Purdue University
Purdue e-Pubs

Open Access Theses | Theses and Dissertations

5-2018

Multicore XINU

Philip M. Van Every

Follow this and additional works at: https://docs.lib.purdue.edu/open_access_theses

Recommended Citation
Van Every, Philip M., "Multicore XINU" (2018). *Open Access Theses*. 1468.
https://docs.lib.purdue.edu/open_access_theses/1468

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

MULTICORE XINU

A Thesis

Submitted to the Faculty

of

Purdue University

by

Philip M. Van Every

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

May 2018

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF THESIS APPROVAL

Dr. Douglas Comer, Chair

Department of Computer Science

Dr. George Adams

Department of Computer Science

Dr. Suresh Jagannathan

Department of Computer Science

Approved by:

Dr. Voicu Popescu by Dr. William J. Gorman

Head of the Graduate Program

To Kailee and Elizabeth, my other cores.

ACKNOWLEDGMENTS

I would like to express my profound gratitude to Dr. Comer, my advisor, for taking me on as a Master's student and allowing me to work on his elegantly crafted operating system. Thank you for washing away the obscurity related with the study of computing systems in general, and helping me sift through the nonsense to see clearly into operating systems and networking internals. Finally, thank you for your continued advice and words of encouragement.

To Dr. Adams and Dr. Jagannathan: Thank you for volunteering your time to provide advice and direction on this project. Your expertise as well as your teaching excellence drew me to invite you onto my Graduate Committee, and your enthusiasm and willingness to help me along demonstrates the depth of your generosity and dedication to your careers.

To Tom Trebisky: Thank you for spending so much time helping me troubleshoot hardware. Your website has been a constant companion. Your kind words of encouragement and thoughtful suggestions kept my spirits up throughout this project.

To Rajas Karandikar: As a TA and mentor you have helped me and countless others learn a great deal. I would especially like to thank you for getting the auxiliary cores running on XINU-VM. This project would likely not have been possible without you.

To Dr. Samanta: Thank you for your advice and the reference material you lent me. Your lecture on concurrency in a non-preemptive scheduling environment proved particularly helpful.

To Arnab, Vineeth, Sruthi, Matthew, Sam, Adib, and the rest of the Purdue Systems Research Group: Thank you for lending me your ears and brains when it was time for troubleshooting. Your insightful suggestions have been invaluable.

To my wife and closest friend, Kailee: Thank you for keeping me sane, helping me organize my thoughts, and tolerating my absentmindedness during times when I was particularly immersed in this project.

Finally, I would like to thank the wonderful folks at Sandia National Laboratories for the opportunity to study at Purdue as a Critical Skills Master's Program student. Thank you Tally, for helping me navigate the program. Thank you Cindi for your unconditional support.

TABLE OF CONTENTS

Page

LIST OF TABLES ix

LIST OF FIGURES x

ABSTRACT xi

1 INTRODUCTION 1

1.1 Processor Speed and Memory Access Speed 1

1.2 Operating Systems and Concurrency 1

1.3 The XINU Operating System 2

1.4 Roadmap 2

2 DEFINITIONS AND ABBREVIATIONS 3

2.1 Parallel Computing Terms 3

2.2 Operating Systems Terms 6

2.3 XINU Terms 7

3 BACKGROUND 8

3.1 Mutual Exclusion Primitives 8

3.1.1 Disabling Interrupts 8

3.1.2 Mutexes 9

3.1.3 Spinlocks 9

3.1.4 Notable Variations 10

3.2 Memory Considerations 10

3.2.1 Memory Consistency 10

3.2.2 Cache Coherence 11

3.2.3 False Sharing 12

3.3 Multicore OS Design 12

3.3.1 Master-Slave Kernels 12

Page

3.3.2 Spinlocked Kernels 13
3.3.3 Scheduling 14
3.3.4 Deadlocks 14
3.3.5 Interrupts 14
3.3.6 Initialization 15
4 MULTICORE XINU 16
4.1 Introduction 16
4.1.1 Medium Grained Spinlocking 17
4.1.2 XINU Invariants 17
4.1.3 XINU Levels 18
4.2 Hardware 18
4.2.1 Spinlocks 19
4.2.2 Inter-Processor Interrupts 20
4.2.3 Core Identification 21
4.3 Lock Management 21
4.3.1 Lock Policies 23
4.3.2 X-Sections 23
4.3.3 Deadlock Prevention with Nested Locking 25
4.3.4 X-Sections and Semaphores 26
4.4 Memory Manager 26
4.4.1 Low Level Memory Manager 27
4.4.2 High Level Memory Manager 27
4.5 Process Manager 29
4.5.1 Multiple Schedulers 30
4.5.2 `create` 30
4.5.3 `resume` 31
4.5.4 `ready` 33
4.5.5 `resched` 34

Page

4.5.6 `ctxsw_ret` . . . 35
4.5.7 `suspend` . . . 36
4.5.8 `kill` . . . 36
4.6 Process Coordination . . . 39
4.6.1 Semaphores . . . 39
4.6.2 Adding Multiple Processes to the Ready Queue . . . 39
4.7 Process Communication . . . 40
4.7.1 Process to Process Message Passing . . . 40
4.7.2 Inter-Process Communication Ports . . . 41
4.8 Real Time Clock Manager . . . 42
4.8.1 Preemption . . . 42
4.8.2 Delayed Events . . . 43
4.8.3 Asymmetry in `clkhandler` . . . 43
4.9 Device Manager . . . 43
4.9.1 TTY Driver . . . 44
4.10 Initialization . . . 45
5 CONCLUSIONS . . . 47
5.1 Remaining Levels . . . 47
REFERENCES . . . 49
A APPENDIX . . . 51

LIST OF TABLES

Table Page

4.1 Example of nested disable/restore critical sections 17

4.2 Example of nested x-sections 25

A.1 Multicore XINU Lock Policies 51

A.2 Allowable XINU process states and their meanings. 51

LIST OF FIGURES

Figure Page

3.1 A typical SMP architecture. 8

4.1 XINU's multilevel software architecture 19

4.2 ARM assembly spinlock implementation 20

4.3 x86 spinlock implementation in C 21

4.4 An entry in the global lock table 22

4.5 The `xsec_beg` and `xsec_end` functions 24

4.6 Allowable Process Manager state transitions in XINU 29

4.7 Mulicore XINU Process Manager state transitions with the DEAD state . 37

4.8 Multicore producers synchronized in the producer consumer problem . . . 42

A.1 The full multicore XINU process state machine 53

ABSTRACT

Van Every, Philip M. M.S., Purdue University, May 2018. Multicore Xinu. Major Professor: Douglas E. Comer.

Multicore architectures employ multiple processing cores that work together for greater processing power. Shared memory, symmetric multiprocessor (SMP) systems are ubiquitous. All software must be explicitly designed to support SMP processing, including operating systems. XINU is a simple, lightweight, elegant operating system that has existed for several decades and has been ported to many platforms. However, XINU has never been extended to support multicore processing. This project incrementally adds SMP support to the XINU operating system. Core kernel modules, including the scheduler and memory manager, have been successfully extended without overhauling or completely redesigning XINU. A multicore methodology is laid out for the remaining kernel modules.

1. INTRODUCTION

1.1 Processor Speed and Memory Access Speed

Several decades ago, a typical processor ran with a clock speed on the order of MHz. Modern processor clocks cycle at speeds on the order of GHz [1]. As clock speeds increase, heat dissipation is becoming a limiting factor. Even with sophisticated multilevel cache architectures, memory systems are unable to keep up with modern processors' demand for data [1]. Modern manufacturing techniques are approaching upper limits on the complexity of computer circuitry that they can produce.

Faced with these barriers, computer vendors have begun to add more processors to increase computational power. Multiprocessor and multicore platforms raise new and unique challenges for software designers.

1.2 Operating Systems and Concurrency

Operating systems often allocate processor time to multiple processes and then switch between them at a high frequency, giving the impression of simultaneous execution. However, because there is only one processor, only one processes is executing at a time. Such concurrency is implicitly provided to an application developer by a unicore operating system.

Multicore systems allow for true parallelism: Two or more processes or threads of execution may execute on two or more processing cores *at the exact same time*. However, such parallel execution cannot be exploited implicitly. An application programmer must design multi-threaded applications and the operating system must manage them. In order to manage multi-threaded applications on a multicore plat-

form, the operating system itself must be designed to provide multicore support. Thus, Multicore support has become standard among modern operating systems.

1.3 The XINU Operating System

Written in the C programming language for deployment on embedded systems, XINU is a small, simple, efficient, and elegant operating system [2]. XINU uses minimalist implementations of all standard kernel modules, including process management and scheduling, devices and drivers, file systems, etc. It is widely used as a teaching tool, but also employed in real industrial systems. While XINU has been ported to run on a diverse collection of architectures and embedded platforms, it has not yet been extended to run in a multicore environment.

This project incrementally adds multicore support to the XINU operating system while maintaining its policies, semantics, organizational paradigms, and intuitive design style. Thus, this is not an overhaul or redesign of XINU, but rather an extension of XINU to the world of parallel execution.

1.4 Roadmap

Chapter 2 defines potentially ambiguous terms related to operating systems and parallel processing. To provide a basis for the discussion of a multicore XINU design, Chapter 3 gives an overview of relevant parallel computing principles and multicore operating system design paradigms. Chapter 4 explains and analyzes the policies and mechanisms of an example multicore XINU design. Finally, Chapter 5 provides some big picture perspective and identifies opportunities for continued work.

2. DEFINITIONS AND ABBREVIATIONS

2.1 Parallel Computing Terms

Concurrent Execution, a.k.a. Multiprogramming, Multitasking: This is an execution scheme in which multiple processes share a processor and take turns running, typically with a preemptive, time-sharing scheduler.

Parallel Execution, a.k.a. True Concurrency: This is an execution scheme in which multiple processes run simultaneously on more than one processor or core.

Semaphore: A Semaphore is a cooperative process synchronization primitive that allows a set number of processes to execute a protected section of code at any given time, causing ineligible processes to reschedule and block until a currently executing process exits the protected code section. These are typically used to synchronize access to a limited, shared resource, such as a device or memory buffer.

