Efficient and Predictable Thread Synchronization Mechanisms for Mixed-Criticality Systems on Shared-Memory Multi-Processor Platforms

by
Alexander Züpke

Approved Dissertation thesis for the partial fulfillment of the requirements for a
Doctor of Natural Sciences (Dr. rer. nat.)
Fachbereich 4: Informatik
Universität Koblenz-Landau

Date of the doctoral viva: December 18, 2020.

Abstract

Real-time operating systems for *mixed-criticality systems* must support different types of software, such as real-time applications and general purpose applications, and, at the same time, must provide strong spatial and temporal isolation between independent software components. Therefore, state-of-the-art real-time operating systems focus mainly on *predictability* and *bounded worst-case behavior*. However, general purpose operating systems such as Linux often feature more efficient—but less deterministic—mechanisms that significantly improve the average execution time. This thesis addresses the combination of the two contradicting requirements and shows *thread synchronization mechanisms* with efficient average-case behavior, but without sacrificing predictability and worst-case behavior.

This thesis explores and evaluates the design space of *fast paths* in the implementation of typical blocking synchronization mechanisms, such as mutexes, condition variables, counting semaphores, barriers, or message queues. The key technique here is to *avoid unnecessary system calls*, as system calls have high costs compared to other processor operations available in user space, such as low-level atomic synchronization primitives. In particular, the thesis explores *futexes*, the state-of-the-art design for blocking synchronization mechanisms in Linux that handles the uncontended case of thread synchronization by using atomic operations in user space and calls into the kernel only to suspend and wake up threads. The thesis also proposes *non-preemptive busy-waiting monitors* that use an efficient *priority ceiling mechanism* to prevent the *lock holder preemption problem* without using system calls, and according low-level kernel primitives to construct efficient *wait* and *notify* operations.

The evaluation shows that the presented approaches improve the average performance comparable to state-of-the-art approaches in Linux. At the same time, a worst-case timing analysis shows that the approaches only need constant or bounded temporal overheads at the operating system kernel level. Exploiting these fast paths is a worthwhile approach when designing systems that not only have to fulfill real-time requirements, but also best-effort workloads.

Zusammenfassung

Echzeitbetriebssysteme für *Systeme mit gemischten Kritikalitäten* müssen unterschiedliche Arten von Software, wie z.B. Echtzeitanwendungen und Allzweckanwendungen, gleichzeitig unterstützen. Dabei müssen sie eine solide räumliche und zeitliche Isolation zwischen unabhängigen Softwarekomponenten bieten. Daher fokussieren sich aktuelle Echtzeitbetriebssysteme hauptsächlich auf *Vorhersagbarkeit* und ein *berechenbares Worst-Case-Verhalten.* Allerdings bieten Allzweck-Betriebssysteme wie Linux häufig effizientere, aber weniger deterministische Mechanismen, welche die durchschnittliche Ausführungszeit signifikant erhöhen. Diese Thesis befasst sich mit der Kombination der beiden gegensätzlichen Anforderungen und zeigt *Mechanismen zur Thread-Synchronisation* mit einem effizienten Durchschnittsverhalten, ohne jedoch die Vorhersagbarkeit und das Worst-Case-Verhalten zu beeinträchtigen.

Diese Thesis untersucht und bewertet den Entwurfsraum von *Abkürzungen* (engl. *fast paths*) bei der Umsetzung von typischen blockierenden Synchronisationsmechanismen wie Mutexen, Bedingungsvariablen, Zähl-Semaphoren, Barrieren oder Nachrichtenwarteschlangen. Der Ansatz ist dabei, *unnötige Systemaufrufe zu vermeiden.* Systemaufrufe haben im Vergleich zu anderen Prozessoroperationen, die im Benutzermodus verfügbar sind, wie z.B. atomaren Operationen, höhere Kosten. Insbesondere erforscht die Thesis *Futexe*, ein aktuelles Design für blockierende Synchronisationsmechanismen in Linux, welches den konkurrenzfreien Fall der Synchronisierung mithilfe atomarer Operationen im Benutzermodus löst und den Kern nur aufruft, um Threads zu suspendieren und aufzuwecken. Die Thesis untersucht auch *nicht-unterbrechbare Monitore mit aktivem Warten.* Dort wird ein effizienter Mechanismus mit Prioritätsschranken verwendet, um das sogenannte *Lock-Holder-Preemption-Problem* ohne Systemaufrufe zu vermeiden. Ebenfalls werden passende niedere Kernprimitive beschrieben, die effiziente *Warte-* und *Benachrichtigungsoperationen* ermöglichen.

Die Evaluation zeigt, dass die vorgestellten Ansätze die durchschnittliche Leistung vergleichbar zu aktuellen Ansätzen in Linux verbessern. Gleichzeitig zeigt eine Analyse des Worst-Case-Zeitverhaltens, dass die Ansätze nur konstante oder begrenzte zeitliche Mehraufwände auf der Ebene des Betriebssystemkerns benötigen. Die Nutzung dieser Abkürzungen ist ein lohnender Ansatz für den Entwurf von Systemen, die nicht nur Echtzeitanforderungen erfüllen, sondern auch Allzweckanwendungen gut unterstützen sollen.

Contents

1 Introduction **1**
1.1 Efficient Synchronization Mechanisms 2
1.2 Predictability and Determinism 3
1.3 Contributions 3
1.4 Organization 5

2 Basics **7**
2.1 Basic Concepts and Terminology 7
2.1.1 Operating System Concepts 7
2.1.2 Shared Resources and Critical Sections 9
2.1.3 The Lock Holder Preemption Problem 10
2.1.4 Real-Time Scheduling 11
2.1.5 Real-Time Locking Protocols 13
2.1.6 WCET Analysis Methods 19
2.1.7 Operating System Architectural Concepts 20
2.1.8 Processor Architecture 26
2.1.9 Predictability Issues on Multicore Processors 28
2.2 User-Level Synchronization Mechanisms 28
2.2.1 Spinlocks 30
2.2.2 Mutexes 31
2.2.3 Condition Variables 32
2.2.4 Reader-Writer Locks 33
2.2.5 Counting Semaphores 34
2.2.6 Barriers 35
2.2.7 One-Time Initializers 35
2.2.8 Queuing Ports and Buffers 36
2.2.9 Sampling Ports and Blackboards 36
2.2.10 Events 37
2.2.11 Futexes 39
2.2.12 Adaptive Mechanisms 40
2.2.13 Waiting With Timeouts 41
2.3 Synchronization Inside an Operating System Kernel 42
2.3.1 Level of Indirection 42
2.3.2 Queuing 42
2.3.3 Locking and Preemption Control 43
2.3.4 Waiting and Wake-up 44

3 Analysis **45**
3.1 Related Work 46
3.1.1 Futexes and Fast Synchronization Mechanisms 46

3.1.2 Lazy Techniques and Optimization 47
3.2 Settings and Requirements 48
3.2.1 System Model 48
3.2.2 Relative Costs of CPU Instructions 50
3.2.3 Requirements for Predictability 51
3.2.4 Metrics to Evaluate Efficiency and Predictability 54
3.3 Building Blocks of Real-Time Locking Protocols 55
3.4 Building Blocks of User-Level Blocking Synchronization 57
3.4.1 High-Level Analysis and Generalization 58
3.4.2 Mapping to Low-Level Building Blocks in the Kernel . . . 62
3.5 Futexes in Linux Revisited 67
3.5.1 Conceptual OS Implementation of Futexes 67
3.5.2 Linux Futex Implementation 69
3.5.3 Issues with Futexes in Linux 72
3.5.4 Safety and Security Aspects of Futexes 76
3.5.5 Summary 77
3.6 Alternative Approaches for Efficient Synchronization Mechanisms 77
3.7 Preemption Control Mechanisms 81
3.7.1 Related Work on Lock Holder Preemption 81
3.7.2 Preemption Control inside OS Kernels 83
3.7.3 Building Blocks of Preemption Control Mechanisms 84
3.7.4 Temporary Non-Preemption in Symunix II 85
3.7.5 Fast IPCP Implementation by Almatary et al. 87
3.7.6 Discussion 90
3.8 Low-Level Wait and Wake-up Mechanisms 90
3.9 Analysis Summary 93

4 Design 96
4.1 Deterministic Futexes 96
4.1.1 Design Considerations 97
4.1.2 Binary Search Trees 99
4.1.3 Address Tree Management 99
4.1.4 Wait Queue Management 101
4.1.5 Preemptible Operation 102
4.1.6 Interference of Shared Futexes 103
4.1.7 Summary 104
4.2 Static Futexes 104
4.2.1 Design Considerations 104
4.2.2 Futex Operations for ARINC 653 in AUTOBEST 106
4.3 Higher-Level Synchronization Mechanisms based on Futexes . . . 107
4.3.1 Blocking Mutexes 108
4.3.2 Condition Variables 109
4.3.3 Counting Semaphores 112

4.3.4 Barriers . . . 113
4.3.5 One-time initializers . . . 114
4.3.6 ARINC 653 Synchronization Mechanisms . . . 115
4.4 Non-Preemptive Busy-Waiting Monitors . . . 117
4.4.1 Efficient IPCP Protocols . . . 117
4.4.2 Light-Weight Blocking for IPCP . . . 121
4.4.3 Monitor Synthesis . . . 126
4.5 Higher-Level Synchronization Mechanisms based on Monitors . . . 128
4.5.1 Data Model of Monitor-Based Synchronization Mechanisms 128
4.5.2 Blocking Mutexes . . . 129
4.5.3 Condition Variables for Blocking Mutexes . . . 130
4.5.4 Low-Level Monitor API . . . 131

5 Evaluation **134**
5.1 Validating Assumptions . . . 135
5.2 Performance Measurements . . . 137
5.2.1 Specifics of Deterministic Futexes . . . 138
5.2.2 IPCP Performance . . . 141
5.2.3 Comparison for Different Mutex Implementations . . . 142
5.3 Analysis of Worst-Case Timing Behavior . . . 146
5.3.1 Reference Architecture for Analysis . . . 147
5.3.2 Analysis of Kernel Primitives . . . 149
5.3.3 Analysis of Mutexes and Condition Variables . . . 158
5.4 Evaluation Summary . . . 165

6 Discussion **169**
6.1 Design Space Exploration . . . 169
6.2 Futexes . . . 172
6.2.1 Deterministic Futexes . . . 172
6.2.2 Static Futexes . . . 175
6.2.3 Practicality of Futexes . . . 176
6.3 Monitors . . . 179
6.3.1 Efficient IPCP Protocols . . . 179
6.3.2 Light-Weight Monitors . . . 180
6.3.3 Practicality of Monitors . . . 182
6.4 Comparison of Futexes and Monitors . . . 184
6.5 Fast Paths and Real-Time Protocols . . . 186
6.6 Discussion of Contributions . . . 187

7 Conclusion **193**

Bibliography **195**

List of Abbreviations

ABA problem	The *ABA problem*
ACET	Average-Case Execution Time
ALU	Arithmetic Logic Unit
API	Application Programming Interface
ARINC 653	Avionics Application Standard Software Interface
ASIL	Automotive Safety Integrity Levels
AUTOSAR	Automotive Open System Architecture
AVL trees	Binary search trees proposed by Adelson-Velsky and Landis
BCET	Best-Case Execution Time
BPF	Berkeley Packet Filter
BST	Binary search tree
CAS	Compare-And-Swap
CLH locks	Locks proposed by Craig, and Landin and Hagersten
CPU	Central Processing Unit
DFP	Deadline Floor Inheritance Protocol
DPCP	Distributed Priority-Ceiling Protocol
EAL	Evaluation Assurance Levels
ECU	Electronic Control Unit
EDF	Earliest-Deadline First
FIFO	First-In First-Out
FMLP	FIFO Multiprocessor Locking Protocol
FP	Fixed-Priority
FPU	Floating Point Unit
Futex	Fast User Space Mutex
I/O	Input/Output
IPC	Inter Process Communication
IPCP	Immediate Priority-Ceiling Protocol
IRQL	Interrupt Request Level in Windows NT
ISR	Interrupt Service Routine
JVM	Java Virtual Machine
LL/SC	Load-Linked / Store-Conditional
MCS locks	Locks proposed by Mellor-Crummey and Scott
MMU	Memory Management Unit
MPCP	Multi-processor Priority-Ceiling Protocol
MPU	Memory Protection Unit
MrsP	Multiprocessor Resource Sharing Protocol
Mutex	Mutual Exclusion
MSRP	Multiprocessor Stack Resource Policy
NPCS	Non-Preemptive Critical Sections

LIST OF ABBREVIATIONS

OS	Operating System
OSEK	Open Systems and Interfaces for Electronics in Motor Vehicles
PCP	Priority-Ceiling Protocol
P-FP	Partitioned Fixed-Priority
PI	Priority Inheritance
PIP	Priority Inheritance Protocol
plist	priority-sorted linked list (Linux data structure)
POSIX	Portable Operating System Interface
RMS	Rate-Monotonic Scheduling
RPC	Remote Procedure Call
RTOS	Real-Time Operating System
SWaP	Size, Weight and Power
SIL	Safety Integrity Level
SRP	Stack Resource Policy
TAS	Test-And-Set
TCB	Thread Control Block
TDMA	Time Division Multiple Access
TLB	Translation Look-aside Buffer
TLS	Thread Local Storage
TOCTTOU	Time-of-Check to Time-of-Use
TOWTTOS	Time-of-Wake-up to Time-of-Scheduling
WCET	Worst-Case Execution Time

Chapter 1

Introduction

The trend of integrating multiple functions into a single computer system due to *size, weight and power* (SWaP) considerations, as well as the rise of on-chip multicore processors, bring new challenges to research in embedded operating systems [BD19]. For such systems, the concept of *mixed criticality* defines not only the idea of higher relative importance of some functions compared to others, but also that these functions have different requirements regarding safety and security certification standards. Using safety-critical, mission-critical, and non-critical software components on a single computer requires a system design that enforces *strict separation* of these components [BBB^{+}09].

