0 Registers and the TLB

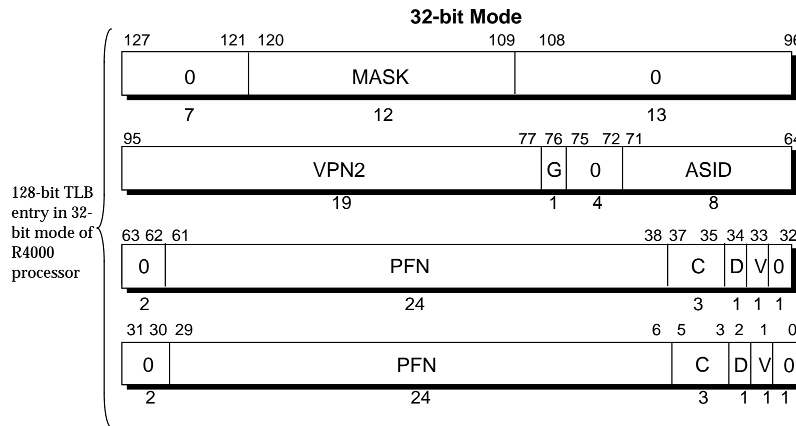


Figure A.3: Format of a TLB Entry

The format of the *EntryHi*, *EntryLo0*, *EntryLo1*, and *PageMask* registers are nearly the same as the TLB entry. The one exception is the *Global* field (*G* bit), which is used in the TLB, but is reserved in the *EntryHi* register. Figures A.4 and A.5 describe the TLB entry fields shown in Figure A.3.

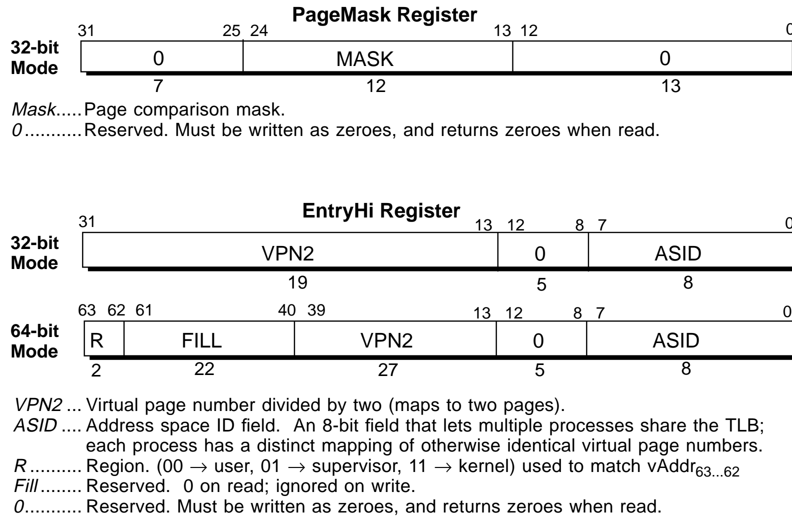


Figure A.4: Fields of the PageMask and EntryHi Registers

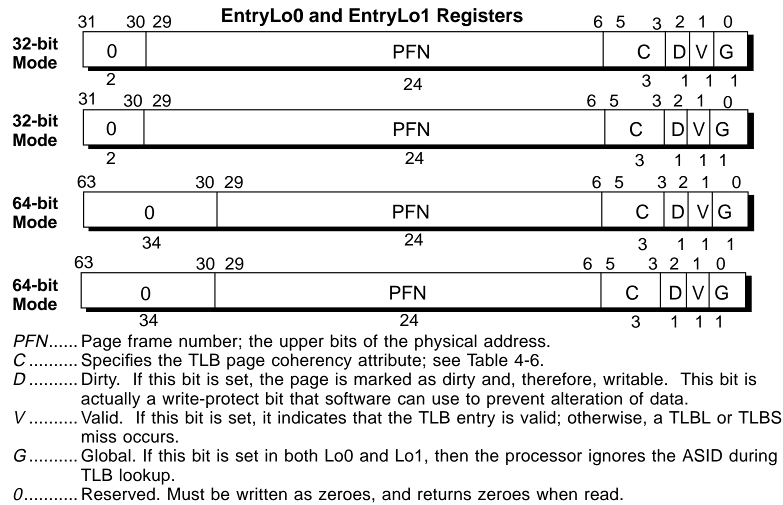


Figure A.5: Fields of the EntryLo0 and EntryLo1 Registers

The TLB page coherence attribute (*C*) bits specify whether references to the page should be cached; if cached, the algorithm selects between several coherency attributes. Table A.1 shows the coherency attributes selected by the *C* bits.

***C*(5:3) Value**	**Page Coherency Attribute**
0	Reserved
1	Reserved
2	Uncached
3	Cacheable noncoherent (noncoherent)
4	Cacheable coherent exclusive (exclusive)
5	Cacheable coherent exclusive on write (sharable)
6	Cacheable coherent update on write (update)
7	Reserved

Table A.1: TLB Page Coherency Bit Values

A.3.3 CP0 Registers

The following sections describe the CP0 registers, shown in Figure A.2, that are assigned in Topsy as a software interface with memory management (each register is followed by its register number in parentheses).

- *EntryLo0* (2) and *EntryLo1* (3) registers
- *PageMask* (5) register
- *Wired* (6) register
- *EntryHi* (10) register
- *PRId* (15) register

EntryLo0 (2), and EntryLo1 (3) Registers

The *EntryLo* register consists of two registers that have identical formats:

- *EntryLo0* is used for even virtual pages.
- *EntryLo1* is used for odd virtual pages.