Mutex: A mutex is binary semaphore, i.e. a semaphore that protects a singular resource which only one process may access at a time.

Spinlock: Similar to a mutex, a spinlock ensures mutually exclusive execution of a critical section of code. However, a splinlock does so by causing the process to stall and waste CPU execution cycles rather than reschedule.

TAS - Test And Set: TAS is a spinlock implementation algorithm that works by continually attempting to atomically set a bit in memory until successful, thereby

marking the location as locked.

CAS - Compare And Swap: CAS is a spinlock implementation algorithm that works by continually comparing a memory location to a given value and attempting to atomically exchange the value in memory with the given value, thereby marking the location as locked.

CPU, Processor, and Core: Typical SMP systems have a single CPU with multiple cores. In such a system, all cores can be thought of as separate processors, and the terms "core", "processor", and "CPU" become synonymous. When discussing multicore XINU, this work uses the terms interchangeably.

Processor/Cache Affinity: To improve cache performance, a multicore OS may attempt to schedule a process on the same core each time the process is selected to run. The process is said to have affinity to the processor on which it is scheduled. The intent is that each time the process is scheduled, the processor's L1 cache will still hold cache lines from its previous execution, leading to a lower cache miss ratio.

Processes, Lightweight Processes, and Threads: The XINU operating system does not use virtual memory addressing. When discussing XINU, the concept of a process is similar to a thread or lightweight process. This work broadly refers to any unit of execution as a process.

SMP - Symmetric Multiprocessing: SMP is a multiprocessing paradigm in which all cores behave identically, share a memory system, and execute within the bounds of a common, shared OS.

AMP - Asymmetric Multiprocessing: AMP can be broady defined as anything that is not SMP [3]. In this scenario, cores behave differently. For example, one

core runs a main OS and another core runs a real time OS for specific, time sensitive computations.

MESI: MESI refers to a cache coherence protocol on SMP systems where each processing core has its own memory cache. Cache lines are tracked and maintained in one of four states: Modified, Exclusive, Shared, and Invalid.

MOESI: The MOESI protocol performs cache coherence in a manner similar to the MESI, but uses an additional cache line state: the Owned state.

ERG - Exclusive Reservation Granule: The ERG is the smallest amount of memory that can be marked for exclusive access in an SMP system.

Reentrant Spinlock, a.k.a. Recursive Spinlock: A reentrant spinlock allows a process that already holds a spinlock to reacquire the spinlock without spinning again. Similarly, a reentrant mutex allows a process that already holds a mutex to reacquire the mutex without blocking again.

Data Race, a.k.a. Race Condition: A data race occurs when multiple processes or threads attempt to modify the same piece of shared memory. If access to the shared memory is not mutually exclusive, the data object it represents may end up in an incorrect state.

Lock Acquisition: To acquire a synchronization primitive is to obtain the primitive, preventing other processes from executing the critical section of code that the primitive protects, e.g. to lock a spinlock or mutex.

Lock Release: To release a synchronization primitive is to reset it so that other processes may acquire it. E.g. to unlock a spinlock or mutex.

Critical Section: In a parallel or concurrent execution environment, a critical section is a body of code that contains a potential data race. Execution of critical sections of code must be mutually exclusive.

ISR - Interrupt Service Routine, Interrupt Handler: An ISR is the body of code assigned to be executed when a particular interrupt occurs.

GIC - Generic Interrupt Controller: On an ARM multicore system, the GIC is configurable interrupt control device. It enables/disables interrupt forwarding, controls which core will receive particular interrupt signals, and controls the order in which interrupts will be forwarded and processed based on assigned interrupt priorities.

APIC - Advanced Programmable Interrupt Controller: On an x86 multicore system, the APIC is configurable interrupt control device. It has similar duties to the ARM GIC.

IPI - Inter-Processor Interrupt: An interrupt generated by a processing core and sent to other processing cores via the GIC or APIC.

2.2 Operating Systems Terms

OS - Operating System: An operating system is a body of system management software that directly interfaces with and controls computer hardware.

System Call, a.k.a. Trap, Supervisor Call: A system call is a functions implemented in OS software that provides access to system resource in a controlled manner that upholds OS operational policies.

2.3 XINU Terms

LLMM - Low Level Memory Manager: LLMM refers to XINU's low level memory manager.

HLMM - High Level Memory Manager: HLMM refers to XINU's high level memory manager.

Lock: A lock (Section 4.3) is a multicore XINU wrapper on a hardware spinlock that implements a reentrant spinlock.

x-section: A multicore XINU critical section starting with the `xsec_beg` function and ending with the `xsec_end` function, described in Section 4.3. A process executing in an x-section may not be interrupted or preempted, and it may not yield the processor.

3. BACKGROUND

In an SMP system, all processing cores have symmetric access to shared memory and devices. Often, SMP cores have their own L1 cache, but share access to higher levels of the memory system. Figure 3.1 illustrates how caches are associated with cores.

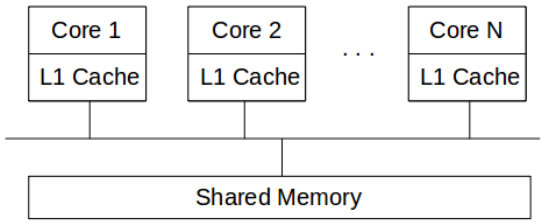

Fig. 3.1.: A typical SMP architecture.

3.1 Mutual Exclusion Primitives

Any concurrent or parallel software must protect shared data from data races. Mutual exclusion primitives ensure that only a single process will execute a critical section of code at any given time. Commonly used mutual exclusion primitives are discussed below.

3.1.1 Disabling Interrupts

If a processor is interrupted while accessing shared data, it may be directed to run an ISR that accesses the same shared data, creating a data race. If a process is preempted while accessing shared data, the next process may access the same shared data, again leading to a data race. A single core OS may prevent such preemption by disabling interrupts while executing critical sections of kernel code. Single core

versions of XINU, UNIX, Linux, and Windows disable interrupts while executing critical sections in OS kernel code [2–6]. This method of mutual exclusion is usually only used by the OS, and not by application programs. Disabling interrupts is not sufficient for mutual exclusion in a multicore environment because it does not prevent other cores from simultaneously accessing a critical section of code in parallel.

3.1.2 Mutexes

The name "mutex" is derived from the first letters of the words "mutual" and "exclusion". After a process acquires a mutex, other processes attempting to acquire the mutex will reschedule, yielding the processor and blocking until the mutex is released. Only one process may hold the mutex at a time, and will thus have mutually exclusive access to the critical section that the mutex protects. An operating system is responsible for implementing a mutex and may provide a mutex a.p.i. for use by application programs.

3.1.3 Spinlocks

The mechanics of a spinlock are well hinted by its name. Spinlocks attempt to "lock" a memory location by atomically setting the location to a designated value, usually 1. When attempting to acquire a spinlock that is already taken, a core will continuously attempt to acquire the spinlock until successful. The continuous, repetitive lock attempting behavior is called "spinning".

A process acquiring a spinlock avoids rescheduling while waiting to enter a critical section, but wastes processor execution cycles. Spinlocks are used to provide mutual exclusion between cores on a multicore system.

Multicore architecture allows for atomic memory accesses with particular instructions, like the atomic exchange instruction (XCHG) in x86 architectures [7] and the exclusive load and store instructions (LDREX/STREX) in arm architectures [8]. These

instructions can be used to implement spinlocks using the *test and set* and *compare and exchange* algorithms [3, 6, 9].

3.1.4 Notable Variations

Various locking primitives can be built on top of mutexes and spinlocks. For instance, read/write mutexes are designed to cater to the classic readers/writers problem, and allow multiple processes to read protected memory or a single process to modify the protected memory [1, 9].

Re-entrant mutexes and spinlocks may be acquired multiple times by a single owner without causing the owner to block. Allowing multiple acquisitions is desirable in situations where a mutex or spinlock owning entity may make calls to multiple functions that execute critical sections protected by the same mutex or spinlock. These are also called recursive locks [1, 9].

3.2 Memory Considerations

Implementation of synchronization primitives and other parallel or concurrent processing algorithms is constrained by the capabilities of the underlying hardware, particularly the memory system. The following sections address some particularly relevant SMP memory considerations.

3.2.1 Memory Consistency

Processors may use store buffers for writes to the memory system. The store buffer may reorder writes to memory, and may store them for a short time before sending them out on the memory bus. While store buffers make the memory system more efficient, they temporarily force multiple cores in an SMP system to have an inconsistent view of the state of shared memory locations. From the programmer's

point of view, such computational inconsistency is compounded a if compiler reorders memory writes.

In an SMP system, such inconsistencies are particularly troublesome for critical sections of code. If a spinlock acquisition and the first line of a critical section of code are swapped by memory or compiler reordering, the beginning of the critical section will be executed before the spinlock is held. If the release of a spinlock is swapped with the last line of a critical section, the last line of the critical section will be executed after the spinlock has been released.

ARM uses a weakly consistent memory model, meaning ARM systems are subject to both processor reordering and compiler reordering [8,10]. The x86 architecture uses a stronger memory consistency model [7,11] that is less prone to processor reordering, but still subject to compiler reordering.

C compilers provide directives to eliminate memory reordering when they are invoked. ARM and x86 provide memory barrier instructions, like DMB (Data Memory Barrier) and MFENCE, that ensure memory consistency when they are executed. These directives and instructions are used in spinlock implementations to prevent critical sections of code from being executed outside the bounds of the spinlock.

3.2.2 Cache Coherence

Processing cores in SMP systems often have their own L1 caches. If two or more caches hold the same cache line and that cache line is modified by one of the processors, the cache line must be updated in the other processor's cache. Otherwise, the two caches are incoherent.

Fortunately, modern SMP architectures implement fast cache coherence protocols in hardware. The MESI protocol is implemented at the hardware level in x86 architectures [7]. ARMv7 and up uses the MOESI protocol in conjunction with hardware monitors to maintain not only cache coherence, but also implement LDREX/STREX instructions used for memory spinlocks [8]. Thus, on modern platforms, cache coher-

ence is transparent to the multicore OS designer; It is handled automatically by the hardware.