The way these systems are designed follows the specifications of domain standards, such as *ARINC 653* in avionics, which requires *temporal* and *spatial partitioning* mechanisms to ensure the separation [AEE15]. Partitioning by itself does not only ensure necessary *fault isolation* as required by safety standards, like *RTCA DO-178C* for avionics and *ISO 26262* for the automotive industry, but also allows for *independent analysis* of the different software components. This simplifies the design phase of such systems and allows to exchange or update software components to a certain degree.

However, the strong focus on *predictable timing* and *bounded worst-case behavior* often neglects the demands of non-real-time software components, which can have a vastly different and much more dynamic program structure. For example, best-effort applications do not have the limitation to allocate and initialize all resources at start time, as dynamic memory allocation is a source of non-determinism. Also, non-real-time operating systems and execution environments provide state-of-the-art techniques to optimize the *average-case performance* of both the applications and the whole system. Examples include *opportunistic* mechanisms like *overcommit* of virtual memory allocations, *biased locking* in a Java Virtual Machine that pins shared objects without contention to the recent thread [Kaw05], or *fair process scheduling* to prevent starvation.

1.1 Efficient Synchronization Mechanisms

When looking at mutual exclusion, there are different mechanisms that optimize for the average-case in thread synchronization mechanisms in user space.

For example, the design of *fast user space mutexes (futexes)* in Linux is based on the two observations that contention is usually rare and that system calls are expensive. Here, both `mutex_lock` and `mutex_unlock` operations feature a *fast path* based solely on atomic operations in user space, skipping system call overhead most of the time and only calling the kernel to actually suspend or wake up threads on contention [FRK02]. But futexes have not been developed for real-time systems or mixed-criticality systems in the first place and predictability issues have been raised [Bro16, ZK19].

Another example is the use of *spinlocks* for short critical sections in user space, i.e. busy-waiting synchronization based on atomic primitives of the processor. But using spinlocks in user space introduces a source of non-determinism due to *lock waiter preemption* and *lock holder preemption.* Various techniques to address these problems were proposed, e.g. *preventing* preemption while holding a lock, *recovery* from preemption by spinning only for a limited time, or *mitigating the side effects* of preemption which violate either scheduling or fairness constraints [ELS88, KWS97, MS98, ULSD04, OL13].

When we compare these examples, we can observe that the main technique to achieve better average execution time is to split a synchronization mechanism into an *efficient fast path* and a *robust slow path* or *fallback mechanism.* The fast path *avoids system calls*, as calls into the operating system kernel are expensive. The slow path is always implemented as system call and deals with the parts that require involvement of the operating system anyway, i.e. suspension and wake-up of threads. Under the *optimistic* assumption that the slow path is rarely needed, the usage of the fast path dominates and the average execution time improves.

However, as we will present in Chapter 3, avoiding system calls comes at a cost. For example, futexes in their originally proposed form are prone to unbounded loops when the blocking test in the kernel fails [FRK02]. And mechanisms to deal with lock-holder preemption correctly include watchdog mechanisms, superfluous spinning, or yielding execution time to a preempted lock-holder, see Section 2.1.3. Even worse, these additional overheads must be accounted with their worst-case impact when doing a worst-case execution time analysis.

For synchronization mechanisms, this thesis addresses the fundamental concern of whether efficient average-case performance and predictable real-time behavior can co-exist, as state-of-the-art synchronization mechanisms designed for best-effort systems choose efficiency over determinism. Synchronization mechanisms for mixed-criticality systems should be both *deterministic*, i.e. must not fail unexpectedly and have predictable temporal behavior, and *efficient*, i.e. avoid costly system calls.

1.2 Predictability and Determinism

In this thesis, the terms *predictability* and *determinism* describe the following different properties of software.

We use the term *predictability* in the sense that software shows *predictable timing* and *predictable resource usage*. In our specific meaning as temporal property, all parameters that contribute to the execution time of software are *identified* and we can construct a timing model, and, if all these parameters are *known*, either exactly or via an upper bound, we can compute the *worst-case execution time*. Similarly, for resource usage, we can construct a model when all depending parameters are defined, and we can derive the worst-case resource usage if these parameters are known. This follows the common usage of *predictability* in the real-time context [WEE+08, AEF+14].

In contrast, determinism describes the functional property to deliver the expected output for a given input in a given time. In the context of real-time operating systems, the term *determinism* is used in a more colloquial sense of "something behaves as intended, unexpected things must not going to happen" instead of the more formal sense used in theoretical computer science.

This thesis analyzes the impact of techniques to improve efficiency of synchronization mechanisms on the timeliness in a real-time system. In our particular case, the key technique is to use a fast path with lower average costs, and a fallback or slow path mechanism with higher average costs. Determinism ensures that the results must be the same, regardless if the fast path or the slow path was taken. Predictability ensures that, for both cases, non-conflicting timing models exist, and actual worse cases can be determined from input parameters when necessary. This requires that the timing model of the fast path must be included in the model of the slow path, so the slow path defines the worst case, see also Section 3.2.4.

1.3 Contributions

In this thesis, we provide synchronization mechanisms for mixed-criticality systems which can handle both sides—best-effort *and* real-time—well. The proposed mechanisms provide good average-case performance, but without sacrificing predictability and worst-case behavior. This does not only help best-effort applications. Also real-time applications benefit from a faster average performance, as it helps to achieve secondary design goals, such as reducing the overall power consumption.

The application-specific context of this thesis is to provide the higher-level blocking synchronization primitives of POSIX PSE51, ARINC 653 part 1, and AUTOSAR OS in an efficient way. These operating system standards cover the domains of industrial, avionics, and automotive software systems with different degrees of real-time requirements. For this, the thesis analyzes existing state-of-the-art mechanisms for efficient synchronization, in particular futexes, for the usability in real-time environments, and then extends the state-of-the-art with designs which address the problems of these best-effort approaches. In Linux and other operating systems, futexes are the underlying synchronization primitives to implement POSIX synchronization mechanisms, such as blocking mutexes, condition variables, semaphores, and barriers.

The thesis provides the following contributions:

- The *analysis* decomposes blocking synchronization mechanisms into low-level building blocks, identifies potential fast paths, and discusses the trade-offs of moving these building blocks beyond the classical barrier between the operating system kernel and user space.

- *Deterministic futexes* address the determinism and predictability issues found in the Linux futex design. Deterministic futexes provide similar features as futexes in Linux and target similar use cases.

- *Static futexes* are a futex design for statically configured systems and resource constrained embedded systems. Static futexes provide a selected subset of the futex features found in Linux by using only a minimal implementation at the kernel level.

- *Fast priority switching* is an efficient implementation of an immediate priority ceiling protocol (IPCP) without using system calls in the average case and can be used to temporarily disable preemption in user space. This enables non-preemptive critical section in user space.

- *Non-preemptive busy-waiting monitors*[1] allow synchronization mechanisms with richer semantics than futexes in user space. Key techniques here are fast priority switching and optimized wait and wake-up primitives in the operating system kernel which reduce the number of system calls to the required minimum.

For each presented mechanism, the thesis provides an analysis of worst-case behaviors and the impact on a timing analysis. For this, we do not perform an actual WCET analysis, as this requires detailed knowledge of the underlying processor architecture and the overall system, see e.g. [BSC$^+$11]. Instead, we keep the worst-case considerations on an abstract level and identify the worst case on the level of the low-level building blocks of a synchronization mechanism in a kernel using fine-grained locking. The analysis remains on a coarse functional level and does not go down to the level of basic blocks. The individual worst cases are often either constant or depend on a variable argument, such as the number of

[1]In this thesis, the term *monitor* refers to the monitor synchronization mechanism [BFC95], and not the monitor language construct first introduced in the Mesa programming language.

threads or the number of processors. We think this is the right level for a system designer to decide for or against a mechanism in general, see Section 2.1.6.

As further boundary, the thesis addresses multicore interference problems only at the software level and not at the hardware level, such as sharing of caches, interconnects and memory controllers. Section 2.1.9 lists techniques and related work to properly partition shared resources.

Parts of the content of this thesis have been presented in the following peer-reviewed workshop and conference papers:

- *Deterministic Fast User Space Synchronization*, in *OSPERT Workshop 2013* [Zue13].
- *Fast User Space Priority Switching*, in *OSPERT Workshop 2014* [ZBK14].
- *AUTOBEST: A United AUTOSAR-OS and ARINC 653 Kernel*, in *RTAS 2015* [ZBL15].
- *Deterministic Futexes: Addressing WCET and Bounded Interference Concerns*, in *RTAS 2019* [ZK19].
- *Turning Futexes Inside-Out: Efficient and Deterministic User Space Synchronization Primitives for Real-Time Systems with IPCP*, in *ECRTS 2020* [Zue20].

The author of this thesis is the main author of these papers.

The presented mechanisms have been evaluated and are going to be used in *ecHypervisor*, an automotive microkernel operating system developed in the *AUTOBEST* project and commercially distributed by easycore GmbH[2], in *Marron*, a research kernel developed in the *AQUAS* project[3], and in *PikeOS*, an avionics partitioning kernel/hypervisor by SYSGO GmbH[4]. The author of this thesis is the author of the first two systems and the author of the thread synchronization mechanisms in PikeOS.

1.4 Organization

The rest of this thesis is organized as follows:

Chapter 2 explains necessary concepts of synchronization mechanisms from both the *operating systems* and the *real-time systems* point of view and defines the terminology used in this thesis.

Chapter 3 presents related work, describes the system model, and defines metrics to evaluate efficiency and predictability. We then analyze higher-level

[2]`https://www.easycore.com/`.
[3]`https://aquas-project.eu/`.
[4]`https://www.pikeos.com/`.

synchronization mechanisms and their lower-level building blocks. Based on the analysis, we identify potential approaches in the design space of blocking synchronization mechanisms. Particularly we unroll the futex design in Linux as a state-of-the-art approach for efficient synchronization mechanisms with a special focus on predictability. We also analyze mechanisms to prevent lock holder preemption problems and other low-level wait and wake-up mechanisms.

Chapter 4 presents different approaches for both efficient and predictable synchronization mechanisms. The first three sections discuss different futex-based designs. Section 4.1 presents *deterministic futexes*, a kernels-level design for futexes which addresses the predictability issues found in the Linux kernel. Section 4.2 provides a futex design for statically configured systems named *static futexes*. Section 4.3 shows higher-level synchronization mechanisms in user space on top of the presented futex primitives in the operating system kernel. The next three sections present *light-weight spin-based monitors*. Its building blocks are an *efficient IPCP implementation* discussed in Section 4.4.1, related *wait and wake-up primitives* shown in Section 4.4.2, and higher-level synchronization mechanisms described in Section 4.5.

Chapter 5 evaluates these presented approaches by benchmarking the gains in average-case performance compared to a system-call-based baseline implementation on 32-bit ARM processors, and provides an analysis of the impact of fast paths on the worst-case timing in a model of worst-case memory accesses.

Chapter 6 discusses the results and compares the different approaches.

Chapter 7 concludes and summarizes the thesis.

The writing style in this thesis uses the first person plural form *"we"*, even if the author is a single person.

Listings and programming examples in this thesis use the C programming language.

Chapter 2

Basics

Firstly, we define the terminology and basic concepts used in the rest of this thesis in Section 2.1. Then, in Section 2.2, we present typical user-level synchronization mechanisms with their APIs, and take a brief look at synchronization mechanisms inside an operating system kernel.

2.1 Basic Concepts and Terminology

We first describe basic concepts of operating systems and resource management, real-time scheduling and locking protocols, architectural concepts and operating system standards, and the hardware architecture and its predictability issues. With it, we define the terminology further used in this thesis.

2.1.1 Operating System Concepts

A *process* is an instance of a computer program executing in an *address space*. Different processes have their own distinct address spaces, but processes can share parts of their address spaces via *shared memory segments*. A shared memory segment may be mapped at different virtual addresses in each address space.

A process comprises one or more *threads*, which are the executing parts of a program and active entities in a system. From the operating system's point of view, a thread consists of a *register context*, a *scheduling state*, and further attributes that define how and when threads are scheduled.

Next to threads, *interrupt service routines (ISRs)* are also active entities in the system. ISRs are activated by hardware events to perform service activities inside the operating system. Interrupt handling can have multiple steps, e.g. an ISR can notify a thread for further non-urgent follow-up activities. *Inter-processor interrupts* are a special class of interrupts that are activated internally by the operating system to enforce housekeeping on remote processors.

A *resource partition* consists of one or more processes and their threads as the next logical level of grouping above processes. It defines an upper bound of the physical resources that entities inside the partition may use. This concept of *spatial partitioning* provides stronger separation than the Unix process model. All communication channels between partitions must be explicitly configured, there are no implicit communication channels.

Temporal partitioning provides partitioning of time and confines any temporal misbehavior of threads. Here, multiple approaches are available: *Time partitioning* describes a two-level scheduling scheme mandated by the avionics standard ARINC 653. On the lower level, a *time partition* defines a set of independently schedulable threads. Threads inside a time partition compete against each other for CPU time, but not against threads in other time partitions. On the higher level, the different time partitions are scheduled using a repeating cyclic schedule called *major time frame*. This cyclic schedule is the *hyperperiod* of the periodic activation of all thread. On a multi-processor systems, partitions spanning more than one processor use *coscheduling* or *gang scheduling* and run simultaneously on all processors [Ous82]. Alternatives to time partitioning with its cyclic schedule are *server scheduling* approaches, e.g. *constant bandwidth server (CBS)*, which guarantee an amount of execution time for periodically executing groups of threads, or monitoring of execution time per thread in automotive systems. In any case, temporal partitioning defines an upper bound of available CPU time threads can consume and provides according mechanisms either to enforce these bounds or to detect if these bounds are exceeded.