The *EntryLo0* and *EntryLo1* registers are read/write registers. They hold the physical page frame number (PFN) of the TLB entry for even and odd pages, respectively, when performing TLB read and write operations. Figure A.5 shows the format of these registers.

PageMask Register (5)

The *PageMask* register is a read/write register used for reading from or writing to the TLB; it holds a comparison mask that sets the variable page size for each TLB entry. TLB read and write operations use this register as either a source or a destination; when virtual addresses are presented for translation into physical addresses, the corresponding bits in the TLB identify which virtual address bits among bits 24:13 are used in the comparison.

Wired Register (6)

The *Wired* register is a read/write register that specifies the boundary between the *wired* and *random* entries of the TLB shown in Figure A.6. Wired entries are fixed, non-replaceable entries, which cannot be overwritten by a TLB write operation. Random entries can be overwritten.

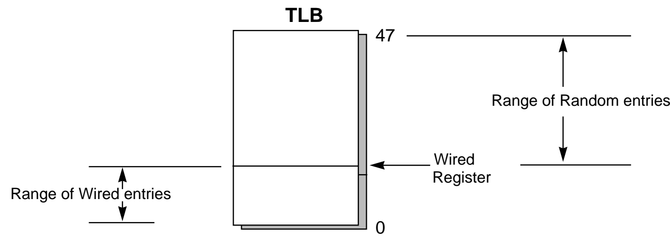


Figure A.6: Wired Register Boundary

The *Wired* register is set to 0 upon system reset. Writing this register also sets the *Random* register to the value of its upper bound. Figure A.7 shows the format of the *Wired* register; Table A.2 describes the register fields.

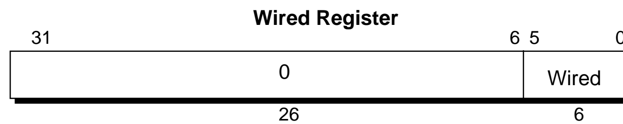


Figure A.7: Wired Register

Field	Description
Wired	TLB Wired boundary
0	Reserved. Must be written as zeros, and returns zeros when read.

Table A.2: Wired Register Field Descriptions

EntryHi Register (10)

The *EntryHi* holds the high-order bits of the TLB entry for TLB read and write operations. Figure A.4 shows the format of this register.

Processor Revision Identifier Register (15)

The 32-bit, read-only *Processor Revision Identifier (PRId)* register contains information identifying the implementation and revision level of the CPU and CP0. Figure A.8 shows the format of the *PRId* register; Table A.3 describes the register fields.

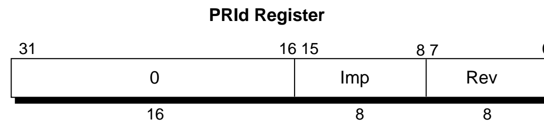


Figure A.8: Processor Revision Identifier Register Format

Field	Description
Imp	Implementation number
Rev	Revision number
0	Reserved. Must be written as zeros, and returns zeros when read.

Table A.3: PRId Register Field Descriptions

The low-order byte (bits 7:0) of the *PRId* register is interpreted as a revision number, and the high-order byte (bits 15:8) is interpreted as an implementation number. The implementation number of the R4000 processor is `0x04`. The content of the high-order halfword (bits 31:16) of the register are reserved.

The revision number is stored as a value in the form *y.x*, where *y* is a major revision number in bits 7:4 and *x* is a minor revision number in bits 3:0.

A.4 Exception Handling

The processor receives exceptions from a number of sources, including TLB misses, arithmetic overflows, I/O interrupts, and system calls. When the CPU detects one of these exceptions, the normal sequence of instruction execution is suspended and the processor enters kernel mode. The kernel then disables interrupts and forces execution of a software exception processor (called a *handler*) located at a fixed address. The handler saves the context of the processor, including the contents of the program counter, the current operating mode (user or supervisor), and the status of the interrupts (enabled or disabled). This context is saved so it can be restored when the exception has been serviced.

When an exception occurs, the CPU loads the *Exception Program Counter (EPC)* register with a location where execution can restart after the exception has been serviced. The restart location in the EPC register is the address of the instruction that caused the exception or, if the instruction was executing in a branch delay slot, the address of the branch instruction immediately preceding the delay slot.

A.4.1 Exception Processing Registers

This section describes only those CP0 registers that are used in exception processing, having changed from the MIPS R3000 to the MIPS R4000 processor family, or have significant meanings for the exception handling of Topsy. The complete list of CP0 registers can be found in [Hei94].

Status Register (12)

The *Status* register (SR) is a read/write register that contains the operating mode, interrupt enabling, and the diagnostic states of the processor. The following list describes the more important *Status* register fields:

- The 8-bit *Interrupt Mask (IM)* field controls the enabling of eight interrupt conditions. Interrupts must be enabled before they can be asserted, and the corresponding bits are set in both the *Interrupt Mask* field of the *Status* register and the *Interrupt Pending (IP)* field of the *Cause* register.
- The *Reverse-Endian (RE)* bit (bit 25) reverses the endianness of the machine. The processor can be configured as either little-endian or big-endian at system reset.

Figure A.9 shows the format of the *Status* register. Table A.4 describes the significant register fields used in Topsy.