3.2.3 False Sharing

Because cache coherence protocols operate on an entire cache line at a time, two spinlock words that share a cache line both appear locked when either one is locked. An unlocked spinlock that resides in the same cache line as a locked spinlock will appear to be unavailable, and the processor will spin unnecessarily when trying to acquire it. The phenomenon is known as "false sharing", and the cache line size is known as the Exclusive Reservation Granule. To prevent false sharing, multicore OS designers often space spinlocks at least the size of the ERG apart in memory [3, 8].

3.3 Multicore OS Design

Operating systems use global data structures to track and coordinate process execution, manage memory, files, devices, and otherwise maintain computing infrastructure. As noted in Section 3.1, disabling interrupts is not sufficient for mutual exclusion in a multicore system. Operating systems on multicore platforms must therefore provide some other means of mutual exclusion in kernel critical sections. Several multicore OS design paradigms are described below.

3.3.1 Master-Slave Kernels

An early multicore UNIX implementation avoided data races by only allowing kernel code to execute on a particular core [12], called the "master" core. To execute kernel code, a process running on a non-master core, called a "slave" core, would have to stop running and add itself to the scheduling list for the master core. Such a design is called a "master-slave" kernel. In master-slave kernels, the master core becomes a bottleneck [12].

3.3.2 Spinlocked Kernels

The alternative approach is to allow parallel execution of kernel code. Kernel functions often require complete and immediate execution, making it infeasible to reschedule when acquiring a mutex. Furthermore, high level synchronization primitives like mutexes and semaphores employ shared data structures that must also be protected. For instance, a mutex might track waiting processes on a list. If multiple processes attempt to add themselves to the list while acquiring an already locked mutex, they would have to modify the shared mutex list atomically, which would require another mutex. To protect that mutex list, another mutex would be required, and so on. Therefore, spinlocks must be used to protect parallel access to an operating system kernel. Operating Systems that use spinlocks to allow for parallel kernel access are called "spinlocked kernels" [12].

When designing a spinlocked kernel, consideration must be given to the granularity of allowed parallel access. Coarse grained locking allows minimal parallelism and requires relatively low design complexity. Fine grained locking allows for a higher degree of parallelism but requires greater complexity of design.

Early multicore UNIX and Linux designs used "giant locking" between two cores [5,12,13]. A single spinlock protected access to the entire kernel. Only one core could access any part of the kernel at a time. This approach allows for the same level of parallel kernel execution as a master-slave kernel without the overhead of migrating processes that wish to access the kernel to a specific core.

Subsequent UNIX and Linux designs used coarse grained locking [5,12,13]. With coarse grained locking, a small number of spinlocks protect entire kernel modules. For instance, one spinlock may protect access to the entire process table and all processes, while another spinlock protects access to all devices and device drivers. However, as the number of cores running in a OS increases, the contention on the spinlocks becomes an efficiency bottleneck, leading to poor scalability [5,12,13].

More modern SMP kernels employ fine grained locking, in which more locks allow for a greater degree of parallel kernel access with less contention [5, 12, 13]. For instance, all process entries, devices, files, etc. may be protected by their own locks and can thus be accessed and modified independently and in parallel. Fine grained spinlocked kernels are more scalable, but exhibit a much greater complexity of design than their coarse grained counterparts.

3.3.3 Scheduling

To reduce contention on its ready list of eligible processes, an SMP OS scheduler might maintain a process queue for each core [3]. This allows for L1 cache affinity. However, it makes priority inversions impossible to avoid [14]. Priority inversions occur when a high priority process is waiting on one core's queue while a low priority process runs on another core. The benefits of L1 cache affinity in such a scheduling scheme may be negligible for SMP systems in which all cores share an L2 cache [15].

3.3.4 Deadlocks

Deadlocks occur when processes attempt to acquire multiple synchronization primitives cyclically. All processes in the cycle may become blocked, waiting for the other blocked processes in the cycle to release a synchronization primitive. To avoid locking cycles, operating systems often use nested locking [5, 12]. If all processes attempt to acquire synchronization primitives in the same nested order, no deadlock can occur.

3.3.5 Interrupts

In an SMP system, interrupts can be dynamically processed on any available core. Generally, however, interrupts are statically routed to predetermined cores [3]. Static interrupt routing allows interrupt handlers to exploit cache affinity.

On ARM SMP systems, a configurable interrupt management device called the Generic Interrupt Controller (GIC) routes each interrupt to the core that is configured to handle the interrupt. On x86 SMP systems, the Advanced Programmable Interrupt Controller performs similar interrupt management duties.

3.3.6 Initialization

Typically, a single core is used to boot the OS, initializing all OS data structures. Auxiliary cores are started up, but they simply block on a spinlock created by the bootstrap core until it is ready for them to be released [3]. Linux calls this a "holding pen" [5]. The single bootstrap core may initialize key kernel data structures, including spinlocks, without the possibility of data races related to parallel execution. When auxiliary cores begin running, they perform any necessary self initialization and may or may not help to perform other system initialization in parallel.

4. MULTICORE XINU

4.1 Introduction

XINU disables interrupts to achieve mutual exclusion between concurrently executing processes, as shown in Algorithm 4.1.

Algorithm 4.1 Unicore XINU critical sections use disable/restore.

```
intmask mask = disable();
/* execute kernel critical section */
restore(mask);
```

The `disable()` function call marks the beginning of a XINU critical section. It disables interrupts and returns the value of the processor's interrupt mask before it was called. The `restore(mask)` function call ends the critical section, restoring the processor's interrupt mask to its state just before the call to `disable()`.

XINU disable/restore supports nested critical sections. A nested critical section will will restore a saved interrupt mask before returning control to the outer critical section. When the outermost critical section is finished executing, interrupts will be re-enabled. This recursive mutual exclusion paradigm greatly simplifies kernel code organization because kernel functions may freely call other kernel functions. Table 4.1 shows an example function call sequence with nested disable/restore critical sections.

As mentioned previously, preventing preemption is not sufficient for mutual exclusion on a multicore platform. Spinlocks must be added to the XINU kernel to provide multicore support.

Table 4.1.: Example of nested disable/restore critical sections

Stack Frame	Code Execution	Interrupts
top level	mask = disable(); func_a(); restore(mask);	enabled
func_a	mask = disable(); func_b(); restore(mask);	disabled
func_b	mask = disable(); /* critical section */ restore(mask);	disabled

4.1.1 Medium Grained Spinlocking

With the goal in mind of maintaining XINU's simplicity and elegance, multicore XINU uses medium grained spinlocking. The medium spinlocking granularity allows for a high degree of kernel parallelism while keeping kernel logic simple.

Global structures each have their own lock. However, they each have only one lock, even in instances where multiple locks on a single data structure may be possible. For example, the tty device control block (Section 4.9) consists of three character buffers and a several configuration variables. A fine grained locking design might use a spinlock to protect each buffer and another spinlock to protect the configuration variables. In such instances, multicore XINU uses a single spinlock to protect an entire composite data structure. Medium spinlock granularity allows for a design that generally maximizes parallel efficiency without diminishing XINU's economy of mechanism.

4.1.2 XINU Invariants

Operating systems can be analyzed in terms of their *policies* and *mechanisms* [2,6]. Policies express desired system behavior at a high level. Mechanisms exist to enforce operating system policies. XINU expresses policies in clearly and concisely stated *invariants*. For example, the *scheduling invariant* states:

> At any time, the highest priority eligible process is executing. Among processes with equal priorities, scheduling is round-robin. [2]

Multicore XINU preserves XINU invariants as much as possible. In instances where invariants cannot be perfectly preserved, they are minimally amended.

For example, consider the scheduling invariant mentioned above. In a multicore XINU, multiple processes may execute in parallel. The scheduling invariant must expand to define the new scheduling capability. Preserving as much of the original invariant as possible, the multicore scheduling invariant pluralizes *process* to *processes*, as follows:

> At any time, the k highest priority eligible processes are executing, where k is the number of cores. Among processes with equal priorities, scheduling is round-robin.

4.1.3 XINU Levels

XINU kernel modules are conceptually organized into a multilevel software architecture. The lowest level consists of the platform hardware. Hardware is abstracted further and further away from successively higher levels, which use the services provided by lower levels. A level is comprised of a cohesive set of data object definitions, an a.p.i. that provides access to the services defined within the level, and XINU invariants that define the behavior within the level. Figure 4.1 shows the organization of XINU levels.

This chapter will explain multicore XINU in a bottom up fashion, starting with the hardware.

4.2 Hardware

Multicore XINU is built on top of three basic multicore operations: spinlocks, inter-processor interrupts, and core self-identification. Hardware-dependent implementations of each of the three are discussed below. As a consequence of XINU's multilevel software organization, any platform that supports the three capabilities can support multicore XINU.

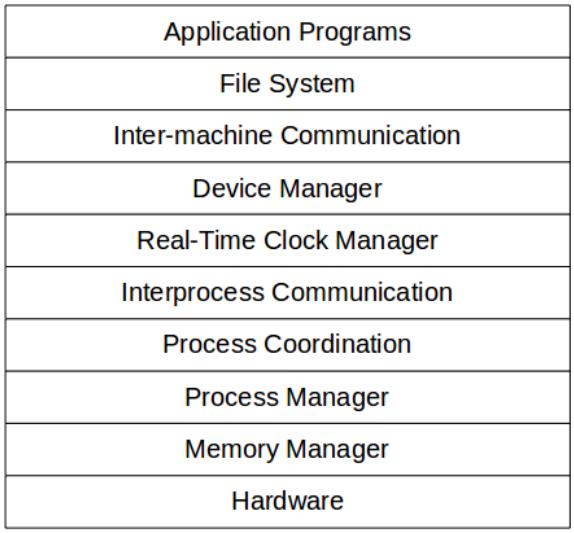

Fig. 4.1.: XINU's multilevel software architecture

An example ARM platform used herein is the H3 Allwinner system on the Orange Pi development board. The Orange Pi has a quad-core ARM cortex A7 processor with a 32 KB L1 instruction and data cache per core, and a 512 KB shared L2 cache. Various versions of the Orange Pi provide different devices and main memory capacities.

An example x86 platform is emulated with a virtual machine on QEMU and VirtualBox for multicore XINU. A XINU VM running on each emulator can be configured to use as many cores as the host machine can provide.