The operating system's scheduler keeps threads eligible for scheduling on a *ready queue* or *run queue*. We call these threads *ready* or *runnable*. The currently executing thread on each processor is called *current* or *running* thread.

We say that the current thread is *preempted* when the thread is *involuntarily* descheduled in favor of another thread. And when the current thread *yields*, it *voluntarily* deschedules itself. Sometimes variants of a yield operation also allow to specify a target thread to hand the execution over to. In all cases, the former current thread remains in *ready* state.

In the context of a multi-processor system, threads can be scheduled on a limited set of the available processors only. This set is called *CPU affinity* of a thread and typically expressed by an *affinity mask*. Partitioning and process settings usually impose a *hard limit*, while each thread may further refine this set by a *soft limit*. A thread's *current* or *assigned processor* is the processor the thread is currently executing on or where the thread was already scheduled before. A thread is said to be *migrated* when it is moved to another processor. A current thread can also voluntarily change its assigned processor.

2.1.2 Shared Resources and Critical Sections

Resources describe both *physical entities* of a computer system, such as memory, I/O, and hardware devices, and *virtual entities* that require some kind of internal consistency, such as shared data structures. We denote threads as *in conflict* when two or more threads concurrently access a shared resource during an overlapping time period. We call the parts of a program that access shared resources *critical sections*. *Race conditions* happen when the accesses to a shared resource cause different results depending on the temporal order of these accesses. *Mutual exclusion* mechanisms ensure that only one thread can enter a critical section at a time. In this thesis, we use the term *critical sections* in a colloquial sense for all code sequences that need to be protected by a mutual exclusion mechanism (regardless of its type) for correctness and consistency.

An operating systems usually provides various types of *exclusive locks* to implement safe access to resources. A thread uses a *lock acquire* operation to get exclusive access to a shared resource, and a corresponding *lock release* operation marks the end of the exclusive access. We can further break down lock acquisition into the following steps: at first, the thread issues a *request* for the shared resource, and then the thread might have to *wait* until exclusive access to the resource is granted. Eventually, the thread becomes the *owner* the lock and with that *owns* the shared resource *exclusively*. Synonymously for the acquire and release operations, we *lock* and *unlock* a specific lock or a shared resource. We also say we *enter* and *leave* a critical section.

Next to exclusive locks, operating systems also often provide *reader-writer locks* or *shared locks*. Reader-writer locks allow more than one thread to enter a critical section if they access the resource in a non-modifying way as a reader. However, a lock would be unnecessary if the resource never changes, so these locks also provide exclusive access to a single writer. Reader-writer locks are beneficial for resources that are modified rarely. Note that when we discuss locks in this thesis, we refer to exclusive locks and to exclusive access to a resource.

Depending on its *scope*, a resource is *local* to a specific CPU if all accesses are guaranteed to happen only on that CPU. Otherwise, the resource is *global* and shared by all processors in a shared memory multi-processor system. In a single processor system or when protecting a local resource against concurrent access, it is already sufficient to temporarily inhibit scheduling, for example by masking interrupts during the execution of the critical section. When the resource is not immediately available, *blocking mutexes* are a viable solution. A *waiting* or *blocked* thread suspends execution until the thread is *woken up* or *unblocked* again. The waiting or blocking threads are often kept on a *wait queue* or *blocking queue*. We will use the verb *waiting* when a thread is waiting for a hardware resource or software signal, and *blocking* refers to any necessary waiting induced by other threads because the resource is not immediately available. For global resources, *spinlocks* in a shared memory segment serialize threads on *remote* processors of a tightly coupled multiprocessor system via *busy-waiting* on an atomic variable. We further refine blocking to *remote blocking* caused by threads on remote processors and *local blocking* caused by threads on the local processor.

2.1.3 The Lock Holder Preemption Problem

The *lock holder preemption* problem is specific to synchronization using busy-waiting, e.g. spinlocks. It describes the problem that on one CPU, a thread holding a lock is preempted, while on other CPUs, threads continue to spin to acquire the lock. A related problem is the *lock waiter preemption* problem when a *fair* spinlock is used. Here, an older waiting thread is preempted and hinders younger waiting threads to acquire the lock when the lock is passed over to the preempted thread [OL13]. In both cases, the spinning threads unnecessarily waste CPU time (and energy), as they can not acquire a lock.

Note that spinning or busy-waiting synchronization primitives can easily be implemented using atomic instructions in user space, but the problem with spinning synchronization is that the operating system scheduler is not aware whether a thread to preempt is currently inside a critical section or not. This is typically less a problem when using blocking synchronization, as the scheduler is notified implicitly.

Both the lock holder preemption and the lock waiter preemption problems are well studied problems in the context of operating system design, e.g. [MS98, ULSD04, OL13, KWS97]. Next to simply *ignoring* the problems of lock holder preemption and lock waiter preemption, proposed solutions to mitigate the problems fall into the following categories: (i) *preventing* or *avoiding* preemption while holding or waiting for a lock, (ii) *recovery* from preemption, and (iii) *mitigation of side effects* of preemption.

Preventing and avoiding preemption: Preventing preemption is the typical approach used inside an operating system kernel. The operating system kernel *disables preemption* on the local CPU before trying to lock a spinlock, and re-enables preemption after unlocking.

Similarly, for spinlocks in user space, Edler et al.'s approach in *Symunix II* temporarily disables preemption before trying to enter a critical section [ELS88]. For uncooperative threads not enabling preemption, the kernel sets up a short timeout to enforce preemption.

An alternative approach to avoid preemption is the *two-minute warning* mechanism proposed by Marsh et al. in *Psyche* [MSLM91]: the kernel indicates upcoming preemption (i.e. end of time slice) in a user readable flag, and a user space thread then avoids acquiring any spinlocks and rather yields.

Lastly, in the context of real-time systems, the use of real-time locking protocols such as the *stack resource policy (SRP)* [Bak91] or the *immediate priority ceiling protocol (IPCP)* [BW09] can also prevent lock holder preemption.

Preemption recovery: Ousterhout proposed a *two-phase synchronization scheme* where a thread first spins for a limited time, and then changes to blocking (first spin, then block). The time for spinning should be at least twice the time of a context switch [Ous82]. Karlin et al. [KLMO91] and Lim and Agarwal [LA93] later studied several strategies to adapt the spinning time. He et al. presented a technique for queue-based spinlocks using time stamps to indicate activity of the lock holder [HSS05].

Black presented the `thread_switch` primitive in Mach that provides a hint to the operating system which threads to schedule next. The mechanism was primarily designed to optimize message passing, but is also used by spinning threads to yield to a preempted lock holder and effectively *push the preempted thread out of its critical section* [Bla90]. Anderson et al. presented a similar mechanism with *scheduler activations* [ABLL92].

Takada and Sakamura proposed spinlocks with a helping scheme where a spinning processor completes the critical section for a preempted processor [TS97].

Kontothanassis et al. presented *scheduler-conscious synchronization* where a thread can determine and alter preemption indicators of other threads and then pass a lock to another thread and notify the scheduler to make it non-preemptible [KWS97].

Mitigating the side effects of lock holder preemption: Ousterhout's *coscheduling* puts threads that share locks into groups, and schedules all threads of a group at the same time [Ous82]. With this, preemption of a lock holder no longer matters to the other threads, as they are preempted at the same time.

For virtual machine scheduling, Uhlig et al. observed that a virtual machine is safe to be preempted w.r.t. internal locks of the virtualized operating system kernel when the virtual machine executes in user mode [ULSD04].

Interruptible critical sections for quantum scheduling were proposed by Johnson and Harathi [JH95]. Takada and Sakamura presented an extension to MCS locks to allow preemption to service interrupts [TS94]. Ouyang and Lange proposed preemptible ticket locks for virtual machines [OL13].

2.1.4 Real-Time Scheduling

The previous section described the entities of an operating system from a system's perspective, but different terminology is used in real-time scheduling theory. Real-time scheduling theory formalizes a system and its entities w.r.t. *timeliness*.

A real-time system comprises a set $\mathcal{T}$ of n_T tasks $T_1, ..., T_{n_T}$. Each *task* describes a repeatedly executed code sequence. When a task T_i is activated or invoked, e.g. at a specific time or by an internal or external event, it releases a *job*. A job T_i^j is the specific *instance* of the j's recurring execution of task T_i. The time the job is released is called *release time* r_i^j. After release, the job must complete its execution in a specified time interval, the so-called *relative deadline*

D_i, which is an attribute of the task. The resulting *absolute deadline* of the job is $d_i^j = r_i^j + D_i$. The maximum execution time budget a task needs to complete is called the *worst-case execution time (WCET)* e_i. However, a job can (and often will) finish earlier. We denote this time as *finish* or *completion time* f_i^j. The overall *response time* or *answer time* of the job is $a_i^j = f_i^j - r_i^j$. Also, the *maximum response time* of the task is $a_i = \max\limits_{\forall j}\{a_i^1, ..., a_i^j\}$.

To further model a system, we also need to know the time between two releases of a task. Liu and Layland's seminal work [LL73] defines the *periodic task model*, where the period p_i describes the time between two job releases of T_i. This model was later extended to the *sporadic task model* where p_i describes the *minimum time* between two jobs releases. In this model, the *utilization* $u_i = e_i/p_i$ of a task T_i describes the share of execution time the task needs, and the *total utilization* of the system is $U(\mathcal{T}) = \sum\limits_{T_i \in \mathcal{T}} u_i$. We then say that a task set is *schedulable* under a specific *scheduling algorithm* if all jobs finish before their deadlines expire, i.e. $\forall T_i \in \mathcal{T} : a_i \leq D_i$. The exact way *how* this *prioritization of tasks* is achieved depends on the actual scheduling algorithm.

Classic real-time scheduling algorithms are *rate-monotonic scheduling (RMS)* and *earliest deadline first (EDF)* [LL73]. RMS uses *fixed priority (FP)* scheduling and assigns a higher priority π_i the shorter a task's period is. In contrast, EDF schedules jobs of a task based on their current deadlines d_i^j. Compared to RMS, EDF uses *dynamic priorities*. The benefits of fixed-priority scheduling are a simpler implementation with $\mathcal{O}(1)$ timing characteristics, and predictable behavior on system overload[1]. In contrast, EDF provides a better utilization bound of 100% processor time. In both cases, a scheduler implementation in the operating system does not need to know or track execution times to operate correctly. The execution time is used as a parameter in a prior *offline analysis*. Note that we express higher priorities by higher integer numbers in this thesis.

Also, on a multi-processor system, we must consider the *scope* of the scheduler. *Partitioned scheduling* considers that each CPU is scheduled independently of each other CPU and does not migrate tasks between CPUs automatically. For example, on a dual processor system, the first CPU has two tasks with priorities 100 and 99 in *READY* state, and the second CPU has one task with priority 1, P-FP schedules the task with priority 100 on the first CPU, and the task with priority 1 on the second CPU, even if a higher priority task 99 exists. In contrast, *global scheduling* considers all CPUs at the same time and migrates tasks when necessary. For example, G-EDF on a four processor system would schedule the four tasks with the four earliest deadlines in the overall system in parallel. As a hybrid approach, *clustered scheduling* groups different processors into clusters. Tasks are then assigned statically to the specific clusters (partitioned), with global

[1]With FP, a misbehaving task that exceeds its execution time budget only harms tasks with a lower priority.

scheduling inside a cluster. Clustering for example helps in situations where processor cores of a multicore system share caches and the overhead of task migration is cheap [CAB07]. In the context of this thesis, we will further focus on P-FP only. For a recent overview of real-time scheduling theory, see the survey of Davis and Burns [DB11] and Brandenburg's dissertation [Bra11].

Finally, the task and job model relates to threads as follows: we can consider a thread to be an actual implementation of a task. In an endless loop, the thread then waits for its release, e.g. for the next periodic activation or for an external event, and when the thread is woken up, it executes the job in the specific iteration of the loop.

In the rest of this thesis, we will mostly use the term *thread* instead of task and job, as we focus only on *parts* of the overall execution of a job. Also, we will omit the job-specific index j for readability where possible.

2.1.5 Real-Time Locking Protocols

Sharing of multiple resources by different threads can lead to problems such as *deadlock* and *priority inversion*.

A deadlock occurs when multiple threads wait indefinitely for each other to release shared resources. When modeling the lock acquisition requests in a graph, deadlocks become visible as circular dependency. Deadlocks only can happen when lock attempts need to be *nested*, and a common technique to prevent deadlocks is to enforce a *linear order* on resource types that defines a strict sequential order how locks must be taken.

The priority inversion problem is at best explained by the following example. Assume a low priority thread T_L with priority π_L and a high priority thread T_H with priority π_H share a resource. Thread T_L locks the shared resource, but T_L is preempted inside its critical section by the high priority thread T_H. Now, T_H also wants to access the shared resource, but has to wait until T_L releases the resource. We call this *direct blocking*. Next, thread T_L executes again, but is subsequently preempted by a *third* thread T_M of medium priority π_M with $\pi_L < \pi_M < \pi_H$. In this scenario, the medium priority thread T_M now *indirectly* blocks the progress of the higher priority thread T_H and can cause a (potentially unbounded) priority inversion. Real-time scheduling theory allows to handle priority inversion problems.