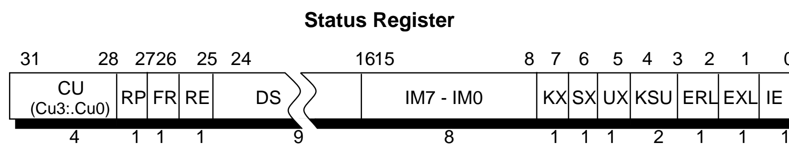


Figure A.9: Status Register

Field	**Description**
RE	*Reverse-Endian* bit, valid in User mode.
IM	*Interrupt Mask*; controls the enabling of each external, internal, and software interrupts. An interrupt is taken is interrupts are enabled, and the corresponding bits are set in both the *Interrupt Mask* field of the *Status* register and the *Interrupt Pending* field of the *Cause* register. $0 \rightarrow$ disabled $1 \rightarrow$ enabled
KSU	Mode bits $10_2 \rightarrow$ User $01_2 \rightarrow$ Supervisor $00_2 \rightarrow$ Kernel
ERL	Error Level; set by processor when Reset, Soft Reset, NMI, or Cache Error exception are taken. $0 \rightarrow$ normal $1 \rightarrow$ error
EXL	Exception Level; set by the processor when any exception other than Reset, Soft Reset, NMI, or Cache Error exception are taken. $0 \rightarrow$ normal $1 \rightarrow$ exception
IE	Interrupt Enable $0 \rightarrow$ disable interrupts $1 \rightarrow$ enable interrupts

Table A.4: Status Register Fields

Cause Register (13)

The 32-bit read/write *Cause* register describes the cause of the most recent exception. Figure A.10 shows the fields of this register; Table A.5 describes the register fields used by Topsy. A 5-bit exception code (*ExcCode*) indicates one of the causes, as listed in Table A.6. All bits in the *Cause* register, with the exception of the `IP(1:0)` bits, are read-only; `IP(1:0)` are used for software interrupts.

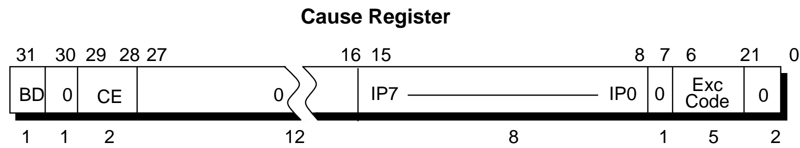


Figure A.10: Cause Register Format

Field	Description
BD	Indicates whether the last exception taken occurred in a branch delay slot. 1 → delay slot 0 → normal
IP	Indicates an interrupt is pending. 1 → interrupt pending 0 → no interrupt
ExcCode	Exception code field (see Table A.6)
0	Reserved. Must be written as zeros, and returns zeros when read.

Table A.5: Cause Register Fields

ExcCode Value	Mnemonic	Description
0	Int	Interrupts
1	Mod	TLB modification exception
2	TLBL	TLB exception (load or instruction fetch)
8	Sys	Syscall exception
16–22	-	Reserved
24–30	-	Reserved
31	VCED	Virtual Coherency Exception data

Table A.6: Cause Register ExcCode Field

Exception Program Counter Register (14)

The Exception Program Counter (EPC) is a 32-bit read/write register that contains the address at which processing resumes after an exception has been serviced. For synchronous exceptions, the *EPC* register contains either:

- the virtual address of the instruction that was the direct cause of the exception, or
- the virtual address of the immediately preceding branch or jump instruction (when the instruction is in a branch delay slot, and the *Branch Delay* bit in the *Cause* register is set).

The processor does not write the EPC register when the *EXL* bit in the *Status* register is set to a 1.

A.4.2 Exception Vector Location

The Reset, Soft Reset, and NMI exceptions are always vectored to the dedicated Reset exception vector at an uncached and unmapped address. Address for all other exceptions are a combination of a *vector offset* and a *base address*. During normal operation the regular exceptions have vectors in cached address space. Table A.7 shows the 32-bit-mode vector base address for all exceptions. Table A.8 shows the vector offset added to the base address to create the exception address.

Exception	**Vector Base Address**
Cache Error	`0xA000 0000`
Others	`0x8000 0000`
Reset, NMI, Soft Reset	`0xBFC0 0000`

Table A.7: Exception Vector Base Addresses

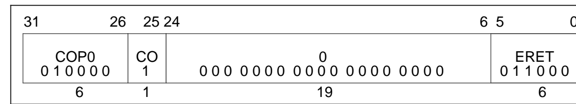
Exception	**Vector Base Address**
TLB refill, EXL = 0	`0x000`
Cache Error	`0x100`
Others	`0x180`
Reset, NMI, Soft Reset	none

Table A.8: Exception Vector Offsets

A.5 Instructions Set Details

This section provides a detailed description of the operation of the new R4000 instructions used in Topsy. The instructions are listed in alphabetic order. Details about all R4000 instructions can be found in [Hei94].

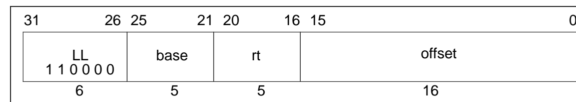
A.5.1 ERET – Exception Return



Format : `ERET`

Description: `ERET` is the R4000 instruction for returning from an interrupt, exception, or error trap. Unlike a branch or jump instruction, `ERET` does not execute the next instruction. `ERET` must not itself be placed in a branch delay slot.