4.2.1 Spinlocks

The functions `spin_lock` and `spin_unlock` encapsulate hardware spinlocking on each platform.

On the ARM platform, hardware monitors in each L1 cache coordinate with each other, and with a global monitor in the L2 cache to implement the MOESI cache consistency protocol. The monitors track the state of entire cache lines, and work together to implement the ARM exclusive store instruction, `strex`. Multicore XINU

uses the ARM exclusive store instruction to implement a spinlock as outlined in [8]. Figure 4.2 shows ARM spinlock assembly code that implements a spinlock.

```
/*-----------------------------------------------------------------------
 * spin_lock  -  Lock a spinlock using arm exclusive monitors
 *-----------------------------------------------------------------------
 */
spin_lock:
                ldr r1, =LOCKED
spin:           ldrex r2, [r0]
                cmp r2, r1      /* Test if mutex is locked or unlocked  */
                beq spin        /* If locked, wait until released       */
                strexne r2, r1, [r0] /* Not locked, attempt to lock it  */
                cmpne r2, #1    /* Check if Store-Exclusive failed      */
                beq spin        /* Failed - retry                       */
                /* Lock acquired */
                dmb     /* Required before accessing protected resource */
                bx lr

/*-----------------------------------------------------------------------
 * spin_unlock  -  Unlock a spinlock using arm exclusive monitors
 *-----------------------------------------------------------------------
 */
spin_unlock:
                ldr r1, =UNLOCKED
                dmb     /* Required before releasing protected resource */
                str r1, [r0]    /* Unlock mutex */
                bx lr
```

Fig. 4.2.: ARM assembly spinlock implementation

The ARM spinlock assembly code waits until a memory location is unlocked and then continuously attempts to lock it with an exclusive store until successful. As discussed in Section 3.2.1, the `dmb` memory barrier instruction enforces memory consistency after a spinlock is acquired and before it is released.

The GNU C compiler contains compiler directives that can be used to implement the test_and_set algorithm. The x86 VM spinlock implementation leverages the directives as shown in Figure 4.3.

4.2.2 Inter-Processor Interrupts

The function `sendipi` sends an inter-processor interrupt to a target core. A core uses the `bcastipi` function to broadcast an inter-processor interrupt to all other cores. As discussed in Section 3.3.5, the GIC on the ARM platform and the APIC on the x86 platform are used to send inter-processor interrupts.

```
/*------------------------------------------------------------------------
 * spin_lock  -  Lock a spinlock using test and set
 *------------------------------------------------------------------------
 */
void spin_lock(volatile int* lockaddr){
    while (__sync_lock_test_and_set(lockaddr, 1)){
        while(*lockaddr);
    }
}

/*------------------------------------------------------------------------
 * spin_unlock  -  Unlock a spinlock
 *------------------------------------------------------------------------
 */
void spin_unlock(volatile int* lockaddr) {
    __sync_lock_release(lockaddr);
}
```

Fig. 4.3.: x86 spinlock implementation in C

4.2.3 Core Identification

Multicore XINU determines which core is currently executing a section of code with the `getcid` function. On the ARM platform, each core reads its ID from a per core co-processor register. On the x86 platform, each core has its own local APIC register from which it can read its ID

4.3 Lock Management

As mentioned previously, on a multicore platform spinlocks are required to protect access to kernel data structures. Spinlocks are organized in the same way as other XINU data objects. A global spinlock table tracks spinlock entry structures. A spinlock is accessed individually by its index in the table, known as its lock ID.

As mentioned in Section 4.1, XINU uses recursive disable/restore. Because spinlocks must be used to protect critical sections in multicore XINU, and critical sections can be nested, a lock that protects a critical section may be locked multiple times by the same process in nested critical sections. Section 3.1.4 introduced the re-entrant/recursive spinlock, which allows for such behavior. Multicore XINU uses recursive spinlocks. The multicore XINU recursive spinlock data structure is called

a "lock." A global table of lock entries tracks information about all XINU locks. A lock table entry, `lentry`, is shown in Figure 4.4.

```
/* Spinlock table entry: each entry must be equal to the size of the ERG
 * in order to prevent false sharing.
 */
struct lentry {
        volatile int32  lock;   /* Actual lock word used by hardware    */
        cid32   lowner;         /* id of cpu that currently owns lock   */
        uint32  lcount;         /* count locks by lowner                */
        int32   lpad[13];       /* padding to reach size of ERG         */
};
```

Fig. 4.4.: An entry in the global lock table

The `lock` member of each `lentry` is the actual memory word used by the `spin_lock` function. The `lownwer` and `lcount` members are used to implement recursive locking. The `lpad` array in each entry is used to pad the lock to the size of the ERG to prevent false sharing.

When a process acquires a lock with the `lock` function, it first checks `lowner` to see if the core on which it is running already owns the lock. If so, it increments `lcount`. Otherwise it calls the `spin_lock` function to acquire the `lock` memory word. When a process releases a lock with the `unlock` function, it decrements `lcount`. When `lcount` reaches 0, the process releases the `lock` memory word with the `spin_unlock` function.

XINU provides semaphores to higher levels, including the application level, for synchronization and mutual exclusion. As mentioned in the previous chapter, semaphores allow for a process to reschedule if necessary when attempting to access a shared resource. Spinlocks are only needed by the kernel to synchronize access to its internal data structures. As a consequence, there is no need to create or destroy spinlocks dynamically. Multicore XINU statically initializes all spinlocks during the boot sequence.

4.3.1 Lock Policies

Section 4.5.5 will explain the motivation for listing a core as a lock owner in the `lentry` data structure. However, when a process acquires a lock with the `lock` function, the matching call to `unlock` must eventually be made by the same process. Therefore, a process can conceptually be considered the owner of the lock.

Because other processes waste execution cycles while spinning on a spinlock, a lock owning process should release the lock as soon as possible after acquiring it. Two spinlock policies arise from this directive:

Lock Policy 1: No process should be interrupted while holding a lock.

If a process is interrupted while holding a lock, other processes will spin unnecessarily while the process holding the lock executes its interrupt service routine. Furthermore, if the process holding the lock is preempted by a timer interrupt, other processes that require the lock will wastefully spin until the lock holding process resumes execution. Because there is no guarantee that the lock holding process will ever resume execution, a deadlock could occur.

Lock Policy 2: No process should yield a core while holding a lock.

If a process yields a core while holding a lock, other processes attempting to acquire the lock will be forced to spin until the process holding the lock resumes execution.

4.3.2 X-Sections

Lock policy 1 can be enforced with the existing disable/restore paradigm. However, disabling interrupts will not prevent a process from voluntarily yielding the processor by rescheduling. XINU provides an existing function to temporarily defer rescheduling: `resched_cntl`. The function uses a defer count to support nested calls. Each time a process makes a nested call to defer rescheduling, `resched_cntl`

increments a defer count. Each time a process makes a call to resume rescheduling, the defer count is decremented. When the defer count reaches 0, `resched_cntl` triggers rescheduling if a rescheduling attempt was made. The nesting capabilitiy of `resched_cntl` makes it a suitable enforcer for lock policy 2.

Multicore XINU combines the two lock policies and their supporting mechanisms into the *x-section*[1] a.p.i. The function `xsec_beg()` marks the beginning of an x-section, and the complimentary `xsec_end()` function marks its end.

```
/*------------------------------------------------------------------------
 *  xsec_beg  -  Begin XINU kernel critical section
 *------------------------------------------------------------------------
 */
intmask xsec_beg(
                lid32 lid       /* id of spinlock to lock */
                ){
        intmask mask;   /* Saved interrupt mask */

        /* Disable interrupts to prevent handler deadlocks */
        mask = disable();

        /* Defer rescheduling so the process doesn't yield the processor
         * while holding a spinlock.
         */
        resched_cntl(DEFER_START);

        /* Lock the spinlock necessary to protect XINU critical section */
        lock(lid);

        return mask;
}

/*------------------------------------------------------------------------
 *  xsec_end  -  End XINU kernel critical section
 *------------------------------------------------------------------------
 */
 status xsec_end(
                intmask mask,   /* saved interrupt mask to restore */
                lid32 lid               /* id of spinlock to lock */
                ){

        /* Undo operations in reverse order of xsec_beg */
        unlock(lid);

        resched_cntl(DEFER_STOP);

        restore(mask);

        return OK;
}
```

Fig. 4.5.: The `xsec_beg` and `xsec_end` functions

Because the functions called within an `xsec` function (disable/restore, resched_cntl, and lock/unlock) all support nested calling, x-sections may be nested.

[1]short for XINU critical section

Table 4.2.: Example of nested x-sections

Stack Frame	Code Execution	Saved Interrupt Mask	Defer Count	Lock Count
top level	mask = xsec_beg(lock); func_a(); xsec_end(mask,lock);	enabled	0	0
func_a	mask = xsec_beg(lock); func_b(); xsec_end(mask,lock);	disabled	1	1
func_b	mask = xsec_beg(lock); /* x-section */ xsec_end(mask,lock);	disabled	2	2

Subsequent sections will illustrate scenarios in which it is necessary to acquire multiple spinlocks. Variadic versions of the `xsec_beg` and `xsec_end` functions, support multi-locking. The functions perform nested locking by acquiring a given list of locks in the given order and releasing the locks in reverse order to prevent deadlocks.

Ultimately, an x-section is a special kind of critical section in which interrupts are disabled, rescheduling is deferred, and one or more locks are held. If a rescheduling attempt is made within an x-section, actual rescheduling will be deferred and later triggered at the end of the x-section by the `resched_cntl` function. Algorithm 4.2 shows the typical use of an x-section. Table 4.2 shows an example nested x-section function call sequence.

Algorithm 4.2 typical use of an x-section

```
mask = xsec_beg(lock);
/* execute kernel critical section protected by lock */
xsec_end(mask, lock);
```

4.3.3 Deadlock Prevention with Nested Locking

As mentioned in Section 3.3.4, one way to prevent deadlocking is to acquire locks in a common nesting order. The nature of XINU's multilevel architecture is such that higher level functions generally call lower level functions but not vice versa. As a consequence, some nested locking is already accomplished. Locks in lower levels will be nested within locks in higher levels by the preexisting XINU system call architecture. Another locking policy follows:

> Lock Policy 3: Locks used in lower levels should be nested within locks used in higher levels.