Next to threads, a real-time system also includes a set of n_r shared resources $\ell_1, ..., \ell_{n_r}$. When a thread requires a resource ℓ_q with $q \in \{1, ..., n_r\}$, it first issues a request for l_q. The request is *satisfied* when the thread eventually holds l_q, and the request completes when the thread releases l_q. This allows to model the *blocking time*, i.e. the waiting time between issuing the request and holding the lock, and the time spent inside the critical section.

Scheduling theory now provides different *real-time locking protocols* that address priority inversion [Liu00]. We start with an overview of classic protocols for single processor systems, and later discuss their extensions for multiprocessor systems.

Locking Protocols for Single Processor Systems

The *non-preemptive critical section (NPCS) protocol* is the simplest protocol to avoid priority inversion. In NPCS, a thread performing any lock acquisition attempt is assigned the highest scheduling priority, so it cannot be preempted by other threads. This solves both the priority inversion and the deadlock problem, but now low priority threads can block unrelated high priority threads and affect their response time.

To address this problem, Sha et al. introduced the *priority inheritance protocol (PIP)* [SRL90]. In the basic (non-nested) form of PIP, a thread acquires a free lock immediately, but blocks when the lock is not free. In the latter case, the blocked thread T_H must have a higher priority π_H than the current lock holder T_L with priority π_L (otherwise T_H would not have preempted T_L in the first place). To solve the priority inversion, the blocked thread T_H hands down its current scheduling priority to the current lock holder T_L, until T_L finally releases the lock and T_H can in turn acquire the lock. When T_L releases the lock, the thread also reverts back to its original scheduling priority π_L.

Letting the lock holder T_L inherit the higher priority π_H of the blocked thread T_H effectively prevents preemption by all threads with lower priority. With this, the duration of the priority inversion for T_H is never longer than the time T_L spends in the critical section [Liu00]. A limitation of PIP is that *deadlocks* can occur when lock requests are nested. Also, PIP does not minimize the blocking time of higher priority threads as far as possible, as any locking request is immediately granted if the lock is free.

Both problems are addressed by the *priority ceiling protocol (PCP)* [SRL90]. PCP introduces the concept of *priority ceilings* and requires that resource requests of all threads are known in advance: The *resource priority ceiling* Π_{ℓ_q} is the highest priority of all threads that can acquire a specific resource ℓ_q, and the *current priority ceiling* $\hat{\Pi}(t)$ of the system is the highest priority ceiling of the resources that are currently in use (or 0 if all resources are free). Now, when a thread requests a resource, and the resource is not free, the thread blocks as usual. Otherwise, when the resource is free, the following rules apply:

1. If the current priority $\pi_i(t)$ of the thread T_i is higher than the current priority ceiling $\hat{\Pi}(t)$, the thread immediately acquires the resource.

2. If the priority of the thread is equal to the current priority ceiling $\hat{\Pi}(t)$ and the thread already holds another resource ℓ_r with same resource priority ceiling Π_{ℓ_r} as $\hat{\Pi}(t)$, the thread acquires the resource.

3. Otherwise, the thread blocks.

When a thread T_i blocks, it hands down its current priority $\pi_i(t)$ to the lock holder T_j, until the lock holder T_j releases all resources with a ceiling priority greater than or equal to $\pi_j(t)$ and reverts to its previous priority. Firstly, the priority ceilings of the resources define a strict order in which locks can be taken. This prevents deadlocks. Secondly, the protocol effectively *denies* locking attempts of sequences of nested locks by medium priority threads when one lock of the sequence is already in use by a lower priority thread. This effectively shortens the blocking time of higher priority threads.

Baker's *stack resource policy (SRP)* is a variant of PCP that allows non-blocking threads to share a single execution stack [Bak91]. Based on the same concepts of system ceiling and resource ceilings, the main difference of this protocol is to prevent any preemption in problematic locking scenarios instead of denying lock attempts. In case a priority inversion *might* happen, the protocol delays the release of any conflicting threads until the current thread completes. Also, requests for resources of the current thread are always immediately satisfied. Still, higher priority threads can preempt the current thread if they do not have any conflicting resource requests.

SRP requires that jobs of the threads never suspend their execution. This property allows the aforementioned stack sharing, as higher priority jobs simply allocate new stack frames upon release and clean these stack frame when they complete.

Another variant of PCP with similar analytical properties to SRP is the *immediate ceiling-priority protocol* or *immediate priority ceiling protocol (IPCP)* [BW09]. The key idea here is to raise the priority of a thread on each resource request to the ceiling priority of the resource and lower the priority back to the previous priority when releasing the resource. Now, while executing in a critical section, the thread can never be preempted by other threads competing for the same resource. And again, any locking attempts are immediately satisfied.

A related variant to IPCP for EDF scheduling is the *deadline floor inheritance protocol (DFP)* where instead of the thread's priority its deadline is temporarily set to the minimum (floor) of the relative deadlines of all threads that use a resource [Bur12].

Locking Protocols for Multi-Processor Systems with Busy-Waiting

On multi-processor systems, shared resources that are local to a specific CPU, i.e. when all requesting threads reside on the same CPU, can be effectively managed by the single processor protocols discussed before. But resource sharing between multiple processors (global resources) requires different protocols, and the set of available protocols also depends on the scheduling strategy, namely global scheduling or partitioned scheduling. We do not further consider clustered scheduling in this thesis.

A natural approach is to disable preemption locally and use *spinlocks with FIFO ordering* (in short: FIFO spinlocks) for global synchronization. For example, in the context of partitioned EDF scheduling, Gai et al. proposed MSRP, using SRP for local resource requests and non-preemptible FIFO spinlocks for global resources [GLN01]. With non-preemptive FIFO spinlocks on a system with m processors, a thread has to spin for at most $m - 1$ other processors to complete the same conflicting critical section. But *non-preemptive spinning* exposes the same general problems as NPCS, namely that non-preemptive blocking of low priority threads can severely affect the response time of unrelated high priority threads.

The *multiprocessor resource sharing protocol (MrsP)* by Burns and Wellings addresses this particular problem for P-FP scheduling and extends IPCP for multiple processors [BW13]. For both local and global resource requests, the requesting threads increase their scheduling priorities to the resources' ceiling priorities first. For shared resources, MrsP then additionally uses FIFO spinlocks. Now if the spinning threads detect that the current lock holder on a remote processor is not executing (because the lock holder is currently preempted), they try to *help out* the lock holder by *migrating* the lock holding thread to one of the spinning processors and let it finish its critical section at the spinning thread's priority ceiling. MrsP effectively prevents that lower priority threads can impact higher priority threads in any kind due to the priority ceiling, while the temporal impact of higher priority threads on lower priority threads is bounded.

Locking Protocols for Multi-Processor Systems with Blocking

A different approach for global synchronization is to use blocking synchronization. Rajkumar et al. proposed two extension of PCP, first the *distributed PCP* (DPCP) and later the *multi-processor PCP* (MPCP), for systems with partitioned scheduling [Raj91]. The key technique of both protocols is *priority boosting*. When a thread successfully acquires a global resource, its priority is temporarily raised above the priorities of threads not holding any resources, into a second boosted priority space mimicking the order of the normal priorities[2]. Similar to NPCS, this delays the release of new threads and prevents potential preemption by them. However, boosted threads can still be preempted by *other* boosted threads.

In DPCP[3], a protocol originally conceived for resource sharing in *distributed systems*, all threads and all resources are statically assigned and bound to specific host processors, and we can calculate the different ceiling priorities up-

[2]In textbooks, where higher priorities are expressed by decreasing integer values, the boosted priorities are often expressed by negative numbers. However, in an implementation using increasing priority values, we can simplify this and for example assume that priorities 1 to 100 relate to normal thread priorities and that priorities 101 to 200 relate to boosted priorities.

[3]Rajkumar et al. first introduced DPCP as *multiprocessor PCP* in [RSL88], but later Rajkumar renamed the protocol to DPCP in [Raj91].

front [RSL88,Raj91]. Threads can only directly access the resources that are local to their processors. To access global resources on remote processors, threads must temporarily migrate to the resource's remote host processor. Alternatively, we can model this by sending remote procedure calls (RPC) to the target processors and let remote threads handle the resource requests. While a thread is temporarily migrated to a remote processor (or waiting for an RPC reply), its original processor is free to schedule lower priority threads. Now, the rules of the PCP apply to each processor, but with all global resources (local or on remote processors) using their boosted priorities as ceiling[4]. Note that local resources are only accessed by local threads and they are not subject to priority boosting. In general, DPCP tries to minimize remote blocking, as migrated threads are never preempted by newly released local threads or local threads only holding local resources. On the other hand, migration or sending RPCs to a resource's target processor is a very costly operation.

In contrast, MPCP allows resource requests to happen from any processor, as resources are no longer bound to specific processors and migration or RPCs to remote processors are no longer needed [Raj90,Raj91]. The protocol follows the same rules as DPCP, where requests for local resources use PCP with non-boosted priorities, and requests for global resources use PCP with boosted priorities[5]. The priority ceiling for a global resource is based on the highest priority of any remote thread that shares the resource. However, low priority threads accessing a global resource enable priority boosting, which in turn causes local blocking to high priority threads. Because of this, MPCP requires a much more complex analysis to derive blocking times.

Another classic blocking protocol to extend for multiple processors is PIP. However, priority inheritance by itself does not help in P-FP scheduling scenarios, because thread priorities only have a meaning in relation to other threads on the *same* local processor, but not necessarily in global scope. To solve this problem, PIP can be combined with CPU migration of preempted lock holders to the CPU where threads observe blocking on a shared resource. Hohmuth and Peter proposed this approach as *local helping* in the context of the Fiasco microkernel [HP01]. As a corner case, local helping must use busy-waiting when the current lock holder is already running on another processor. To overcome this issue and use blocking rather than spinning, Brandenburg and Bastoni proposed *migratory priority inheritance* for locking in Linux [BB12]. Next to priorities, the lock holder also inherits the blocked threads' affinity masks and becomes eligible for scheduling on all processors in the superset of the affinity masks of all blocked threads. However, an implementation is not trivial.

[4]The protocol follows the spirit of the IPCP rather than PCP, as the priority boost is applied unconditionally each time.

[5]Again, MPCP follows the spirit of IPCP when applying priority boosting unconditionally.

Next to classic protocols, new approaches emerged in the recent decades due to advances in schedulability analysis. These newer protocols were designed to provide lower blocking bounds than the classic protocols by modeling suspensions differently. As example of the newer protocols, we describe the *FIFO Multiprocessor Locking Protocol (FMLP$^+$)* by Brandenburg [Bra11, Bra14]. The protocol is a refinement of Block et al.'s FMLP protocol [BLBA07] for partitioned scheduling. The protocol uses a FIFO-ordered blocking queue for each global resource and a modified priority boosting rule to ensure progress of lock holders. In FMLP$^+$, the lock holders' boosted priorities reflect the temporal order of their lock requests, thus ordering the execution of lock requests in FIFO order as well. With this, lock-holding threads are never preempted by local non-lock-holders and are never delayed by later arriving requests.

Locking Protocols in Practice

In practice, both PIP and IPCP are found as mechanisms to handle priority inversion in programming languages and operating system standards. IPCP is called *priority ceiling emulation* in Java, *ceiling-priority protocol* in Ada, and *priority protect protocol* in POSIX. PIP requires a sophisticated implementation to handle nesting and corner cases correctly. Zhang et al. note that textbooks like [Liu00] often get the corner cases of PIP wrong [ZUW20], and Moylan et al. observe that implementation often defer the restore to normal priority to the outermost nested critical section [MBM93]. SRP is used in environments where memory is scarce, and NPCS (combined with FIFO spinlocks) is used inside OS kernels and typically implemented by disabling interrupts or preemption. PCP is its original form is not used very often because of the complexity of its implementation compared to the other approaches. However, PCP, SRP, and IPCP requires upfront knowledge of all potential resources threads will use to

Table 2.1: Overview of locking protocols

Abbreviation	Protocol	Type	Comment
Uni-processor protocols			
NPCS	non-preemptive critical section	non-blocking	simple
PIP	priority inheritance protocol	blocking	nesting: risk of deadlock
PCP	priority ceiling protocol	blocking	complex
IPCP	immediate priority ceiling protocol	blocking	simple
SRP	stack resource policy (for EDF)	blocking	complex
DFP	deadline floor protocol (for EDF)	blocking	simple
Multi-processor protocols			
NPCS	non-preemptive critical section	busy-waiting	simple
MSRP	multi-processor stack resource policy (EDF)	busy-waiting	non-preemptive spinning
MrsP	multi-processor resource sharing protocol	busy-waiting	migration
DPCP	distributed priority ceiling protocol	blocking	priority boosting, migration
MPCP	multi-processor priority ceiling protocol	blocking	priority boosting
-	PIP + local helping	blocking	busy-waiting, migration
-	migratory priority inheritance	blocking	migration, complex
FMLP$^+$	FIFO Multiprocessor Locking Protocol	blocking	priority boosting

determine the correct resource priority ceilings. When this is not possible, PIP must be used. For multiprocessor systems with P-FP scheduling, PIP combined with CPU migration can be found. Newer protocols like FMLP$^+$ are currently not widely used. Table 2.1 summarizes the locking protocols Brandenburg's recent review of real-time locking protocols provides an in-depth overview of further real-time locking protocols [Bra19].

2.1.6 WCET Analysis Methods

For real-time scheduling, it is essential to know the upper bound of a thread's execution time, e.g. to determine the final system utilization and check utilization bounds. However, determining a thread's execution time is not an easy task, due to loops and branches in the program and speculative hardware components, such as caches, pipelines, and branch prediction. We denote the shortest execution time *best-case execution time (BCET)*, the average execution time *average-case execution time (ACET)*, and the longest execution time *worst-case execution time (WCET)*.