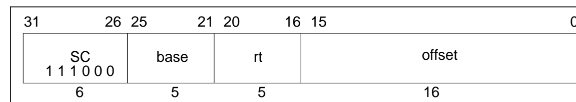
A.5.2 LL – Load Linked



Format : `LL rt, offset(base)`

Description: The 16-bit `offset` is sign-extended and added to the content of general register `base` to form a virtual address. The contents of the word at the memory location specified by the effective address are loaded into general register `rt`. The processor begin checking the accessed word for modification by other processors and devices.

A.5.3 SC – Store Conditional



Format : `SC rt, offset(base)`

Description: The 16-bit `offset` is sign-extended and added to the content of general register `base` to form a virtual address. The contents of general register `rt` are conditionally stored at the memory location specified by the effective address.

If any other processor or device has modified the physical address since the time of the previous Load Linked instruction, or if an `ERET` instruction occurs between the Load Linked instruction and this store instruction, the store fails and is inhibited from taking place.

B SimOS

SimOS is an environment for studying the hardware and software of computer systems. SimOS simulates the hardware of a computer system in enough detail to boot a commercial operating system and run realistic workloads on top of it. This chapter briefly describes SimOS as development environment used to design and run TopsySMP.

B.1 Introduction

SimOS is a machine simulation environment developed at the Computer System Laboratory at Stanford University. It is designed to study large complex computer systems. SimOS differs from most simulation tools in that it simulates the complete hardware of the computer system. In contrast, most other environments only simulate portions of the hardware. SimOS simulates the computer hardware with sufficient speed and detail to run existing system software and application programs. For example, the current version of SimOS simulates the hardware of multiprocessor computer systems in enough detail to boot, run, and study Silicon Graphics' IRIX operating system as well as any application that runs on it.

Simulating machines at the hardware level has allowed SimOS to be used for a wide range of studies including this thesis. Operating system programmers can develop their software in an environment that provides the same interface as the target hardware, while taking advantages of the system visibility and repeatability offered by a simulation environment. They can non-intrusively collect detailed performance-analysis metrics such as instruction execution, memory-system stall, and interprocessor communication time.

Although machine simulation is a well-established technique, it has traditionally been limited to small system configurations. SimOS includes both high-speed machine emulation techniques and more accurate machine simulation techniques. Using emulation techniques based on binary translation, SimOS can execute workload less than 10 times slower than the underlying hardware.

SimOS includes novel mechanisms for mapping the data collected by the hardware models back to concepts that are meaningful to the user. It uses a flexible mechanism called annotations to build knowledge about the state of the software being executed. Annotations are user-defined scripts that are executed when hardware events of particular interest occur. The scripts have non-intrusive access to the entire state of the machine, and can control the classification of simulation statistics. For example, an annotation put on the context switching routine of the operating system allows the user to determine the currently scheduled process and to separate the execution behavior of the different processes of the workload.

B.2 The SimOS Environment

Despite its name, SimOS does not model an operating system or any application software, but rather models the hardware components of the target machine. As shown in Figure B.1, SimOS contains software simulation of all the hardware components of modern computer systems: processors, memory management units (MMUs), caches, memory systems, as well as I/O devices such as SCSI disks, Ethernets, hardware clocks, and consoles. SimOS currently simulates the hardware of MIPS-based multiprocessors in enough detail to boot and run an essentially unmodified version of a commercial operating system, Silicon Graphics' IRIX.

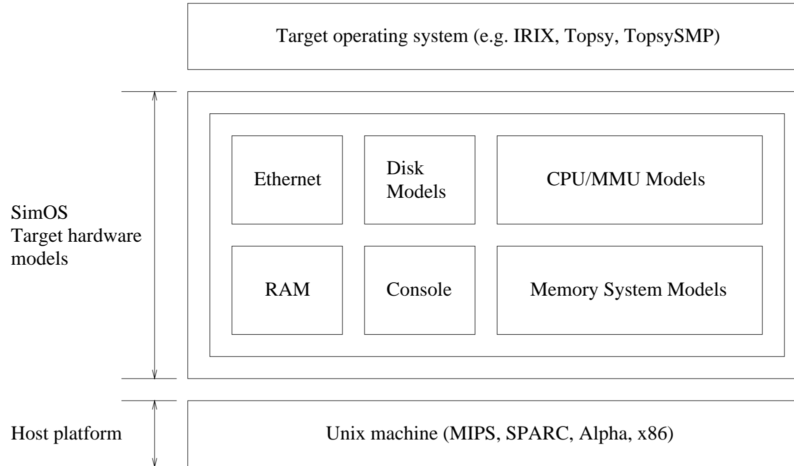


Figure B.1: The SimOS Environment

In order to run the operating system and application programs, SimOS must simulate the hardware functionality visible to the software. For example, the simulation model of a CPU must be capable of simulating the execution of all MIPS CPU instructions including the privileged instructions. It must also provide the virtual address to physical address translation done by the memory management unit (MMU). For the MIPS architecture this means implementing the associative lookup of the translation lookaside buffer (TLB), including raising the relevant exceptions if the translation fails.

SimOS models the behavior of I/O devices by responding to uncached accesses from the CPU, interrupting the CPU when an I/O request has completed, and performing direct memory access (DMA). The console and network devices can be connected to real terminals or networks to allow the user to interactively configure the workloads that run on the simulator.