This policy does not address multi-locking of locks that reside in the same level. A designated locking order must be assigned to locks within each level. This problem is solved with another locking policy:

> Lock Policy 4: Locks in the same level must be nested in the same predetermined order.

Strictly enforcing a globally agreed upon nested locking order ensures that deadlocks will not occur on spinlocks within the XINU kernel. To express a nested locking hierarchy, the '>' notation will be used hereafter. Specifically, if lock B should be nested within lock A, the mandated nesting order will be denoted as A > B.

4.3.4 X-Sections and Semaphores

Because x-sections defer rescheduling, a process cannot synchronize on a semaphore from within an x-section. In instances where semaphore synchronization cannot be avoided, explicit calls to the `lock` and `unlock` functions can be used, but the programmer must explicitly ensure that the lock holding process will not yield the processor. An example sequence that uses both locks and semaphores is shown in Algorithm 4.3. Semaphores are described in more detail in Section 4.6.

4.4 Memory Manager

In XINU, memory management is split between a High Level Memory Manager (HLMM) and a Low Level Memory Manager (LLMM) [2]. In the spirit of illustrating the multicore XINU bottom up, this section starts with the LLMM.

Algorithm 4.3 critical section that requires both locks and semaphores

```
mask = disable();
lock(lock);
/* execute kernel critical section protected by lock */
unlock(lock);
wait(semaphore); /* wait on a semaphore, potentially rescheduling */
lock(lock);
/* execute kernel critical section protected by lock */
unlock(lock);
restore(mask);
```

4.4.1 Low Level Memory Manager

The LLMM organizes memory as a linked list of blocks called the *free list*. The head of the list is stored globally. Processes use the LLMM to allocate arbitrarily sized sections of memory.

In multicore XINU, access to the free list is protected with a lock named the `memlock`. Where versions of the LLMM functions disable and restore interrupts, the multicore versions begin and end an x-section with the `memlock`.

4.4.2 High Level Memory Manager

The HLMM leases statically sized buffers to processes from preexisting buffer pools. Buffer pools are managed in a global table, `buftab`. Individual buffer pools are implemented as a linked list of buffers. A notable difference between the HLMM and the LLMM is that the HLMM uses synchronous memory allocation [2]. A counting semaphore tracks the number of available buffers left in a buffer pool. To allocate a buffer from a buffer pool, the function `getbuf` first waits on the counting semaphore until a buffer is available[2]. When a buffer is available, `getbuf` unlinks it from the

[2]Semaphores are described in more detail in Section 4.6

buffer pool list and returns it to the calling process. Conversely, the function `freebuf` returns a buffer to the pool by linking it back into the buffer pool list and then signaling the buffer pool's counting semaphore, allowing other processes to use it.

In multicore XINU, race conditions can be identified in two places: when multiple processes attempt to create a buffer pool in parallel and when multiple processes attempt to allocate or free a buffer in parallel. Thus, the `buftab` and each buffer pool must be protected by locks. This could be accomplished with a single lock on the buffer pool table. However, such coarse grained locking would needlessly prevent parallel access to separate buffer pools. Providing slightly finer lock granularity by assigning a spinlock to each buffer pool allows for parallel access to separate buffer pools.

Multicore XINU uses an x-section with a designated lock, the `buftablock`, to protect access to the global buffer pool table.

The functions `getbuf` and `freebuf` alter a buffer pool's linked list of buffers. To prevent race conditions on this linked list, each buffer pool is assigned a `lock`. Where `getbuf` and `freebuf` use `disable/restore` to achieve mutual exclusion, multicore XINU uses an x-section on the buffer pool's `lock`, with a notable difference: buffer pools are designed to be allocated synchronously. Processes must therefore be allowed to synchronize on a semaphore from within calls to `getbuf`. Under the `disable/restore` paradigm, blocking calls to `wait` can be made from within a critical section without issue. Section 4.3.4 explains why a call to `wait` that may result in rescheduling cannot be made from within an x-section. Fortunately, the solution to the problem is trivial. The x-section simply begins *after* the call to `wait`. Thus, the buffer pool's counting semaphore ensures that the number of process attempting to allocate a buffer in parallel will be no larger than the number of available buffers. The subsequent x-section on the buffer pool's spinlock ensures safe parallel access to the buffer pool's internal data structures.

4.5 Process Manager

The XINU Process Manager Layer is responsible for the creation, scheduling, and termination of processes. Processes are identified by their entries in a global process table. XINU uses a finite state machine (FSM) to maintain process states. Allowable transitions between process states are defined by state transition system calls. Figure 4.7 shows allowable state transitions in the Process Manager Layer.

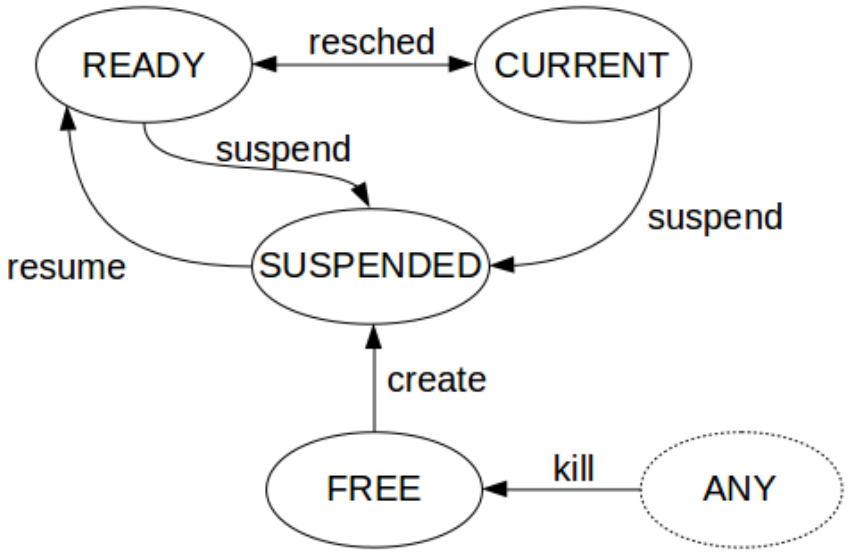

Fig. 4.6.: Allowable Process Manager state transitions in XINU

XINU uses a global ready queue to hold processes that are eligible for execution but are not currently executing. Chapter 3 analyzes the trade offs associated with the use of separate ready queues for each core, and points out that priority inversions are unavoidable in a per core queueing scheme. Recall that multicore XINU's scheduling invariant states:

> At any time, the highest priority eligible processes are executing. Among processes with equal priorities, scheduling is round-robin.

Priority inversions directly violate the scheduling invariant. Therefore, multicore XINU continues to use a single, shared ready queue.

4.5.1 Multiple Schedulers

The following XINU global data structures maintain scheduling state information:

- The `Defer` structure holds state information that allows the processor to defer rescheduling temporarily.
- The variable `currpid` holds the I.D. of the currently executing process.
- The variable `preempt` acts as a preemption counter. When the counter reaches 0, the currently executing process is preempted so that another process may run.

In order to run its own scheduler, each core requires its own instance of each of the global variables listed above. A new global table, the `cputab`, contains an entry with separate instances of these variables for each core. After the XINU boot sequence[3], no core ever accesses the `cputab` entry of another core. Therefore, protected access to the `cputab` is not necessary.

4.5.2 `create`

The `create` system calls accesses two shared, global data structures: the process table and a process entry in the table. A global `lock`, `proctablock`, protects access to the process table. Process table scanning is done within an x-section on the `proctablock`. To protect each process entry, multicore XINU assigns each process a `lock`. As `create` checks the state of each process, it first locks the process' `lock`. If the process is in the FREE state, `create` uses the process entry to create the new process. Otherwise, `create` unlocks the process `lock` and continues its scan.

Before returning, `create` unlocks the new process `lock` and then ends its x-section on the `proctablock`. Thus, the acquisition and release of each process' `lock` is nested within the acquisition and release of the `proctablock`: i.e. `proctablock` > process lock.

[3]Section 4.10 describes the multicore boot sequence in greater detail

4.5.3 `resume`

The `resume` system call moves a process from the SUSPENDED state to the READY state and places the process on the ready queue. The function accesses two shared, global data structures: the ready queue and the process entry. Both data structures must be locked to avoid race conditions. Process locks have already been introduced. The global `readylock` protects access to the ready queue.

Subsequent sections show that several operations remove a process from the head of a queue. Initially unaware of which process will be found at the head of the queue, these operations must *first* lock the queue to check which process is at the head of the queue, and *then* lock the process that is being removed. As a consequence, a new lock policy dictates the nested locking order between queues and processes:

Lock Policy 5. queue lock > process lock

Because `resume` is only defined for processes in the SUSPENDED state, `resume` begins by checking the state of the process. Only the process' `lock` must be locked to check the process' state. A naïve multicore implementation of `resume` might therefore start by acquiring the process' `lock`. However, if it finds the process in the SUSPENDED state, `resume` will then have to acquire the `readylock`. To adhere to `lock` policy 5, the naïve `resume` would have to release the process' `lock`, acquire the `readylock`, and then re-acquire the process' `lock`. The naïve `resume` would then have to check the state of the process *again* because the process' state may have changed after its lock was released.

In order to avoid such repetitive and cumbersome `lock` logic, the actual multicore XINU `resume` starts its x-section by acquiring both the `readylock` *and* the process' `lock` using the variadic version of `xsec_beg` described in Section 4.3.2.

The drawback of this locking style is that a kernel function may occasionally acquire a `lock` that it ends up not needing. For instance, if `resume` acquires the `readylock` and a process' `lock` and finds that the process is not in the SUSPENDED state, it will simply exit its x-section and return in error. Because the ready queue

Algorithm 4.4 cumbersome locking in a naïve multicore `resume` implementation

```
lock(process)
if process is SUSPENDED then
  /* enforce correct nesting order */
  unlock(process)
  lock(readylock)
  lock(process)
  /* process state may have changed */
  if process is SUSPENDED then
    ready(process)
    unlock(process)
    unlock(readylock)
    return OK
  else
    unlock(readylock)
  end if
end if
unlock(process)
return SYSERR
```

is not accessed in this case, it is unnecessary to acquire the `readylock`. However, because the `readylock` is held for such a short time, it is unlikely that it will cause notable contention. Multicore XINU employs a *lock once approach*, in which all necessary locks are acquired once at the beginning of an x-section.