One common approach to determine the execution times is to use benchmarks and specific testcases in a *dynamic timing analysis*. The resulting *observed time* often depends on the overall system structure, such as the maximum number of involved threads. However, results obtained by benchmarking are just approximations of the exact execution times. For the BCET, the result is an overestimate, and for the WCET, the result is an underestimate [WEE$^+$08].

In contrast, techniques for *static timing analysis* use a computational approach to consider all possible execution times of a thread. The static timing analysis splits a program into smaller parts (down to sequential code sequences named *basic blocks*), determines the execution time for each part, and then infers a bound for the whole program with information on invariants such as loop bounds. Due to abstractions of the program execution and the hardware platforms, the computed WCET is usually an overestimate of the exact WCET. The presented bound can be considered a *worst-case guarantee*, however, analysis results are often very pessimistic [WEE$^+$08].

For this thesis, we do not need an exact WCET of either dynamic or static timing analysis. Rather, we focus on a high-level decomposition of the code, and determine *loop bounds*, *algorithmic code complexity*, and *locking architecture*, based on input parameters such as the number of threads and the number of involved processors. These results would be needed as first steps for both dynamic or static timing analysis to derive more realistic values of the execution time. The typical approach here is to decompose an application into separately measurable parts, determine the execution times of the parts in isolation, and combine the results into an overall execution time again.

The focus on these limited design aspects seems to be sufficient to detect major design problems w.r.t. WCET. For example, when comparing linked lists to balanced binary search trees (BST), linked lists require $\mathcal{O}(1)$ time for operations such as insertion at head or tail and removal, but $\mathcal{O}(n)$ time for sorted insertion. In contrast, balanced BSTs such as AVL or red-black trees provide operations like $find$, min/max, $insert$, and $remove$ in $\mathcal{O}(\log n)$ time. For a large number of queued objects n, when sorted insertion is needed, linked lists perform worse than BSTs with algorithmic complexity and show significant trends towards their worst case [ZK19].

2.1.7 Operating System Architectural Concepts

In this section, we provide a brief introduction to the common ideas and concepts of real-time operating systems (RTOS). We focus on the domain-specific RTOS standards for automotive, industrial, and avionics systems and related safety standards that represent the application-specific background to this thesis.

The key aspects for the following systems and standards are best explained by the two properties *predictability* and *isolation*. Predictability is a typical property of real-time systems. In the functional sense, it guarantees correctness, and in the temporal sense, it describes the precise behavior to derive precise *bounds* for the execution time in a WCET analysis. In contrast, the second property isolation does not necessarily improve timeliness, but it provides *failure propagation boundaries* and therefore increases *fault tolerance*. In particular, isolation concepts become necessary when different software components are put together on a single system (*integration*), and a failure in one part could impact the other.

Real-Time Operating Systems

The main idea of an RTOS is to ensure that the applications on top of the OS can react in a timely manner. In general, two different types of RTOS designs are considered, *event-triggered* and *time-triggered* systems. In an event-triggered system, an activity (a thread or an interrupt handler) is started by the occurrence of an external or internal event, while in a time-triggered system, the activities (only threads) are started periodically at given points in time [Kop91]. Both types result in an equal outcome when all deadlines can be met. However, time-triggered systems are less flexible towards non-real-time applications but resilient against event storms. Time-triggered systems can be considered a subset due to the periodic nature of the more general sporadic model of event-triggered systems.

The RTOS helps the developer to implement a system based on the foundation of real-time scheduling theory, and the RTOS provides a framework that helps to ensure that timing constraints of the application are met, or, if something goes wrong, the timing violation is at least detected. With this, the design philosophy of an RTOS is to aim for a high level of determinism, and therefore the RTOS can ensure timeliness. In contrast, general purpose operating systems like Linux or Windows often optimize for throughput.

The basic concepts of real-time operating systems follow the naming convention of real-time theory and use the term task for a thread. Inside threads, jobs are often modeled as repeated execution of waiting for an event and then reacting on it.

Listing 2.1: Modeling a job as repeated execution of waiting in a loop

```
 1 void task_5ms(void)
 2 {
 3     time_t next_expiry;
 4     err_t err;
 5
 6     next_expiry = current_time();
 7     while (1) {
 8         // activate thread every 5 ms
 9         next_expiry += TIME_MS(5);
10         err = sleep_abs(next_expiry);
11         if (err == ERROR_TIME_IN_THE_PAST) {
12             // implicit deadline missed
13             ...
14         }
15
16         // run periodic jobs A and B
17         job_A();
18         job_B();
19     }
20 }
```

As Listing 2.1 shows, the thread waits with a period of 5 ms (lines 9 and 10) before running two non-blocking jobs A and B (lines 17 and 18). Running two jobs in a single thread is not unusual if we consider that the tasks of both jobs have the same period and therefore must also have the same priority under rate-monotonic scheduling. However, the example above is not specific to any RTOS and only shows one possible way to implement periodic waiting. For example, the RTOS could also handle time keeping internally and provide a `sleep_until_next_period()` function instead, or the RTOS could even hide the full periodic task activation and simply call the job functions.

OSEK OS and AUTOSAR OS

In mid 1990s, German and French automotive manufacturing companies standardized an RTOS API called *OSEK OS* (*Open Systems and Interfaces for Electronics in Motor Vehicles*) for event-driven control systems on automotive electronic control units (ECUs) [OSE05]. ECUs are small, resource-constrained microcontrollers, thus the focus of OSEK OS is to provide a light-weight and tailorable RTOS to exploit existing hardware as much as possible. To achieve this, OSEK OS uses a static system design that is fully known at compile time, so unused features can be omitted.

OSEK OS uses a task model based on preemptive P-FP scheduling and proposes IPCP for synchronization. Threads come in two flavors: *basic tasks* (threads) cannot enter a waiting state, while *extended tasks* (threads) can wait for *OSEK events*. OSEK events are the only notification mechanism to wake up blocked threads. An event relates to a bit in a per-thread bitmask a thread can wait for. Basic tasks support a concept named *multiple activation* where pending activations are queued up to a configured upper limit to immediately restart the thread upon completion. Combined with non-preemptive scheduling, this allows stack sharing between basic tasks on systems where memory resources are tight. Different OSEK conformance classes further allow implementations to remove support for extended tasks or to simplify ready queue management operations to further save memory for small-scale automotive systems. In OSEK, ISRs can be considered a special form of basic tasks that are activated by hardware interrupts and execute above the priority range of normal threads.

Since 2003, the *AUTOSAR (AUTomotive Open System ARchitecture)* consortium has taken over the standardization of automotive software[6], and OSEK OS became AUTOSAR OS. AUTOSAR defines a three layer architecture, comprising standardized software modules and common interfaces at the base, a middleware that abstracts communication and data-flow between and inside ECUs, and an application layer on top.

Today, AUTOSAR defines a *classic platform* based on OSEK OS as AUTOSAR OS and targeting ECUs, and an *adaptive platform* based on POSIX PSE51 for novel driver assistant systems like autonomous driving. The computational demand for the latter exceeds the possibilities of microcontrollers, therefore the AUTOSAR Adaptive Platform lifts the strict resource constraints and targets high performing CPUs. As the AUTOSAR adaptive platform is compliant to POSIX PSE51 (see below), further use of the term *AUTOSAR* in this thesis exclusively relates to the AUTOSAR classic platform and AUTOSAR OS or OSEK OS.

POSIX PSE51

In the mid 1980s, diverging Unix implementations complicated the development of portable software. To ensure compatibility between different Unix implementations, a standardization process for a *Portable Operating System Interface (POSIX)* was started, resulting in multiple *IEEE 1003* and *ISO/IEC 9945* standards over time [IEE17]. The standards cover core services (POSIX.1), real-time (POSIX.1b) and threads (POSIX.1c), but also shells and utilities (POSIX.2). The POSIX standards were quite inclusive to existing implementations, and the standards allow great flexibility in term of features.

[6]AUTOSAR standards are publicly available on `https://www.autosar.org/`.

Since 2003, the POSIX standard defines four profiles that allow to use the POSIX API in smaller embedded system: PSE51 defines a minimal profile for single process multi-threading applications; PSE52 adds simplified file system capabilities; PSE53 adds support for multiple processes and networking; and PSE54 defines the full POSIX system with multi-user support and the full file system. Note that even though the PSE53 and PSE54 profiles define an isolation concept based on Unix processes, this isolation concept is too weak for today's safety-critical systems. Today's Unix systems provide stricter mechanisms to isolate groups of processes, like *containers* on Linux, *jails* on BSD, or *zones* on Solaris. Therefore, many RTOS vendors only comply to the PSE51 profile for real-time multi-threading and add selected parts of the higher profiles, while at the same time using completely different process models or isolation concepts than Unix [Jos04].

PSE51 defines preemptive P-FP or G-FP scheduling with a configurable processor affinity and round-robin scheduling for non-real-time threads. As locking protocols, the standard defines both PIP and IPCP. A wealth of POSIX synchronization and communication mechanisms is available. We will further discuss these in detail in Section 2.2. Lastly non-Unix PSE51 RTOSs often support threaded interrupt handling, allowing ISRs to be scheduled like normal threads.

ARINC 653

Aeronautical Radio, Inc. (ARINC) is a standard body of aviation industry members. First published in January 1997, the *ARINC 653* specification defines the API between the applications and the OS running on an avionics computer platform [AEE15]. *ARINC 653* currently comprises multiple sub-standards (*parts*). Part 1 *"required services"* defines a process model based on statically configured spatial partitions, time partitioning, multi-threading and thread synchronization, health monitoring, and services for communication between partitions. Part 2 *"extended services"* describes a POSIX-like file system interface, logbooks and networking extensions. Part 4 *"subset services"* describes a very minimal system comprising just two threads for highly safety-critical systems.

With this, ARINC defines concepts for both predictability and isolation. The authors of the standard put a special focus on the latter: the standard defines a strict partitioning model that exceeds the rather weak process model of Unix by far. Like Unix processes, ARINC partitions operate independently, have their own address space, and their own access mechanisms to resources, but ARINC lacks a concept of parent-child-relations between partitions and related mechanisms to manipulate other processes, as partitions are statically configured in the systems. Lastly, ARINC partition communication mechanisms are limited to pure data exchange and allow a robust decoupling of partitions.

ARINC 653 uses preemptive P-FP or G-FP scheduling with a configurable processor affinity. The standard defines NPCS and IPCP as locking protocols.

ARINC 653 also supports time-triggered thread activation and includes additional thread attributes for period, deadline, and execution time budgets. We will further discuss ARINC synchronization mechanisms in Section 2.2. Lastly, typical avionic systems use cyclic polling instead of interrupt-driven workloads, therefore ARINC 653 do not define an API for interrupt handling.

Commonalities and Differences of RTOS Standards

The three presented RTOS standards share many concepts, such as fixed-priority scheduling and a common task model comprising running, ready and blocked/waiting states. The author of this thesis provided an analysis for AUTOSAR and ARINC 653 in prior work [ZBL15].

Comparing the three standards, the OS part of AUTOSAR has the narrowest scope and provides a very minimal set of mechanisms to construct simple but robust real-time systems. The original focus of OSEK OS and AUTOSAR OS was on predictability; isolation was added later.

In contrast, ARINC 653 defines already a much wider scope with a support for time-triggered thread activation and a well-defined set of synchronization mechanisms. Both predictability and isolation were included into the system design from the beginning.

POSIX extends the possibilities even further, but also shows that it was not designed as a real-time system in the first place, as unspecified behavior or even conflicting mechanisms can be found, e.g. multi-threading and process creation do not play well together [BAKR19]. Here, the original design focused on isolation, and predictability is a retrofit.

Safety Standards

Safety standards define legal binding procedures and processes to ensure the general safety of products. Safety standards are often driven by customer protection rights, product liability laws, or environmental regulations, and usually define *best practices* for the whole life cycle of a product, from design via production and maintenance through to disposal. A producer of a product shows compliance to relevant safety standards to ensure that the product is safe to use and to protect the producer to a certain degree against recovery of damages in case of failures. Independent third-party organizations supervise and certify compliance.

Depending on the technical domain, different relevant standards apply. The standard *IEC 61508* for *functional safety of electrical/electronic/programmable electronic (E/E/PE) systems* is a general norm covering industrial automation use cases [IEC98]. *IEC 61508* is a general norm for a wide range of domains and applicable if no specialized standard applies. In the automotive field, the norm *ISO 26262* for *functional safety in road vehicles* supersedes and refines *IEC 61508* [ISO11a]. For avionics software, the standard *DO 178C* for *soft-*

ware considerations in airborne systems and equipment certification defines the requirements of civil aviation agencies for certification [RTC11].

IEC 61508 uses a risk analysis based on both *likelihood of occurrence* and *consequence of failures*, such as injuries or loss of life, of a system, and requires that a certain *safety integrity level (SIL)* must be achieved. The standard distinguishes between software without a functional safety impact (*quality management, QM*), and four levels SIL 1 to SIL 4, with SIL 4 being the strictest. For high SILs, the standard recommends rigorous testing and analysis efforts. Applied to software development, this means the higher the level is, the stricter the design process needs to be, and the higher the resulting development costs will be. ISO 26262 refines this concept to *automotive safety integrity levels (ASILs)*, where ASIL A relates to SIL 1, ASIL-B/C to SIL 2, and ASIL-D (strictest) to SIL 3. SIL 4 is (at least today) not relevant in the automotive context. Similarly, DO 178C defines a concept of a *design assurance level (DAL)*, where DAL-A (strictest) roughly relates to SIL-4 and DAL-E to QM.