B.2.1 Interchangeable Simulation Models

Because of the additional work needed for complete machine simulation, SimOS includes a set of interchangeable simulation models for each hardware component of the system. Each of these models is a self-contained software implementation of the device's functional behavior. Although all models implement the behavior of the hardware components in sufficient detail to correctly run

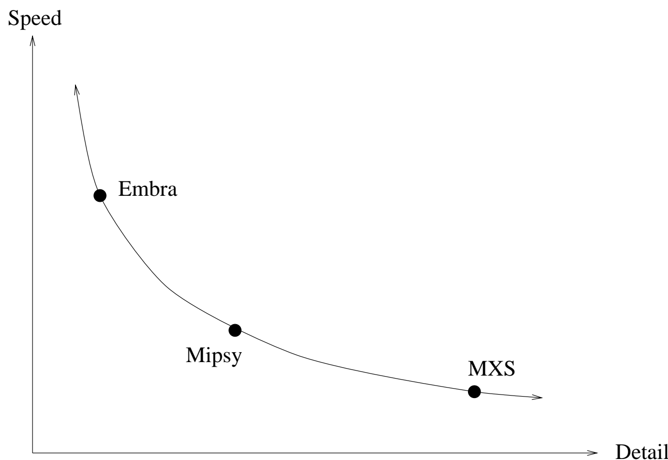


Figure B.2: Speed vs. Details

the operating system and application programs, the models differ greatly in their timing accuracy, interleaving of multiprocessor execution, statistics collection, and simulation speed (see Figure B.2).

Furthermore, the user can dynamically select which model of hardware component is used at any time during the simulation. Each model supports the ability to transfer its state to the other models of the same hardware component.

High-Speed Machine Emulation Models

To support high-speed emulation of a MIPS processor and memory system, SimOS includes *Embra* [WR96], which uses the dynamic binary translation approach. Dynamic binary translators translate blocks of instructions into code sequences that implement the effects of the original instructions on the simulated machine state. The translated code is then executed directly on the host hardware. Sophisticated caching of translations and other optimizations results in executing workloads with a slowdown of less than a factor of 10.

Detailed Machine Simulation Models

Although Embra's use of self-generated and self-modifying code allows it to simulate at high speeds, the techniques cannot be easily extended to build more detailed and accurate models. To build such models, SimOS uses more conventional software engineering techniques that value clean well-defined interfaces and ease of programming over simulation speed. SimOS contains interfaces for supporting different processor, memory system, and I/O device models.

SimOS contains accurate models of two different processor pipelines. The first, called *Mipsy*, is a simple pipeline with blocking caches such as used in the MIPS R4000. The second, called *MXS*, is a superscalar, dynamically scheduled pipeline with nonblocking caches such as used in the MIPS R10000. The two models vary greatly in speed and detail.

Mipsy and MXS can both drive arbitrarily accurate memory system models. SimOS currently supports memory system with uniform memory access time, a simple cache-coherent nonuniform memory architecture (CC-NUMA) memory system, and a cycle accurate simulation of the Stanford FLASH memory system.

For I/O device simulation, SimOS includes detailed timing models for common devices such as SCSI disks and interconnection networks such as Ethernet.

B.3 Data Collection Mechanisms

Low-level machine simulator such as SimOS have a great advantage in that they see all events that happen on the simulated system. These events include the execution of instructions, MMU exceptions, cache misses, CPU pipeline stalls, and so on. The accurate simulation models of SimOS are heavily instrumented to collect both event counts and timing information describing the simulated machine's execution behavior. SimOS's data classification mechanisms need to be customized for the structure of the workload being studied as well as the exact classification desired. A Tcl scripting language interpreter embedded in SimOS accomplishes this in a simple and flexible way. Users of SimOS can write Tcl scripts that interact closely with the hardware simulation models to control data recording and classification. These scripts can non-intrusively probe the state of the simulated machine and therefore can make classification decisions based on the state of the workload. The use of a scripting language allows users to customize the data collection without having to modify the simulator.

Annotations are the key to mapping low-level events to higher-level concepts. Annotations are Tcl scripts that the user can attach to specific hardware events. Whenever an event occurs that has an annotation attached to it, the associated Tcl code is executed. Annotations can run without any effect on the execution of the simulated system. Annotations have access to the entire state of the simulated machine, including registers, TLBs, devices, caches, and memories. Furthermore, annotations can access the symbol table of the kernel and applications running in the simulator, allowing symbolic references to procedures and data. Examples of simulated events on which annotations can be set include:

- Execution reaching a particular program counter address.
- Referencing a particular data address.
- Occurrence of an exception or interrupt.
- Execution of a particular opcode.
- Reaching a particular cycle count.

Annotations may also trigger other events that correspond to higher-level concepts. Using this mechanism we can define annotation types that are triggered by software events.