> The lock once approach considers benefit of avoiding cumbersome lock/unlock/re-lock logic to outweigh the risk of very briefly acquiring a `lock` unnecessarily. Therefore, kernel functions acquire any locks that they may need one time in the correct nested order at the beginning of their x-sections.

Algorithm 4.5 simple multilocking in mulicore XINU `resume`

```
begin x-section with readylock and process lock
if process is SUSPENDED then
    ready(process)
    end x-section
    return OK
end if
end x-section
return SYSERR
```

4.5.4 `ready`

Because `ready` is called from kernel functions other than `resume`, it also begins an x-section with the `readylock` and the `lock` of the process being readied. Recall that nested x-sections are allowable because multicore XINU uses reentrant locking.

In multicore XINU, `ready` must check that a process is not in the FREE or DEAD[4] states before adding the process to the ready list. Multicore `ready` then adds the process to the ready queue and calls `resched` to enforce the scheduling invariant. However, a problem unique to multicore XINU appears at this point: because each core is running its own scheduler, forcing a single core to reschedule to the highest priority process on the ready queue is not sufficient to enforce the multicore XINU scheduling invariant.

Consider the following scenario: Process A is running on core 0 with priority 20. Process B is running on core 1 with priority 10. Process A adds a new process, C, with priority 15 to the ready queue and then reschedules. Because A's priority is higher than C's, process A will continue to run and leave C on the ready list. Because process C has a higher priority than process B on core 1, core 1 must reschedule in order to preserve the multicore scheduling invariant.

[4]Section 4.5.8 describes the DEAD process state.

In general, adding a process to the shared ready queue affects the schedulers running on every core. Therefore, every core must reschedule when a process is added to the ready queue. In multicore XINU, `resched` broadcasts an IPI to all other cores, causing them to reschedule. Because `ready` makes the call to `resched` from within an x-section, the calling core will not actually reschedule until it leaves its x-section.

4.5.5 `resched`

The `resched` function moves processes between the READY and CURRENT states. It cannot use an x-section, because rescheduling must be allowed: that is the whole point of `resched`. Multicore `resched` must therefore use locks to protect its critical section independent of the x-section a.p.i.

If the process at the head of the ready queue has a priority at least as great as the currently executing process, rescheduling must occur, leading to an interesting problem: the old process must context switch to the new process. In accordance with the second lock policy[5], the old process must release the locks it has acquired. However, the instant the old process releases these locks, other cores can alter the two processes that are switching in a way that invalidates the rescheduling. For instance, a process may be attempting to `suspend` the new process. When the old process releases the new process lock, the new process may suddenly be suspended, but the processor will context switch and begin running it anyway. Consequently, `resched` must hold its locks across the context switch and release them afterward. Therefore, `resched` is allowed to break the second lock policy[6]. This constraint provides the motivation for marking a core as a lock's `lowner` in the lock's table entry, rather than a process: While the I.D. of the currently executing process changes during a context switch, the I.D. of the currently executing core does not. Thus, a core may acquire locks, perform a context switch, and then release the locks.

[5]Lock Policy 2. No process should yield the processor while holding a lock.
[6]Fortunately, this is the only occurrence of a policy exception.

In order for the new process to release the previous process' lock, it must be able to retrieve the I.D. of the previous process. Thus, another consequence of the need to hold locks across a context switch arises: The I.D. of the previously executing process must be stored where the new process can retrieve it. Each entry in the global CPU table holds a variable called `prevpid` for this purpose. Before performing the context switch, the old process stores its I.D. in `prevpid` for the new process to retrieve later.

After its context switch, multicore `resched` still has some work to do. This work is encapsulated in the function `ctxsw_ret` (context switch return). Algorithm A.1 shows the full algorithm for `resched`.

4.5.6 `ctxsw_ret`

The `ctxsw_ret` function encapsulates necessary post conext switch clean up. It is called by a process that has just begun running after a context switch. If the previous process is in the DEAD state, `ctxsw_ret` frees the stack that was allocated for the previous process and sets its state to FREE. The `ctxsw_ret` function then releases the locks on the current and previous processes and the `readylock`.

It is not necessarily the case that the next process will resume execution in `resched`. In XINU, a newly created process begins execution at the start of its designated function the first time it is selected to run by the scheduler. In such cases, the newly created process does not have an opportunity to execute `ctxsw_ret`. To ensure that a newly created process will run the `ctxsw_ret` function, a pointer to `ctxsw_ret` is pushed onto each new process' stack just after the pointer to the process' main function. In multicore XINU, a newly created process executes `ctxsw_ret` the first time it runs. When it returns from `ctxsw_ret`, the newly created process then executes its main function.

4.5.7 `suspend`

As shown in Figure 4.7, a process may be suspended from either the CURRENT or READY states. Thus, to ensure mutual exclusion,the x-section in multicore `suspend` locks both the `readylock` and the process' lock. If the process is in the READY state, `suspend` removes it from the ready queue and places it in the SUSPEND state.

If the process is currently executing, `suspend` places the process in the SUSPENDED state. The `suspend` function then calls `resched` to select a new process to run. In multicore XINU, when a core sees that a process is in the CURRENT state, the process may be running *on a different core*. The core executing `suspend` must therefore determine whether the process is executing on another core by checking its process table entry. If so, multicore `suspend` sends an IPI that causes the core on which the process is currently executing to reschedule. Otherwise, multicore `suspend` can simply call `resched`.

4.5.8 `kill`

While other Process Manager functions manage strictly defined state transitions between narrow subsets of XINU process states, the `kill` function terminates a process in *any* non-FREE state.

For single-core XINU, `kill` is accomplished by removing the process from any data structure it might be part of, deallocating all resources associated with the process, and then marking the process' table entry as FREE.

In multicore XINU, `kill` is more complicated for several reasons. First, processes may no longer safely deallocate their own memory. Second, the process selected for termination may currently be running on another core. Finally, process manager data structure locks must now be acquired. Each of these problems is addressed below.

Processes may no longer deallocate their own memory.

A dying XINU process (i.e. a process that calls `kill` on itself) deallocates its own memory in the `kill` function. The deallocated memory includes the process' stack, on which the current call to `kill` is running. Therefore, until the call to `kill` returns, the dying process is executing with a stack that is considered re-allocatable by the memory manager. If another process were being created at that exact moment, the memory manager could allocate the stack to the new process. Both processes, the dying process and the newly created process, would then be running on the same stack memory, possibly leading to inconsistencies. This scenario is impossible, however, because `kill` prevents preemption with `disable/restore`. The dying process is therefore able to finish the call to `kill` before any other process gets a chance to allocate memory.

In multicore XINU, a process running on another core may reallocate the memory that a dying process has just freed. Therefore, processes may no longer deallocate their own memory in multicore XINU. However, a process must still be able to self terminate when it finishes running. To allow a processes to self terminate without deallocating their own memory, a new process state, the DEAD state, is used.

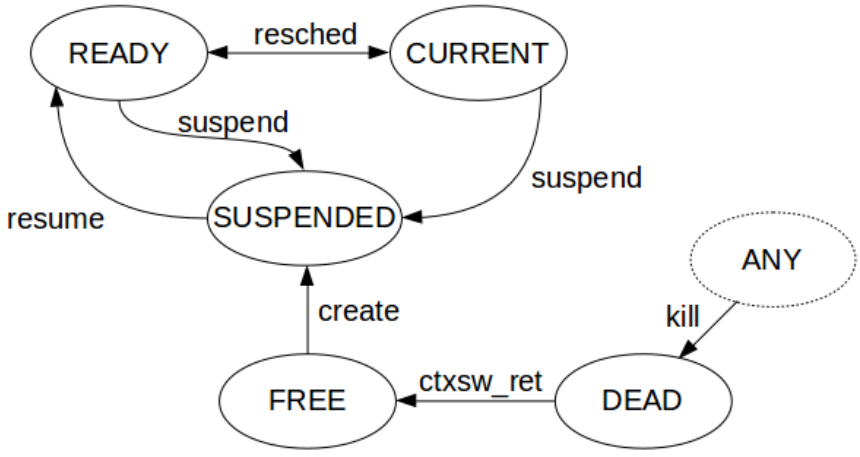

Fig. 4.7.: Mulicore XINU Process Manager state transitions with the DEAD state

In multicore XINU, a self terminating process places itself in the DEAD state. When the `kill` functions exits its x-section, the DEAD process calls `resched`. After the resulting context switch, the newly scheduled process executes `ctxsw_ret`, sees that the previous process is in the DEAD state, and frees the previous process' stack.

The process selected for termination may currently be running on another core.

To terminate a process that is currently running on another core, `kill` uses an inter-processor interrupt. Multicore core `kill` places the process in the `DEAD` state and then sends an IPI to the core on which the process is running, causing it to reschedule.

Process manager data structure locks must now be acquired.

Per the lock once approach and lock policy 5, multicore `kill` locks all queues where a process might be found before attempting to terminate the process. Because multicore `kill` must be able to terminate a process in *any* state other than FREE or DEAD, it must lock all of the queues associated with all possible process states before locking the process to determine its state. Locking all of these data structures temporarily prevents all scheduling on all cores. As a consequence, any core executing a kernel function that has nothing to do with the process being terminated (e.g. suspending or resuming an entirely different process) will needlessly block until the `kill` operation is complete. Such blocking is not ideal for efficiency, but it still allows for reasonably simple code that is guaranteed to be deadlock free via nested locking.

4.6 Process Coordination

4.6.1 Semaphores

Data structures and functions that implement counting semaphores in XINU comprise the Process Coordination Layer. As described in Chapter 3, a counting semaphore is a high level synchronization primitive that causes a processor to yield the processor if necessary and wait until it is allowed to proceed. When called on a semaphore, the `wait` system call places the calling process on a per semaphore queue in the WAITING state if the semaphore's count is less than or equal to 0. A WAITING process remains on a semaphore's queue until it is released by `signal` or until the semaphore is destroyed.