Mixed-criticality systems

The advances in computing power in the last decades allow to put functionality previously provided by separate computing systems into a single system, and the additional computing power allows to implement *more* functionality than before. These concepts of *integration* and *consolidation* allows to reduce *size, weight and power* (SWaP) concerns, which is an important topic in many industries, either to save costs, or to achieve secondary goals such as environmental regulations. However, tighter integration and consolidation brings the risk that an error in one component can now impact other components, as the propagation of errors was limited on the former physically separated computer systems.

For systems with multiple functions, IEC 61508 requires:

> An E/E/PE safety-related system will usually implement more than one safety function. If the safety integrity requirements for these safety functions differ, unless there is sufficient independence of implementation between them, the requirements applicable to the highest relevant safety integrity level shall apply to the entire E/E/PE safety-related system.

This means that an operating system must comply to the SIL of the highest critical application, and, if the operating system does not guarantee the required independence between different applications, to all other software as well. For automotive systems, ISO 26262 requires "freedom of interference" between software components of different safety levels [ISO11a]. Similar requirements are defined in *DO 178C* for avionics software. This drives the need for strong isolation concepts in operating systems to keep development efforts down when combining software of different criticality levels into a single *mixed-critical system.*

Technical challenges in the design of mixed-critical systems comprise the contrast of guaranteeing strict temporal decoupling of applications while at the same time allowing timely reactions to events, efficient system utilization versus overprovisioning, or mitigating the impact of resource sharing, especially on multicore systems [BBB+09].

In the context of real-time scheduling, Vestal proposed the idea to assess the WCET of a high critical application differently when running together with a low critical application [Ves07]. Based on the observation that different techniques to determine the WCET result in different bounds, e.g. a very pessimistic WCET bound e_{HI} obtained by a static code analysis tool and a less pessimistic and more practical WCET bound e_{LO} obtained by measurements, a valid schedule must allow the critical application to always meet its deadlines by using e_{HI} in isolation, but can use e_{LO} when running together with the low critical application and sacrifice the execution of the low critical application in the worst case. Interpreting the high critical application's safety margins differently allows to utilize the time reserved for overprovisioning. Later research applied and extended Vestal's approach to other fields of real-time research.

For an overview on mixed-criticality systems, see the review of Burns and Davis [BD19].

2.1.8 Processor Architecture

For the processor architecture, we assume a contemporary 32-bit or 64-bit CPU, like x86, ARM, PowerPC, or RISC-V. These processor architectures support at least two processor modes: application code executes in *user mode*, while the operating system kernel executes in *supervisor mode* or *privileged mode*. Further levels, such as a *hypervisor mode*, can be supported, but are not necessary.

The CPU provides at least some kind of *memory protection mechanism*, so that kernel and user applications reside in distinct address spaces. The kernel can access data in user space, but code in user space cannot access any data of the kernel. Also, user applications cannot access the data of each other user applications unless this is explicitly configured. A *memory management unit (MMU)* supporting *virtual memory* is not mandatory, but if used, the MMU provides a small cache of virtual to physical memory translations, the *translation look-aside buffer (TLB)*. Memory protection without virtual addressing can be provided by a *memory protection unit (MPU)*.

To call into the operating system and perform a so-called *system call*, the CPU provides a hardware trap mechanism for execution in user space. When taking the trap exception, the CPU then continues execution at a function that is under control of the operating system. This function will further dispatch the system call. In any case, either the trap mechanism in hardware or the code of the operating system in software switches from any previous *user stack* (stack in user space) to a *kernel stack*. When the system call operation returns, the previous context in user space is resumed.

Similar transitions into the operating system must also happen on any further *synchronous exceptions*, like arithmetic errors or memory access violations, and on *asynchronous interrupts*, like the expiry of a timer or a hardware interrupt of an I/O device.

Today's systems have multiple hardware execution units for threads. These execution units can be dedicated processor cores on a multicore chip, or dedicated processor modules accessing memory via a shared bus, a mesh, or a point-to-point link, or any combination of the above. Typically, the first level of caches is not shared with other processors, but other levels of caches might be shared among processors, and the system memory definitively will be shared. Sharing causes *interference* when a shared entity cannot provide the same level of service when more than one processor uses it as compared to when only one processor uses the entity in isolation.

As a low-level mechanism for atomic modification of memory, the processor architecture provides either *compare-and-swap (CAS)* or *load-linked/store-conditional (LL/SC)* instructions. Both mechanisms allow to perform atomic *read-modify-write* operations on values in memory and can be converted into each other [AM95]. Both *read and CAS* and LL/SC sequences might observe that the value in memory has changed between reading and writing. In this case, the update of the value in memory fails, and an algorithm using atomic operations might retry the sequence. This can lead to *starvation problems*. Additionally, CAS does not detect if the value in memory has changed back and forth between the initial read and the CAS instruction. This is known as *ABA problem* [Mic04]. LL/SC prevents the ABA problem and always detects transient changes, but is prone to *spurious failures on SC* due to other memory accesses near the addressed memory, e.g. data in the same cache line. In this thesis, we will further use CAS operating on 32-bit variables and on pointer-sized memory objects.

Lastly, modern processors execute *out-of-order* and may speculatively fetch data earlier than needed, or re-order store operations. When the exact ordering of memory accesses is necessary, the programmer must place *memory barriers* into the instruction stream. These memory barriers often accept further parameters to specify the exact type of ordering, i.e. *loads with loads*, *loads with stores*, *stores with stores*, or *stores with loads*, or define *load-acquire* and *store-release* semantics. Note that memory barriers typically only ensure that other cores *observe* the desired ordering, but the memory barriers do not guarantee that memory accesses have *completed*. This *memory model* depends on the actual processor architecture. Recent revisions of the C and C++ language standards provide an abstract memory model that supports current processor architectures [ISO11b].

2.1.9 Predictability Issues on Multicore Processors

The old adage attributed to Henry Ford

> "If I had asked people what they wanted, they would have said faster horses,"

explains the relationship of real-time programming to multicore chips. Even if the increased computing power of multicore processors allow consolidation of multiple applications on a single platform, programming for multiple cores is much more difficult than programming for a single core, real-time theory does not provide optimal solutions, and multicore processors introduce a high level of non-determinism. Multicore processors do no longer allow WCET analysis of programs in isolation, as applications on additional processors cause interference due to sharing of caches, interconnects and memory controllers. For example, Nowotsch and Paulitsch show that the WCET of an application running on a multicore system can be multiple times slower than the same application running on a single core without other cores running interfering applications [NP12]. Because of this, certification authorities in avionics demand mitigation of interference on multicore systems in the CAST-32 and CAST-32A position papers, but leave specific proposals open [FAA16].

In turn, different techniques were developed to mitigate these effects. The strategies range from preventing side effects, e.g. by guaranteeing certain memory bandwidth in a network-on-chip design, to detection of overuse and enforcement of quotas [PMN+09]. Particular techniques such as *MemGuard* involve using the CPU's performance counters to monitor the bus bandwidth of shared caches and react if the bandwidth is exceeded [YYP+13]. Under the term *Single Core Equivalence* a whole set of related techniques for the whole memory stack were developed [SCM+14]. Similarly, recent work on the *"one-out-of-m" problem* target these hardware interference problems by several hardware and software partitioning techniques, such as cache and memory partitioning [KWC+16].

With SPeCK [WRSP15], Wang et al. present a kernel that provides *scalable predictability*, where predictability bounds observed on a single core, remain constant with an increase in cores.

For low-level synchronization mechanisms, David et al. [DGT13] and Guiroux et al. [GLQ16] provide recent analyses on performance, scalability, and interference of low-level synchronization mechanisms.

The review of Burns and Davis on mixed-criticality systems provides further references to recent work on multicore interference [BD19].

2.2 User-Level Synchronization Mechanisms

We will now briefly discuss user-level synchronization mechanisms following the conventions of POSIX PSE51, ARINC 653 part 1, and AUTOSAR OS, our systems of interest. This covers the full set of synchronization mechanisms between threads in systems using these operating system standards. Note that POSIX defines even more synchronization mechanisms, e.g. pipes, but these mechanisms are not part of the PSE51 subset. We will furthermore explain the *futex* mechanism in Linux [FRK02]. Futexes provide an efficient way to implement most synchronization mechanisms in user space without system calls. For each presented mechanism, we describe use cases and the overall semantics and give a brief description of each operation.

We provide typical API names in a consistent and unique way in *object-verb* order. For brevity, the following description omits the `pthread_` prefix for POSIX functions. ARINC 653 functions are shortened likewise, e.g. `SEND_QUEUING_MESSAGE` becomes `qport_send`[7]. Similarly, the AUTOSAR event mechanism `WaitEvent` becomes `event_wait`. We also omit any necessary functions to destroy the synchronization objects when a description of these functions is not further relevant.

We will start with a brief overview and categorization of synchronization mechanisms. Table 2.2 shows an overview of the synchronization mechanisms.

Synchronization mechanisms can be categorized by their *type*: A mechanism might provide *mutual exclusion* (spinlock, mutex, reader-writer lock) or a *notification* mechanism (condition variable, event), or even both on a low level (semaphore) or a high level (monitor[8]). A notification mechanism might wake up *one* (event, semaphore, monitor) or *many* waiting threads (event, monitor, reader-writer lock). Also, the synchronization mechanism might define different *queuing disciplines* for threads waiting on the mutual exclusion part or the notification part. And for notifications without waiters, the notification might get recorded (semaphore, event) or is lost (condition variable, monitor).

[7]In ARINC 653, endpoints for queuing messages are denoted *queuing ports*.

[8]We consider monitor synchronization constructs with Mesa-style condition variables. Signaling a condition variable never blocks the caller.

Table 2.2: Overview of synchronization mechanisms

Mechanism	**Type**	**Category**	**Wake-up**	**POSIX**	**ARINC**	**AUTOSAR**
spinlock	mutual excl.	busy-waiting	notify next	✓		✓
mutex	mutual excl.	blocking	wake up one	✓	✓	
condition variable	notification	blocking	requeue one/all	✓		
reader-writer lock	mutual excl.	blocking	wake up one/many	✓		
semaphore	mut./notif.	blocking	wake up one	✓	✓	
barrier	barrier	blocking	wake up all	✓		
one-time initializer	barrier	blocking	wake up all	✓		
queuing port	queue	blocking	wake-up one		✓	
buffer	queue	blocking	wake-up one		✓	
sampling port	-	non-blocking	-		✓	
blackboard	barrier	blocking	wake-up all		✓	
eventmask	notification	blocking	wake-up one			✓
event	notification	blocking	wake-up all		✓	
futex	complex	blocking	wake up/req. many	(✓)		

Another category to classify synchronization mechanisms is whether they use *busy-waiting / spinning* or *blocking* to implement mutual exclusion. Higher-level synchronization mechanisms might use a gradual transition between both types, e.g. first use spinning for a limited time, then use blocking as fallback [Ous82].

Synchronization mechanisms can further be classified as *blocking synchronization* when threads may need to block in the scheduler, or as *non-blocking* or *lock-free* if not. Lock-free algorithms often use the atomic read-modify-write primitives provided by the hardware, e.g. CAS and LL/SC, but bear the risk of starvation due to retries of the atomic operations. A stronger variant of lock-free algorithms are *wait-free* algorithms that prevent starvation. Internally, higher-level synchronization mechanisms often use both mechanisms, lock-free mechanisms as fast path and blocking mechanisms as fallback.

2.2.1 Spinlocks

Spinlocks provide *busy-waiting* synchronization and are most often implemented without further involvement of the operating system kernel. Spinlocks operate on variables in user space using atomic operations. Spinlocks are available in both POSIX and AUTOSAR in user space, and are also used inside an OS kernel.

The operations on spinlocks are:

- `spin_init` to initialize a spinlock to unlocked state,
- `spin_lock` to acquire a spinlock with busy-waiting,
- `spin_trylock` to try to acquire a free spinlock without busy-waiting, and
- `spin_unlock` to release a spinlock.

Over time, different types of spinlocks were invented:

Simple spinlocks can be implemented in a single bit encoding the spinlock state (free/locked) using a *test-and-set (TAS)* instruction. However, simple spinlocks do not provide fairness and have scalability issues due to high bus contention [And90, MS91]. Approaches such as to use *exponential back-off* during busy-waiting have been proposed to improve scalability [MS91].

Ticket locks address these issues by using two *counters* [RK79, MS91]. Lock acquisition happens in two steps: first drawing a ticket by atomically incrementing the first counter, then waiting until the ticket becomes *serving* as indicated in the second counter. Still, ticket locks are criticized for poor performance on many-core systems with high lock contention [BWKMZ12].

Queue locks address this scalability issue and distribute the spinning to acquire the lock to different cache lines, e.g. to a state variable of the previous lock holder. Typical queue locks are *CLH locks*, which were independently described by Craig [Cra93a] and Landin and Hagersten [MLH94], and *MCS locks*, named after their inventors Mellor-Crummey and Scott [MS91] (not related to mixed-criticality systems). In practice, MCS locks are used, as they need less space. *Array-based queue locks* exists as well [Cra93b, Rhe96, And90, GT90], but are used less often in practice, as they need an array to hold all possible threads trying to acquire a lock.

The actual implementation choice for spinlocks depends on the operating system or on the application. Operating system kernels usually use ticket locks or MCS locks [Cor14]. User space applications often use simple locks and adapt the lock to a queue-based approach on contention [LA94].

As low-level synchronization primitives, the `lock` and `unlock` operations also contain acquire and release barriers.

2.2.2 Mutexes

Mutexes (short for *mutual exclusion*) are a *blocking* synchronization primitive and require support of the operating system kernel.