It is common that when collecting various numbers you'll want to know the average, standard deviation, minimum or maximum or some other set of values. SimOS provide statistics to help facilitate this common need. By repeatedly submitting entries to a particular statistics bucket, a simple statistical database is kept that allows extraction of all sorts of numerical data at a later point. Figure B.3 shows an example of an annotation script using statistics collection.

```
annotation set simos enter {
    statistics reset spinlock
}
annotation set pc kernel::lock:START {
    set spin_start $CYCLES
}
annotation set pc kernel::lock:END {
    global spin_start
    set cspintime [expr $CYCLES - $spin_start]
    statistics entry spinlock $cspintime
}
annotation set simos exit {
    console "***** SPINLOCK STATS *****\n"
    console "[statistics list spinlock]\n"
}
```

Figure B.3: Annotation Script for SimOS

The first annotation triggers on the initialization of the SimOS environment. Upon startup a statistics bucket called `spinlock` is instantiated and reset to zero. The next annotation triggers when the program counter (PC) is equal to the start address of the kernel function `lock`. Then, a variable called `spin_start` is set to the value of the current cycle counter. The third annotation is triggered on the event that the kernel function `lock` is left. It calculates the spin time (e.g. the difference between the current cycle counter and the value of the variable `spin_start` previously set by the second annotation) and submits its value to the spinlock statistics bucket. The last annotation triggers on the end of the simulation run and prints the spinlock statistics onto the console which will eventually look like this:

```
***** SPINLOCK STATS *****
n 2163 sumX 50394.000 minX 15.000 maxX 323.000
```

This means that there have been 2163 calls to the kernel routine `lock` which spent a total of 50 394 processor cycles inside the function. The minimum the processor spent inside the function was 15 cycles, the maximum 323 cycles.

SimOS provides a wealth of statistics during a simulation run, like execution time, memory system behavior and the number of instructions executed. It is often important to attribute the statistics to different phases of the workload. The most basic example is to split the workload between the time spent in the operating system and the time spent in the user process. Other breakdowns may be more complicated. For example you may want to consider each function of a user application to be a separate phase of execution.

SimOS provides a timing module as an easy way to separate and categorize execution behavior. To use the timing module, simply mark the start and end of each phase. The resulting output is a tree structure where each phase of execution is a node in the tree. The timing abstraction is quite powerful, and the SimOS group continues to improve its implementation. The long-term goal is a visual interface which will make it easy to collapse and expand nodes to further understand workload behavior.

The timing tree structure can be transformed in to a diagram showing the execution profile of a workload. Figure B.4 shows the startup phase of TopsySMP running on a 4-CPU multiprocessor and the execution of a compute bound benchmark. The execution time of each processor is broken down between the kernel, user, and idle modes. For kernel modes, the graph further separates the time spent executing in the individual kernel modules.

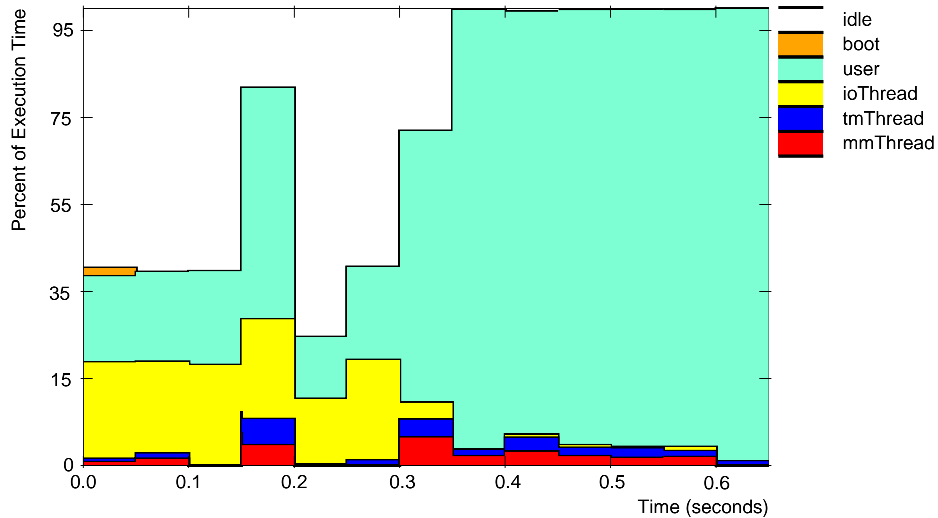


Figure B.4: Processor Mode Breakdown

C Project Description

Thema : TopsySMP (Symmetrical Multiprocessing)

Beginn der Arbeit : 26. Oktober 1998

Abgabetermin : 10. März 1999

Betreuung : George Fankhauser

Arbeitsplatz : ETZ C96

Umgebung : Solaris als Entwicklungsumgebung, Topsy V1.1 Sourcecode, MIPS R3000 Emulator

C.1 Einleitung

Topsy ist ein portables micro-kernel Betriebssystem, das am TIK für den Unterricht entworfen wurde. In der ersten Version wurde es für die Familie der 32-bit MIPS Prozessoren gebaut (R2k/R3k) und wurde bereits auf weitere Plattformen portiert. Es zeichnet sich durch eine saubere Struktur, eine hohe Portabilität (Trennung des Systems in hardware-abhängige und hardware-unabhängige Module) und eine gute Dokumentation aus. Dokumentation über Topsy ist auf der Topsy-Homepage unter http://www.tik.ee.ethz.ch/~topsy verfügbar.