Like other core kernel data structures, XINU semaphores are tracked in a global semaphore table. Multicore XINU uses a global lock, the `semtablock`, to protect the global semaphore table. Because two or more cores may attempt to access a semaphore's internal data structures in parallel, multicore XINU semaphores each have their own lock. Multicore XINU replaces `disable/restore` with an x-section on a semaphore's lock and any other necessary locks.

4.6.2 Adding Multiple Processes to the Ready Queue

When a semaphore is deleted, reset, or signaled multiple times at once, multiple processes are transitioned from the semaphore's queue to the ready queue. The `semdelete`, `semreset`, and `signaln` functions add multiple processes to the ready queue by iterating over a semaphore's queue and calling `ready` on each process. The `ready` function triggers rescheduling. To prevent unnecessary, repetitive rescheduling, XINU defers rescheduling until all iterations are complete. In multicore XINU, each core has its own defer mechanism. When a core defers rescheduling, it only defers rescheduling for itself.

Therefore, the `semdelete`, `semreset`, and `signaln` functions must prevent unnecessary rescheduling on *all* cores by acquiring the `readylock` until they are finished executing. This prevents other cores from accessing the ready queue until all new processes have been transitioned from a semaphore's queue to the ready queue.

4.7 Process Communication

XINU allows for two kinds of interprocess communication: process to process message passing and inter-process communication ports.

4.7.1 Process to Process Message Passing

A XINU process may directly pass single word messages to another process by placing the word in a dedicated location in the receiving process' table entry. The receiving process can then retrieve the word directly. XINU processes use a blocking `receive` function to retrieve messages. Thus, if no message is present when a process calls `receive`, the process places itself in the RECEIVING state and reschedules. The `send` function places a RECEIVING process back on the ready queue.

XINU also offers a timed receive, `recvtime`, which times out if a message is not received within a certain time frame. A process performing a timed receive places itself on a queue of sleeping processes, called the sleep queue, in the TIMED_RECV state. The sleep queue is explained further in Section 4.8. For now, it suffices to say that it is protected by a new lock, called the `sleepqlock`.

The `send` function delivers a message to a process. If the target process is in the TIMED_RECV state, `send` moves it from the sleep queue to the ready queue. If the target process is in the RECEIVING state, `send` moves it to the ready queue. In observance of lock policies 3 (nested locking by layer) and 5 (queue > process), multicore `send` replaces `disable/restore` with an x-section on the `sleepqlock`, the `readylock`, and the lock of the target process.

A XINU process calling `receive` is guaranteed to be in the CURRENT state. In multicore XINU, another process on another core might terminate the receiving process, placing it in the DEAD state, just before the receiving process enters its x-section. Thus, a process executing multicore `receive` must check its own state before attempting to retrieve a message. This exposes an interesting consequence of the addition of the DEAD state in multicore XINU: a process might continue to execute after it has been terminated. Multicore XINU kernel functions must ensure that such a process does not do any *real* processing, and simply exits. Fortunately, XINU kernel functions generally check all process states in accordance with XINU's well defined process state transitions, so changes required by the addition of the DEAD state are minimal.

4.7.2 Inter-Process Communication Ports

A global table, `porttab`, holds information about ports. The `porttab` is trivially protected in multicore XINU with a new lock. A single XINU `port` consists of a bounded message queue, two semaphores, and several state variables. The sender semaphore blocks processes that attempt to add a message to a full port and the receiver semaphore blocks processes that attempt to remove a message from an empty port [2]. When one considers sending processes as message producers and receiving processes as message consumers, XINU ports provide an iconic example of the classical producer/consumer problem [2, 6, 9].

Critical sections in port functions are protected with disable/restore, which allows for use of XINU semaphores without issue. In multicore XINU, locks must be used to protect each port. Section 4.3.4 explained why kernel functions cannot safely `wait` on a semaphore from within an x-section: x-sections defer rescheduling and the `wait` function requires rescheduling to function correctly. Consequently, multicore XINU port related functions do not use x-sections. Instead, before accessing a port data

structure, they acquire the port's lock directly with the `lock` function and release it before any call to `wait`.

Ultimately, semaphores and locks work together in the producer consumer scheme. Semaphores ensure that only the appropriate number of producers or consumers are active while locks ensure correct and consistent access to shared data structures.

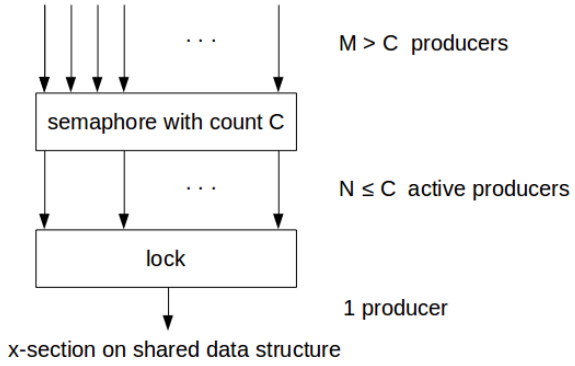

Fig. 4.8.: Multicore producers synchronized in the producer consumer problem

4.8 Real Time Clock Manager

XINU uses a platform's real time clock device to generate timer interrupts every millisecond. Timer events serve two purposes in XINU: triggering scheduling preemption when a process has completed its time slice and implementing delayed events.

4.8.1 Preemption

In unicore XINU, a global preemption counter is used to preempt a process at the end of its time slice. When a process becomes current, the preemption counter is set to a predetermined value, typically two milliseconds. The timer interrupt handler, called `clkhandler`, decrements this counter every millisecond when the timer generates an

interrupt. When the preemption counter reaches 0, `clkhandler` reschedules, giving the process at the head of the ready queue a chance to run if it has sufficient priority.

In multicore XINU, each processing core is running its own scheduler. Therefore, the CPU table contains a preemption counter for each core. Using the APIC on the x86 platform and the GIC on the ARM platform, the timer interrupt is routed to all cores. Each core decrements *its own* preemption counter and reschedules as needed.

4.8.2 Delayed Events

Processes that wish to delay for a predetermined period of time call `sleep` to place themselves on the global sleep queue in the SLEEPING state. During each timer interrupt, `clkhandler` checks the sleep queue and wakes any processes that are finished delaying, moving them to the ready queue and placing them in the READY state.

4.8.3 Asymmetry in `clkhandler`

In multicore XINU, access to the sleep queue is protected with a lock called the `sleepqlock`. All cores receive timer interrupts, but it is only necessary for one core to maintain the sleep queue. Multicore `clkhandler` is therefore implemented as an asymmetric interrupt handler. Only the main core with I.D. 0[7] performs sleep queue maintenance when executing `clkhandler`. Core 0 is also solely responsible for counting the seconds that XINU has been running.

4.9 Device Manager

The data for each XINU device is stored in a structure called a device control block. The data structures, function a.p.i., and interrupt handlers that manage a device comprise the device's driver. Device drivers are relatively large and complex

[7]checked with the `getcid` function (Section 4.2)

code modules with unique operational requirements. Therefore, the implications of operating each device on a parallel platform must be considered independently.

XINU includes device drivers for ethernet devices, remote and local file system devices, remote disk devices, and GPIO devices, to name a few. This section explains how XINU has been extended to support parallel access to its TTY driver, which provides basic user interaction via a terminal and command line.

4.9.1 TTY Driver

The TTY control block contains variables that hold configuration data, three I/O buffers (input, output, and echo), and input and output semaphores for each buffer. All of this shared data must be protected during parallel access. As with XINU ports (Section 4.7.2), the use of the blocking `wait` function with semaphores in TTY driver functions precludes the use of x-sections. Locks must be acquired directly via the `lock` function and locking policies must be explicitly enforced in the driver code.

The three buffers and their semaphores each constitute independent instances of the classical producer/consumer problem. As with ports, a lock on each queue would allow for mutual exclusion. However, yet another lock would then have to be used for protected access to the rest of the TTY control block. In many instances, two or more of these locks may be needed for a device driver operation. In such cases, the device programmer would have to be particularly careful to release all locks in the correct order before waiting on a semaphore. A predetermined lock nesting order would have to be established among all of these locks to prevent deadlocks. This fine locking granularity in a device driver very quickly introduces some non-trivial complexities.

Section 4.1.1 stated the intention of the multicore XINU design to use medium grained spinlocking for this very reason. Rather than add complex, fine grained spinlock management logic throughout its TTY driver, multicore XINU uses a single lock on the entire TTY control block. The benefit of a single lock on the control block is its economy of mechanism: The logic surrounding the use of a single lock is

easier to reason about, understand, implement. The down side of using a single lock lies in the potential for lock contention. Processes accessing entirely separate buffers in parallel my spin unnecessarily on the single, shared lock. Simplicity and elegance are key components of XINU's identity. Therefore, the simpler solution was chosen for multicore XINU.

4.10 Initialization

XINU begins operation with interrupts disabled. The boot sequence initializes interrupt handlers and all kernel data structures before enabling interrupts, creating and resuming several key system processes, and then falling into a busy loop to become the *null process*. The null process is a process that spins and does nothing. It has the lowest possible scheduling priority (0), and exists purely so that the processor will always have a process that is ready to run.

Multicore XINU begins running on a single core. Multicore XINU is therefore able to initialize all kernel data structures safely without acquiring any spinlocks. The main core then starts up the other cores. On the x86 platform, the main core sends inter-processor interrupts to other cores via the APIC. On the ARM platform, the main core starts other cores by writing to special CPU control registers.

Where XINU uses a single null process, multicore XINU running on n cores uses n null processes. Stack space for each null process is statically reserved in XINU memory. A newly awakened core sets its stack pointer to point to its reserved null stack.

The newly started core then performs any necessary self initialization, like enabling caches and configuring interrupts. When a new core has finished its self initialization, it calls `resched` to check for any available, higher priority processes. It then enables local interrupts and falls into its null process busy loop, using its dedicated bootstrapping stack to become a null process.

When there are no active processes, many SMP systems will temporarily put a core in a low power, sleeping state until an interrupt wakes them back up when there is more processing to do. However, XINU uses a busy waiting process, so the use of multiple null processes in multicore XINU is a an intuitive extension.