Mutexes are available in both POSIX and ARINC 653.

POSIX standardized the mutex API with a focus on performance, ease of use, and real-time requirements. The latter are well defined, because the threading workgroup within the standardization committee was also part of the POSIX real-time extension specification.

POSIX defines the following behavior of mutexes:

Protocol: The mutex protocol defines the behavior of a mutex w.r.t. real-time scheduling and to prevent *priority inversion problems*. POSIX defines three basic protocols: (i) no protocol (default), (ii) a *priority inheritance protocol (PIP)* (dynamic), and (iii) a *priority ceiling protocol (PCP)* (static). Both priority protocols change the scheduling priority of a thread holding a mutex lock to prevent priority inversion problems and follow the ideas discussed in Section 2.1.5.

Type: The *type* of a mutex defines the behavior regarding deadlock detection. POSIX defines four types: (i) without *deadlock detection*, (ii) with deadlock detection, (iii) *recursive mutexes*, and (iv) an implementation-specific default type. Deadlock detection includes extra sanity checks. Recursive mutexes allow a lock holder to acquire the same mutex multiple times (and requires the same amount of unlock operations). This can be realized by an internal counter.

Sharing: Mutexes have a `pshared` attribute that defines if the mutex is *private* to the creating process or can be *shared* with other processes.

Robustness: The optional robustness attribute was recently added to the POSIX specification. When a lock holding thread or its process dies while having a mutex locked, the critical section is usually in an invalid state. In this case, the next locking attempt of another thread will return an error, and the new lock owner can inspect the critical section and decide if the state is consistent.

Queuing Order / Fairness: POSIX does not explicitly define the queuing order of threads being blocked on a mutex without a priority inheritance protocol.

The standard only specifies: "[...] the scheduling policy shall determine which thread shall acquire the mutex" [IEE17], but implementations typically order blocked threads by their priority, and use FIFO ordering for threads of the same priority.

The API comprises the following functions:

- `mutex_init` to initialize a mutex to unlocked state,
- `mutex_lock` to acquire a mutex with blocking,
- `mutex_trylock` to try to acquire a free mutex without blocking,
- `mutex_timedlock` to try to acquire a mutex for a given time,
- `mutex_unlock` to release a mutex, and
- `mutex_consistent` to mark a mutex as consistent again after its owner died.

Mutexes in ARINC 653 follow the concepts of POSIX. The queuing discipline is selectable and is either FIFO- or priority-ordered. ARINC 653 does not allow nesting of mutexes, but always enables a priority ceiling protocol.

2.2.3 Condition Variables

Condition variables are a primitive for *waiting* and *notification* inside a critical section. Simply using busy-waiting is not sufficient to wait until some condition becomes true, as other threads cannot enter the critical section in the mean time. Instead, a thread wanting to wait for a condition first releases a critical section and then waits *outside* the critical section. After the condition has been asserted, the thread can enter the critical section again. Condition variables are therefore used to wait in mutex-based critical sections and require support of the operating system kernel.

POSIX defines the usual *sharing* attribute and a *clock* attribute to select different *clock sources* of the operating system for waiting with a timeout.

Similar to mutexes, in a POSIX system, the scheduling priority of the blocked threads defines who is woken up first when signaling a condition variable. An implementation can therefore sort the waiting threads in priority order in general and in FIFO order on a priority tie.

The following functions are available:

- `cond_init` to initialize a condition variable,
- `cond_wait` to release the associated *support mutex*, to wait for the condition variable to be signaled, and to acquire the associated mutex again after waiting,

- `cond_timedwait` additionally allows to specify a timeout while waiting,
- `cond_signal` to wake up only *one* waiting thread, and
- `cond_broadcast` to wake up *all* waiting threads.

POSIX specifies that condition variables can experience *spurious wake-ups* and *stolen wake-ups*. Depending on the implementation, a single `cond_signal` operation can wake up two or more threads due to a race condition when multiple threads start to wait on a condition variable at the same time. Therefore, callers of `cond_wait` should loop until the waiting condition becomes true. However, in a real-time system, condition variables should neither show spurious wake-ups nor stolen wake-ups.

While POSIX allows so-called *naked notifies* [LR80], i.e. a notification without having the support mutex locked, the standard recommends that a wake-up operation on a condition variable should only happen from within the related critical section to achieve "predictable scheduling behavior" [IEE17]. This is also recommended for real-time applications, as the test whether to perform a wake-up and the actual wake-up are not atomic without a critical section.

The last side effect of condition variables is known as *thundering herd effects* and can happen when after a `cond_broadcast` all woken-up threads compete to acquire the mutex again [HG09]. An implementation should avoid this.

The described concept of condition variables were first implemented in *monitors* in the *Mesa* programming language and are known as *Mesa-style condition variables* [LR80]. See also Buhr et al. for further classification of monitor synchronization constructs [BFC95]. Condition variables are not available in ARINC 653 and AUTOSAR.

2.2.4 Reader-Writer Locks

Reader-writer locks are an extension to mutexes. A reader-writer lock either grants multiple *reader* threads *shared access* the same time, or grants a single *writer* thread *exclusive access* to the data. Multiple interpretations of reader-writer locks using busy-waiting or blocking are possible, however, POSIX expects reader-writer locks to eventually block. Unlike for mutexes, POSIX does not specify a concept like condition variables for reader-writer locks.

Due to the differentiation into readers and writers, different types of blocking may occur. Reader-writer locks can either prefer readers over writers or vice versa. Consider a reader-writer lock that is currently locked by a reader. Now two additional threads, one writer and one reader, try to gain access to the lock. The implementation could either give precedence to the reader thread, admit an additional reader, and let the writer thread wait (probably unbounded if more readers appear), or give precedence to the writer and let the reader wait. POSIX does not define the preference type of reader-writer locks, but the Linux implementation does and also supports *reader preferring* or *writer preferring* locks, while starving potential threads on the other side. For the ordering of blocked threads, POSIX further defines that after wake-up of real-time threads, these acquire the lock in priority order, and write locks take precedence over read locks on priority tie.

The following operations on reader-writer locks are available:

- `rwlock_init` to initialize a lock to unlocked state,
- `rwlock_{rdlock|tryrdlock|timedrdlock}` for shared access to a lock,
- `rwlock_{wrlock|trywrlock|timedwrlock}` for exclusive access to a lock, and
- `rwlock_rdunlock` and `rwlock_wrunlock` to unlock a lock in each mode. The POSIX API only defines a single unified `rwlock_unlock` function, so an implementation must additionally track whether the current caller was a reader or a writer.

Note that a simple version of reader-writer locks can be implemented using normal mutexes and condition variables. An implementation of fair reader-writer locks is shown in the textbook by Herlihy and Shavit [HS08].

2.2.5 Counting Semaphores

Dijkstra introduced *semaphores* as one of the first mechanisms for process synchronization. A semaphore defines an implicit *resource counter* that expresses the number of *available* resources of a given type. The classic operations on the counter are *P* or *down* to decrement this counter to acquire a resource. If the counter is zero, the caller has to wait until resources become available again. The classic operations *V* or *up* release the resource again and increment the counter or wake up waiting threads. The operations are *commutative*, i.e. the order of *P* and *V* operations does not matter for correctness. In contrast to mutexes, related *P* and *V* operations on semaphores can be done by different threads. Semaphores are always using blocking synchronization and therefore require support by the operating system kernel.

In POSIX, the semaphore API is not part of the `pthread` library, but part of an older real-time API. Therefore, semaphores are available to non-threaded processes as well. Semaphores come in two flavors, *named* semaphores and *unnamed* semaphores. Today, named semaphores are typically implemented in an in-memory file system as small shared memory segments containing an unnamed semaphore.

The semaphore API in POSIX consists of the following functions:

- `sem_init` to set the semaphore's initial counter value,

- `sem_{wait|trywait|timedwait}` to decrement the semaphore counter and to wait on the semaphore if the counter is zero,
- `sem_post` to increment the semaphore and to wake up waiting threads, and
- `sem_getvalue` to get the current counter value of the semaphore.

When more than one thread is blocked waiting for a semaphore, POSIX requires the usual priority ordering, like in mutexes and condition variables. But POSIX also warns that semaphores might not be suitable for real-time applications because of possible priority inversions. The recommended mitigation resembles the priority ceiling protocol in the mutex specification [IEE17].

Counting semaphores are also available in ARINC 653. The queuing discipline supports both priority ordering and FIFO ordering. No means to prevent priority inversion is specified.

2.2.6 Barriers

A *barrier* is a synchronization construct that allows up to `count` threads to synchronize themselves and coordinate their further progress. A thread reaching the barrier has to *wait* until all other threads have reached the barrier as well. A typical implementation uses a counter to be decremented by each incoming thread. If the decremented counter is non-zero, the calling thread blocks. Otherwise, on zero, the last thread has arrived, and the last thread then wakes up all other threads. Note, since all threads are woken up at the same time, no internal ordering of the blocked threads is needed. POSIX also notes that applications using barriers "may be subject to priority inversion" [IEE17].

The barrier API comprises two function:

- `barrier_init` to initialize a barrier for a given number of threads, and
- `barrier_wait` to wait until all participating threads have reached the barrier. The last thread reaching the barrier wakes up all waiters.

2.2.7 One-Time Initializers

POSIX defines so-called *one-time initializers* to lazily initialize certain data structures or software components at runtime on first use. The first thread that calls a service comprising an one-time initializer performs the initialization. Other threads calling the service wait for the initialization to complete before proceeding. One-time initializers were introduced mainly for better support of shared libraries in a multi-threaded environment. Most applications typically only use a small set of the features of a shared library, but the shared library may require an extensive initialization that is deemed too expensive to do upfront if these features of the library are not used.

The POSIX API only consists of a static initializer and an `once` function to perform the initialization on the first call. Other threads calling this function *block* while any initialization is still ongoing, or just do nothing afterwards. The actual initialization must be handled by a function that is passed via a function pointer.

Again, the ordering of blocked threads during concurrent initialization attempts is not important, as all threads are woken up at the same time when the initialization finishes.

2.2.8 Queuing Ports and Buffers

Queuing ports and *buffers* are *unidirectional blocking message queues* with a configurable maximum buffer size and queue depth in ARINC 653 systems. Buffers are used inside a partition, while queuing ports handle the communication between different partitions. If either the queue is empty on a *receive* operation or full on *send*, both services allow the caller to wait for a new message to arrive (receiver side) or for the queue to become writable again.

Both mechanisms queue messages in FIFO order. But as queuing discipline for waiting threads, ARINC 653 supports both priority ordering and FIFO ordering. Unlike buffers, which have both sending *and* receiving ends in the same partition, queuing ports are either the sending *or* the receiving side endpoint of a communication channel and can have a different waiting discipline for sending and receiving side. Further, an ARINC 653 system ensures that a selected queuing port configuration (direction, buffer size, queue depth) in one partition matches its counterpart in another partition.

The queuing port API comprises the following functions:

- `qport_create` to open a message queue and to specify port direction, maximum message size, queue depth, and queuing discipline,
- `qport_send` to send a message in a given time,
- `qport_recv` to receive a message in a given time,
- `qport_clear` to discard any unreceived messages on the receiver side, and
- `qport_status` to retrieve the current number of messages in the message queue and the number of blocked threads.

The buffer API works likewise, except that there is no `clear` operation.

2.2.9 Sampling Ports and Blackboards

Sampling ports and *blackboards* provide non-buffering messages in ARINC 653 systems.

Sampling ports are a non-blocking message store for communication between partitions, with expiration of the validity of the stored data. The mechanisms allows to write a message and read it afterwards. Sampling messages have a built-in timer that starts when a message is written. The sampling message is marked *invalid* when the timer expires. Reading a sampling port never blocks, but always returns the message *validity status* as well. Like queuing ports, sampling ports are used as either sending *or* receiving endpoints in a partition.

The sampling port API comprises:

- `sport_create` to open a sampling port and to specify port direction, maximum message size, and refresh period,
- `sport_write` to write a new valid message,
- `sport_read` to read a message and get its validity, and
- `sport_status` to retrieve the validity of the last message.

Blackboards provide non-buffering messages for use inside an ARINC 653 partition. Messages are read and written, but in contrast to sampling ports, a blackboard shows the last message until the validity is *explicitly* cleared. Reading a blackboard may block until a valid message becomes available. Writing a new valid blackboard message also wakes all waiting processes.

The blackboard API comprises:

- `bb_create` to create a blackboard for a given maximum message size,
- `bb_display` to write a valid message to the blackboard and to wake up any waiting threads,
- `bb_clear` to clear the current message,
- `bb_read` to read the current valid message or to wait until a new message is displayed, and
- `bb_status` to retrieve the validity of the current message and the number of waiting threads.

For the remainder of this thesis, we only focus on blackboards and not on sampling ports, as sampling ports do not represent a blocking synchronization.

2.2.10 Events

For event APIs, we distinguish between events in AUTOSAR and ARINC 653.

In AUTOSAR OS, *events* are the only notification mechanism. Each thread in AUTOSAR has a set of events it can wait for. Each event is represented by a bit in a bitmask, and the bitmask is typically 32 bits wide on 32-bit CPUs. An event is considered *pending* if its bit is set, otherwise the event is *cleared*. A thread can wait for any combination of its events to be set pending by other threads. While waiting, a thread has two bitmasks to consider, the set of pending events, and the set of events a thread is waiting for. Due to the internal bitmasks, we will further refer to the AUTOSAR event mechanism as *eventmask*.