C.2 Mehrprozessorsysteme mit gemeinsamem Speicher

Die Klasse der Mehrprozessorsysteme mit gemeinsamem Speicher bildet die am weitesten verbreitet Gruppe von Parallelrechnern. Aufgrund Memory-Bottlenecks beim Zugriff mehrerer Prozessoren auf den gleichen Speicher ist diese Art von Parallelrechner nicht skalierbar. Für eine kleiner Anzahl CPUs jedoch, kombiniert mit grossen Caches, können solche Maschinen eine gute Leistung als Workstation oder Server bringen. Bekannte Produkte werden auf Basis von intel, sparc, mips, alpha und weiteren CPUs hergestellt. Die Verwaltung mehrerer CPUs durch das Betriebssystem kann prinzipiell nach zwei Verfahren geschehen: Master/Slave oder symmterisch. Master/Slave Techniken werden v.a. in System angewandt, die monolithisch gebaut sind. Dabei bootet das OS auf einer CPU und behandelt weitere Prozessoren als Slaves. Diese Technik wird z.B. im MacOS angewandt. Sie hat den Nachteil, dass das OS selbst immer noch an eine CPU gebunden ist; zudem gibt es Implementationen die keine transparente Prozessmigration zu den Slave-CPUs erlaubt. Der symmetrische

Ansatz hingegen startet auf allen CPUs das gleiche Betriebssystem, wobei dieses den Zugriff auf gemeinsame Ressourcen entflechten muss. Im Fall von Topsy gibt es diese Entflechtung bereits für den Ein-Prozessor-Fall, da es sich bei diesem System um einen Kernel mit quasi-parallel laufenden Threads handelt, die an beliebigen Stellen unterbrochen werden können (pre-emptable kernel servers).

C.3 Aufgabenstellung

C.3.1 Verwandte Arbeiten

Um einen kurzen Überblick zu geben, soll die Arbeit die Implementation von SMP Hard- und Software beleuchten. Auf der Hardware-Seite soll der intel SMP Standard und eine typische RISC-Workstation mit mehreren CPUs gewählt werden (je nach verfügbarer Dokumentation). Bei der Systemsoftware, die SMP unterstützt, sollen kurz die Ansätze in Mach 3.0, Linux 2.1, NT 4.0 und Solaris 2.6 diskutiert werden.

C.3.2 Plattform

Als Entwicklungs- und Testplattform hat sich die MIPS R3000 Architektur wegen ihrer Einfachheit bestens bewährt. Um eine SMP-Plattform zu implementieren, soll auf Basis des R3000 Emulators eine Version entwickelt werden, die beliebig viele CPUs instanzieren kann. Dabei sollen Aspekte wie Debugging, Tracing und Performance (Caches) beachtet werden. Z.B. könnte jede virtuelle CPU als Java-Thread laufen, was eine Verteilung auf echten MP-Maschinen erlauben würde. Die Hardware-Mechanismen des SMP-Systems (Bus/Memory-Locks, Interrupt-Dispatching, etc.) soll sich an realen Systemen orientieren. Zudem soll das Modell der Maschine die Evaluation der Skalierbarkeit des Systems unterstützen, d.h. der Memorybottleneck soll auch auf dem Simulator 'spürbar' sein. Ein einfaches Cachemodell für die CPUs wäre wünschenswert. Der überarbeitete Emulator soll als rpm-Package und als tar.gz-file zur Verfügung gestellt werden.

C.3.3 Der SMP-Kernel

Auf der Basis der virtuellen, dokumentierten SMP-Hardware mit mehreren R3000 CPUs soll nun der Topsy Kernel so erweitert werden, dass sowohl Kernel- wie auch User-Threads auf allen CPUs laufen können und das System effizient ausgelastet wird. Als Grundlage soll ein Design-Dokument dienen, das die Funktionsweise der Erweiterungen detailliert beschreibt und die Entwurfsentscheidungen diskutiert. Falls Erweiterungen an der Syscall-Schnittstelle nötig sein sollten, sind diese ebenfalls zu genau zu dokumentieren. Anpassungen werden u.a. an folgenden Stellen notwendig sein:

- Initialisierung
- Scheduling
- Interrupt/Exception Handling

- Kernel Locks
- Idle Threads

C.3.4 Testen

Das Hauptziel der Arbeit ist eine saubere, verständliche und korrekte Implementation. Diese soll mit geeigneten Testprogrammen und anderen Hilfsmitteln (z.B. Tracing) überprüft werden.

C.3.5 Messungen

Schliesslich soll die Leistung des Systems evaluiert werden. Hier sollen folgende Aspekte beachtet werden:

- Vergleich zwischen single-processor Topsy und SMP Version: Was sind die Kosten des SMP Kernels im normalen Betrieb?
- Skalierbarkeit: Welcher Einfluss hat die Anzahl der Prozessoren auf verschiedene Aufgaben des Betriebssystems?
- Vergleich mit anderen Systemen: Dieser soll soweit wie möglich und aufgrund erhältlicher Messdaten durchgeführt werden.

C.3.6 Dokumentation

Neben der üblichen Diplomarbeit (schriftlicher Bericht) soll eine erweiterte Kurzfassung in Englisch abgegeben werden, die das System genügend gut beschreibt, um als Beilage zum Topsy Manual zu dienen (Appendix o.ä.).

C.4 Bemerkungen

- Mit dem Betreuer sind wöchentliche Sitzungen zu vereinbaren. In diesen Sitzungen soll der Student mündlich über den Fortgang der Arbeit berichten und anstehende Probleme diskutieren.
- Am Ende der ersten Woche ist ein Zeitplan für den Ablauf der Arbeit sowie eine schriftliche Spezifikation der Arbeit vorzulegen und mit dem Betreuer abzustimmen.
- Am Ende des zweiten Monats der Arbeit soll ein kurzer schriftlicher Zwischenbericht abgegeben werden, der über den Stand der Arbeit Auskunft gibt (Vorversion des Berichts).
- Bereits vorhandene Software kann übernommen und gegebenenfalls angepasst werden. Neuer Code soll möglichst sauber in den Bestehenden integriert werden.