5. CONCLUSIONS

Chapter 3 walked through some important parallel programming concepts before cataloguing and analyzing several SMP operating system design patterns. This background provided context and a platform from which to discuss multicore XINU. Typical modern SMP operating systems employ a fine-grained, spinlocked kernel design. Multicore XINU takes a medium grained spinlocking approach.

Chapter 4 explained some key multicore XINU mechanisms, exposing and analyzing foundational principles that arise when extending an operating system to a multicore platform. Multicore XINU maintains XINU's semantics and key principles. To support multicore XINU, a platform must be capable of implementing spinlocks, inter-processor interrupts, and core self identification. Multicore XINU uses recursive locking, and upholds established locking policies with the x-sections. Nested locking prevents deadlocks. Medium grained locking and the lock once approach sacrifice a small amount of potential parallelism for economy of mechanism in core kernel functions. XINU regularly uses classical producer/consumer interactions and multicore XINU locks work harmoniously with semaphores to allow parallel producer and consumer execution while protecting shared data structures.

5.1 Remaining Levels

A diligent reader may have noticed that the Intermachine Communication and File System kernel levels were left out of Chapter 4, and that only a single device driver was presented. As alluded to in Section 4.9, device drivers are large and complex software modules for which multicore support must be engineered on a case by case basis. The ethernet driver is particularly important to subsequent levels that require network communication. Remaining device drivers and levels are a work in

progress. X-section methodology will provide a convenient framework for appending the remaining levels, and the lock per control block precedent established in the multicore TTY driver will guide the development of other multicore device drivers. As more device drivers are brought into multicore XINU, design decisions can be made about how to distribute device interrupt handling among cores.

REFERENCES

REFERENCES

[1] A. Grama, A. Gupta, G. Karypis, and K. Vipin, *Introduction to Parallel Computing*, 2nd ed. Harlow, England: Addison-Wesley, 2003.

[2] D. Comer, *Operating System Design: The Xinu Approach*, 2nd ed. New York: CRC Press, Taylor & Francis Group, 2015.

[3] B. Moyer, *Real World Multicore Embedded Systems: A Practical Approach*, ser. Expert Guide. Oxford: Newnes, 2013.

[4] K. Wang, *Embedded and Real-Time Operating Systems*. New York: Springer International Publishing, 2017.

[5] R. Love, *Linux System Programming*, 2nd ed. Beijing: O'Reilly, 2013.

[6] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts Essentials*. Hoboken, New Jersey: John Wiley & Sons, Inc., 2014.

[7] "Intel 64 and IA-32 Architectures Software Developers Manual," *Volume 3B: System Programming Guide*, 2011.

[8] "ARM Synchronization Primitives," 2009, http://infocenter.arm.com/help/topic/com.arm.doc.dht0008a/DHT0008A_arm_synchronization_primitives.pdf.

[9] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Amsterdam: Elsevier/Morgan Kaufmann, 2008.

[10] N. Chong and S. Ishtiaq, "Reasoning about the ARM weakly consistent memory model," in *Proceedings of the 2008 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness: held in conjunction with the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*. ACM, 2008, pp. 16–19.

[11] P. E. McKenney, "Memory barriers: A hardware view for software hackers," *Linux Technology Center, IBM Beaverton*, 2010.

[12] C. Schimmel, *UNIX Systems for Modern Architectures : Symmetric Multiprocessing and Caching for Kernel Programmers*, ser. Addison-Wesley Professional Computing Series. Reading, Mass.: Addison-Wesley, 1994.

[13] A. Starke, "Locking in OS kernels for SMP systems," in *Seminar on Hot Topics in Operating Systems*, Technische Universität, Berlin, 2006.

[14] S. C. Kim and S. Lee, "Decentralized task scheduling for a fixed priority multicore embedded RTOS," *Computing*, vol. 97, no. 6, pp. 543–555, Jun. 2015.

[15] V. Kazempour, A. Fedorova, and P. Alagheband, "Performance implications of cache affinity on multicore processors," *Euro-Par 2008 Parallel Processing*, pp. 151–161, 2008.

[16] D. Comer, *Essentials of Computer Architecture*, 2nd ed. New York: CRC Press, Taylor & Francis Group, 2017.

[17] A. Downey, *The Little Book of Semaphores*. Needham, MA: CreateSpace Independent Publishing Platform, 2016.

[18] J. Mistry, M. Naylor, and J. Woodcock, "Adapting FreeRTOS for multicores: an experience report." *Software: Practice and Experience*, vol. 44, no. 9, pp. 1129–1154, Sep. 2014. [Online]. Available: http://doi.wiley.com/10.1002/spe.2188

[19] C. Von Praun, H. W. Cain, J.-D. Choi, and K. D. Ryu, "Conditional memory ordering," in *ACM SIGARCH Computer Architecture News*, vol. 34. IEEE Computer Society, 2006, pp. 41–52.

[20] J. Mistry, *FreeRTOS and Multicore*, 2011, University of York, Master's Thesis.

APPENDIX

A. APPENDIX

Table A.1.: Multicore XINU Lock Policies

	Policy	Enforcing Mechanism
1	No process can be interrupted while holding a lock.	{disable/restore}
2	No process can reschedule while holding a lock.	Deferred Rescheduling
3	To avoid deadlocks, lower level locks must be nested within upper level locks.	Multilevel Architecture
4	To avoid deadlocks, locks within a level will follow a predetermined nesting order.	OS Programmer
5	Process locks must be nested within queue locks.	OS Programmer

Table A.2.: Allowable XINU process states and their meanings.

State	Meaning
FREE	Process table entry is unused.
CURRENT	Process is currently running.
READY	Process is on ready queue.
RECV	Process waiting for message.
SLEEP	Process is sleeping.
SUSP	Process is suspended.
WAIT	Process is on semaphore queue.
RECTIM	Process is receiving with timeout.
DEAD	Process has terminated but its resources have not yet been deallocated.

Algorithm A.1 multicore `resched`

if rescheduling deferred *on this core* **then**
  record that attempt was made
  return
end if
lock(`readylock`)
lock(current process)
if current process is still eligible (in the CURRENT state) **then**
  if current process is still highest priority **then**
    unlock(current process)
    unlock(`readylock`)
    return
  end if
  {current process will not remain current}
  set process state to READY and insert in the ready queue
end if
set `prevpid` *for this core* to the ID of the current process
dequeue process from head of ready list and make it current *on this core*
reset time slice for new process and update process entries
context switch between old and new processes
perform `ctxsw_ret` {return from context switch}
return

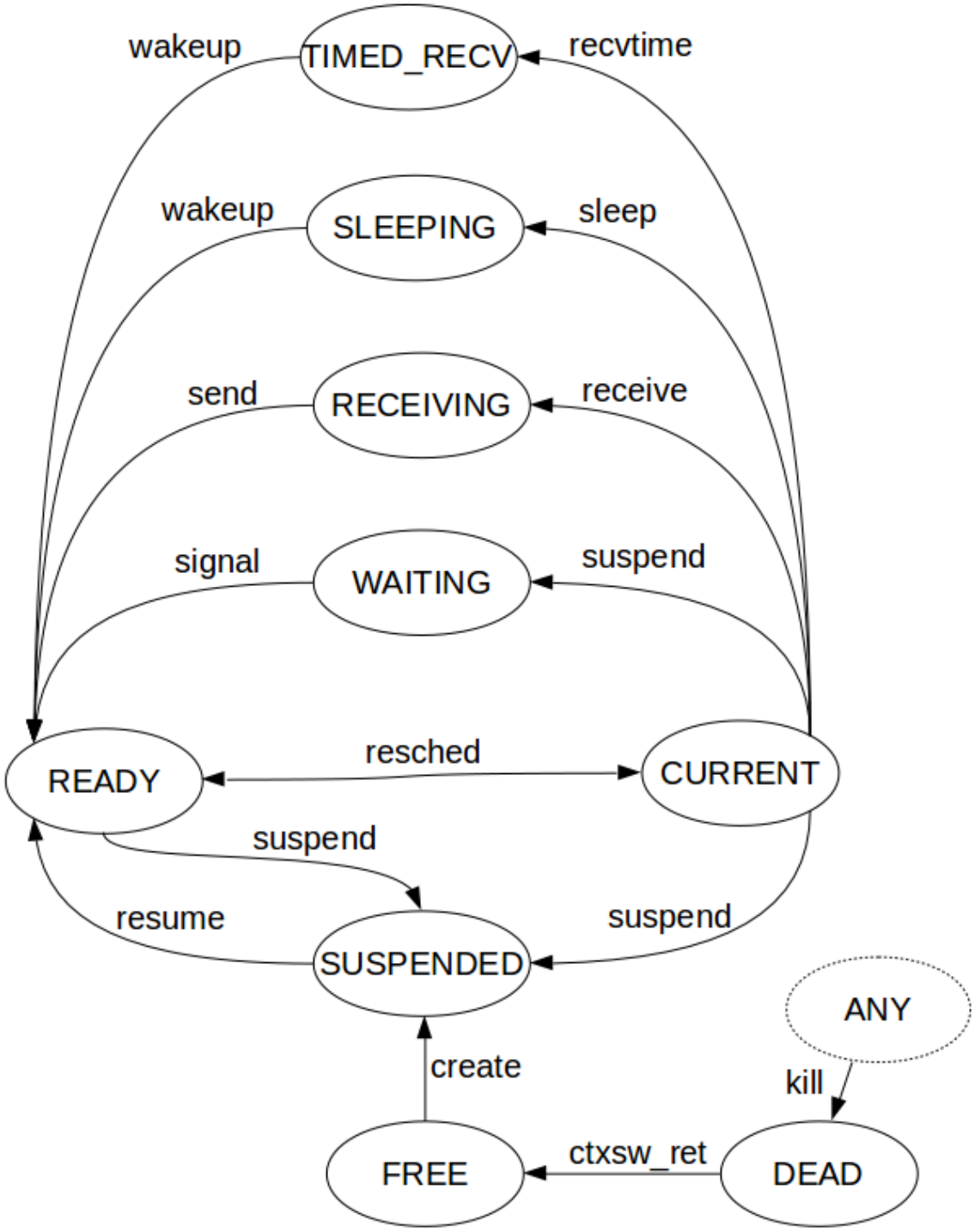

Fig. A.1.: The full multicore XINU process state machine