C.5 Ergebnisse der Arbeit

Neben einem mündlichen Vortrag von 20 Minuten Dauer im Rahmen des Fachseminars Kommunikationssysteme sind die folgenden schriftlichen Unterlagen abzugeben:

- Ein kurzer Bericht in Deutsch oder Englisch. Dieser enthält eine Darstellung der Problematik, eine Beschreibung der untersuchten Entwurfsalternativen, eine Begründung für die getroffenen Entwurfsentscheidungen, sowie eine Auflistung der gelösten und ungelösten Probleme. Eine kritische Würdigung der gestellten Aufgabe und des vereinbarten Zeitplanes rundet den Bericht ab (in vierfacher Ausführung).
- Ein Handbuch zum fertigen System bestehend aus Systemübersicht, Implementationsbeschreibung, Beschreibung der Programm- und Datenstrukturen sowie Hinweise zur Portierung der Programme. (Teil des Berichts)
- Eine Sammlung aller zum System gehörenden Programme.
- Die vorhandenen Testunterlagen und -programme.
- Eine englischsprachige (deutschsprachige) Zusammenfassung von 1 bis 2 Seiten, die einem Aussenstehenden einen schnellen Überblick über die Arbeit gestattet. Die Zusammenfassung ist wie folgt zu gliedern: (1) Introduction, (2) Aims & Goals, (3) Results, (4) Further Work.

Glossary

This section contains all abbreviations used in this thesis.

API : Application Programming Interface

APIC : Advanced Programmable Interrupt Controller

BIOS : Basic Input/Output System

CPU : Central Processing Unit

DMA : Direct Memory Access

DSM : Distributed Shared-Memory

FIFO : First-In-First-Out

FPU : Floating-Point Unit

HAL : Hardware Abstraction Layer

I/O : Input/Output

IPC : Inter-Process Communication

ISA : Instruction Set Architecture

MIMD : Multiple Instruction Multiple Data

MMU : Memory Management Unit

MP : Multiprocessor or Multiprocessing

NUMA : Nonuniform Memory Access

OS : Operating System

PA : Physical Address

PC : Program Counter

PFN : Page Frame Number

RISC : Reduced Instruction Set Computer

SIMD : Single Instruction Multiple Data

SMP : Symmetrical Multiprocessor

TIK : (Institut für) Technische Informatik und Kommunikationsnetze

TLB : Translation Lookaside Buffer

UART : Universal Asynchronous Receiver/ Transmitter

UMA : Uniform Memory Access

UP : Uni-Processor

VA : Virtual Address

VPN : Virtual Page Number

Bibliography

[Bac86] M. J. BACH. *The Design of the UNIX Operating System*. Prentice Hall, 1986.

[Cat94] B. CATANZARO. *Multiprocessor System Architectures*. Sun Microsystems, 1994.

[Dij65] E. W. DIJSKTRA. Co-operating Sequential Processes. In *Programming Languages*. Academic Press, London, 1965.

[Fan] G. FANKHAUSER. *A MIPS R3000 Simulator in Java*. http://www.tik.ee.ethz.ch/~gfa/MipsSim.html.

[FCZP97] G. FANKHAUSER, C. CONRAD, E. ZITZLER, AND B. PLATTNER. Topsy - A Teachable Operating System. Computer Engineering and Networks Laboratory, ETH Zürich, 1997.

[Fly66] M. J. FLYNN. Very High-Speed Computing Systems. *Proccedings of the IEEE*, 54, December 1966.

[Hei94] J. HEINRICH. *MIPS R4000 Microprocessor User's Manual*. MIPS Technology, Inc., 2nd edition, 1994.

[Int94] Integrated Device Technology, Inc. *IDT79R3051, R3052 RISController Hardware User's Manual*, 1994. Revision 1.4.

[Int97] Intel Cooperation. *MultiProcessor Specification*, 1997. Version 1.4.

[KH92] G. KANE AND J. HEINRICH. *MIPS RISC Architecture*. Prentice Hall, 1992.

[KR88] B. W. KERNIGHAN AND D. M. RITCHIE. *The C Programming Language*. Prentice-Hall International, 2nd edition, 1988.

[RBDH97] M. ROSENBLUM, E. BUGNION, S. DEVINE, AND S. A. HERROD. Using the SimOS Machine Simulator to Study Complex Computer Systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, January 1997.

[RHWG95] M. ROSENBLUM, S. A. HERROD, E. WITCHEL, AND A. GUPTA. Complete Computer System Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology*, Fall 1995.

[Ruf98] L. RUF. Topsy i386. Semester Thesis at the Computer Engineering and Networks Laboratory, ETH Zürich, 1998.

[SS94] M. SINGHAL AND N. G. SHIVARATRI. *Advanced Concepts in Operating Systems*. McGraw-Hill, 1994.

[Sta] Stanford University. *The SimOS Home Page*. http://simos.stanford.edu.

[Suz92] N. SUZUKI, editor. *Shared Memory Multiprocessing*. MIT Press, 1992.

[Tan92] A. S. TANENBAUM. *Modern Operating Systems*. Prentice-Hall International, 1992.

[WR96] E. WITCHEL AND M. ROSENBLUM. Embra: Fast and Flexible Machine Simulation. *Sigmetrics '96*, 1996.