The event API in AUTOSAR consists of the following functions:

- `eventmask_set` to set a set of events in the event mask of a target thread to pending state and wake the thread up if it is currently waiting for one of these events,
- `eventmask_wait` to check if any bit of a given bitmask is already set in the calling thread's mask of pending events, and to wait for any of these events to be set by other threads if not,
- `eventmask_get` to retrieve the bitmask of pending events of a thread, and
- `eventmask_clear` to clear the bitmask of pending events of the calling thread.

Implementations often provide a combined `eventmask_wait_get_clear` function to atomically wait for specific events, get the set of then pending events, and clear the events after reading.

In ARINC 653, events are implemented differently. Events exist independently of threads, and each event can be addressed separately. A thread can only wait for one event at at time, but more than one thread can wait for the same event to be signaled. When an event is signaled, all waiting threads are woken up.

The event API in ARINC 653 provides similar functionality and comprises:

- `event_create` to initialize an event in non-signaled state,
- `event_set` to set an event and wake up all threads,
- `event_wait` to check if the event is already set, and to wait with timeout for the event to be set by other threads if not,
- `event_get` to retrieve the state of an event,
- `event_clear` to clear (reset) an event, and
- `event_status` to retrieve the current status of an event and the number of blocked threads.

The event system in ARINC 653 could be easily extended to support event masks as well. However, due to the strict binding of events to threads in AUTOSAR, ARINC 653 events are more flexible than AUTOSAR events.

2.2.11 Futexes

Fast user space mutexes (futexes) are a *compare-and-block* synchronization primitive of the Linux operating system kernel. Futexes allow a thread to *wait* on a variable in user space or to *wake up* a given number of waiting threads. The kernel dynamically creates an internal wait queue based on the given user space address and keeps the wait queue as long as threads are waiting. The third conceptual futex operation allows to *requeue* waiting threads from one wait queue to another.

The original goal of the futex design was to implement fast mutexes in user space. The frequent case of uncontended locking and unlocking operations can be implemented by an atomic operation on a variable in user space, and the operating system kernel provides a blocking primitive for the contended case [FRK02].

Nowadays, futexes are the underlying mechanism to implement all POSIX thread synchronization objects in user space. Due to the dynamic handling of wait queues, the futex concept does not enforce restrictions on the number of synchronization objects or on the number of blocked threads, nor does it require prior registration of synchronization objects in the kernel.

The futex API in Linux supports two different futex types, *generic futexes* and *mutexes*. Each operation needs the address of the variable in user space, the eponymous *futex*, which must be a 32-bit variable. Based on its address, the operations first check whether a wait queue for the futex exists, and create a new wait queue on demand, if necessary. The operations on generic futexes are:

- `futex_wait` to wait on a futex with a given timeout if the current value of the futex still matches a given value,
- `futex_wake` to wake up a given number of threads,
- `futex_requeue` to wake up a number of threads and to move a number of waiting threads to a second wait queue,
- `futex_cmp_requeue` to wake up and requeue threads after a successful compare step,
- `futex_wait_bitset` to wait on a set given by a bitmask like in `futex_wait`,
- `futex_wake_bitset` to selectively wake up threads matching the bitmask, and
- `futex_wake_op` to wake up threads waiting on up to two different futexes after atomically modifying the second futex and checking if a condition still holds for the second futex.

The last five operations were introduced to improve support of condition variables. When signaling condition variables, instead of waking threads up and letting them compete to lock the associated mutex, the requeue operations transfer waiting threads from the condition variable's wait queue to the mutex's one and thus prevent *thundering herd effects* [HG09]. Waiting and wake-up of a subset of waiting threads and the `futex_wake_op` operation can also help in certain implementations of condition variables [9].

The Linux API also offers a specialized set of operations for the second futex type, futexes representing mutexes. In this case, both user space and kernel need to understand the mutex protocol in the futex value. The futex value encodes two pieces of information: the thread ID (`TID`) of the current lock holder or `NULL` if the mutex is free, and a `WAITERS` bit if the mutex has contention. The `WAITERS` bit is set by the first waiting thread upon detection of contention and by the kernel when handing over mutex ownership to the next thread, and it remains set as long as the wait queue is in use. The Linux kernel additionally supports *priority inheritance (PI)* mutexes for threads using POSIX real-time scheduling. Operations on condition variables need special variants as well to support requeuing to a PI mutex:

- `futex_lock_pi` to wait on a mutex with a given timeout if the mutex is locked,
- `futex_trylock_pi` to try to lock a mutex atomically,
- `futex_unlock_pi` to unlock the mutex and to hand over mutex ownership to the next waiting thread,
- `futex_cmp_requeue_pi` to wake up at most one thread and to move a given number of remaining threads, and
- `futex_wait_requeue_pi` to wait on a condition variable for later requeuing to a mutex.

For brevity, we will now omit the `_pi` suffix in the names of the operations.

Lastly, Linux distinguishes between *shared* futexes that can be placed in shared memory segments and shared between different processes, and *private* futexes that are restricted to a single process. The selected mode affects the kernel's indexing mechanism to locate wait queues: for shared futexes, the kernel uses the underlying physical address of a futex variable, while for private futexes, it can use the virtual address in user space.

Finally, *robust futexes* provide a lightweight means to notify pending waiters on a crash or deletion of the lock holder.

2.2.12 Adaptive Mechanisms

Another important mechanism often found in synchronization for non-real-time systems is to use a *two-phase synchronization scheme*. This describes a hybrid

[9]Linux manual pages, `http://man7.org/linux/man-pages/man2/futex.2.html`.

of spinning and blocking. A thread first spins for a short time, and if it cannot acquire the lock, it blocks. Ousterhout noted that the time for spinning should be at least twice the time of a context switch [Ous82]. *Adaptive locking* techniques consider further conditions on lock contention. *Adaptive mutexes* in Solaris [Sun02] assume that critical sections are short and spin for a short time when the lock owner is currently running on a different processor, or block if the lock owner is running on the same processor or not running at all. Mutexes using `PTHREAD_MUTEX_ADAPTIVE_NP` in Linux [Kyl14] spin for a short but variable time derived from previous attempts to acquire a resource.

In both cases, adaptive locking techniques are *opportunistic* approaches, as they do not ensure fairness. Adaptive mechanisms can be found in both mutex and reader-writer lock implementations.

We do not further consider adaptive mechanisms in this thesis.

2.2.13 Waiting With Timeouts

Depending on the API, most blocking primitives[10] allow to specify a *timeout*. We can further classify timeouts as either *relative timeouts* or *absolute timeouts*. A relative timeout t_r is specified as delta to the current system time t_{call} at the time of the call, resulting in an absolute expiry time $t_a = t_{call} + t_r$. Absolute timeouts specify the absolute expiry time t_a directly. We further denote the current system time as t and say that a timeout is expired when the expiry time is in the past, i.e. $t_a \le t$. The definition of the origin of time t_0 depends on the operating system.

POSIX has a concept of different *clock sources*, like a *wall clock* that refers to the current system time and may experience forward or backward *jumps* in time, e.g. daylight saving or to compensate for *clock drift* to a reference clock, or a *monotonically increasing clock* without these jumps that is suitable for real-time usage. Actual implementations often define further clock sources, e.g. clocks that ignore wake-ups when the system is in energy saving mode, or *coarse clocks* that can be read faster but with a coarser granularity. ARINC 653 only specifies a monotonic clock.

Most operating systems support timeouts with *nanoseconds resolution* in the API. But the actual timer implementation often has a much coarser *granularity*, as timers are seldom clocked at gigahertz speed. Furthermore, an implementation may program an even coarser time value as expiry time for the next *timer interrupt*[11]. The actual timer granularity can be retrieved in most operating system APIs to adjust for rounding errors.

[10]Except barriers and one-time initializers. These use an infinite timeout.

[11]Most timer implementations today operate in *one-shot mode* and program the expiration time of subsequent timer interrupts with a small safety margin to prevent system overload by too many interrupts. For non-real-time systems, a low resolution *periodic timer* is often already sufficient.

For the rest of the thesis, when referring to synchronization functions, we assume that the base operations support a timeout, e.g. we will write `cond_wait` instead of `cond_timedwait`.

2.3 Synchronization Inside an Operating System Kernel

This section presents a short overview of synchronization mechanisms used *inside* an operating system kernel. Note there is no standard for the internal operations inside an operating system kernel but we typically find the functionality discussed here. Locking at kernel level is different to locking in user space, mostly because execution in privileged mode allows *kernel threads*[12] to have more control about the execution environment, e.g. to temporarily disable interrupts. Also, different API conventions are used. We often see a *decomposition* of synchronization mechanisms into simpler lower-level mechanisms, as there is no need to consolidate multiple functionality into a single API call like user space APIs often do for performance reasons. Lastly, programming models inside operating systems often mandate that internal APIs are used correctly to remove or minimize superfluous error checks.

2.3.1 Level of Indirection

In an operating system, we can observe a different *level of indirection*. Kernel code can *directly* address a thread by a pointer to the *thread control block (TCB)*. This is not possible in a user-level API for robustness reasons. User APIs often relate to kernel objects by (at least) one level of indirection, such as *handles* or *file descriptors*, which are kept in a *global namespace* or in process-specific *local namespaces*. The look-up of a kernel object from its handle is often realized by *multi-level look-up tables* similar to *page tables* in $\mathcal{O}(1)$ time. Other mechanisms for indirection are *hash tables*, *dynamic arrays*, and *binary search trees*. Sometimes special constraints must be followed, e.g. POSIX requires *linear allocation* of file descriptors. When opening new files, an implementation must always return the lowest-numbered free file descriptor. This impacts scalability [CKZ+13].

2.3.2 Queuing

Queues of blocked or waiting threads are realized as linked lists or binary search trees. As a rule of thumb, when a queue needs FIFO ordering or sorted entries, but

[12]We assume a model similar to the Linux kernel where each thread in kernel mode has a dedicated stack and can block at arbitrary blocking points. In an implementation where all threads of a processor share the same kernel stack, blocking is only viable after a thread releases the stack, i.e. completes its current operation and returns.

the number of elements in the queue is small, linked lists are sufficient. Otherwise, we see balanced binary search trees, like *red-black*, *AA*[13], or *AVL* trees, when a high degree of predictability is needed. Also, specialized data structures with different trade-offs between performance and complexity can be found, such as *timer wheels* [VL87]. Lastly, there are varying assumptions on the usage patterns. For example, connection timeouts in a network stack will almost always never expire and nodes will be removed early. Or nodes will change their relative position in a queue, e.g. when a waiting thread's scheduling priority is changed.

2.3.3 Locking and Preemption Control

In an operating system targeting today's multicore chips, a locking model comprising just a *big kernel lock* (or *giant kernel lock*) has fallen out of favor due to its limited scalability, but seems to remain acceptable in some niches [PDEH15]. Instead, we often see *fine-grained locking schemes.* To support *nested locking*, i.e. incrementally locking two or more locks, the operating system environment must define a *strict order* in which different classes of locks can be acquired to *prevent deadlocks.* Similar restrictions apply to *hand-over-hand locking*, which is used to lock nodes during traversal of tree-like structures.

In an operating system, we see both suspension-based locks and spin-based locks. Suspension-based locks are used for *longer* operations. They either are *fully preemptive* or have explicit *preemption points.* Spin-based locks are used for *short* critical sections. They require *non-preemptive execution* to prevent *lock holder preemption* problems, see Section 2.1.3. There are two mechanisms to prevent lock holder preemption: (i) temporarily *disable interrupts*, or (ii) allow interrupts, but temporarily *disable preemptive scheduling.* Which mechanism is eventually used depends on the users of the lock: if there is the possibility that the lock is acquired from an interrupt handler, the lock must disable interrupts, otherwise disabled preemption is sufficient.

To control interrupt handling and for preemption control, we use the following names and conventions for the in-kernel operations:

- `irq_disable` to disable local interrupts,
- `irq_enable` to enable local interrupts,
- `irq_restore` to restore a previous interrupt state,
- `preempt_disable` to disable preemption,
- `preempt_enable` to enable preemption,
- `preempt_restore` to restore a previous preemption state,

[13]AA trees, named after their inventor *Arne Andersson*, are a variation of the red-black trees with simpler insertion rules but more tree rotations [And93].

- `preempt_is_pending` to check if preemption is pending, and
- `preempt_point` to insert an explicit preemption point.

2.3.4 Waiting and Wake-up

Finally, when a kernel thread needs to wait inside a critical section, it cannot simply call the blocking function of the scheduler, but must release the critical section first. To prevent the problem of *missed wake-ups*[14], the thread sets a *wait indicator* flag[15] inside the critical section, releases the critical section and then calls the blocking function. The blocking function of the scheduler in turn acquires a scheduler lock and evaluates the flag again. Only if the flag still indicates waiting, the thread is suspended. Conversely, a wake-up operation first clears the wait indicator flag inside a critical section, and then calls the scheduler. The wake-up function checks the state of the target thread under the scheduler lock and wakes up the target thread if it is really suspended. In any combination, the wait indicator flag transfers state information between two non-nested critical sections.

For the low-level scheduler API, we define the following functions:

- `sched_wait` to block or wait with a timeout in the kernel, and
- `sched_wakeup` to wake up a blocked or waiting thread.

[14]A second thread locks the critical section and then wakes up the first thread *before* it actually reaches the scheduler.

[15]The logic of this flag follows Reed's *eventcounts* [Ree76].

Chapter 3

Analysis

As goal of this thesis, we want to have *both* efficient *and* predictable blocking synchronization mechanisms for mixed-criticality systems. But as a starting point, we can only make two contradicting observations: *efficient synchronization mechanisms are often not predictable*, and *predictable synchronization mechanisms are often not efficient*.

To achieve both goals, we must first define what both efficiency and predictability means. When talking about efficiency, we want to improve the performance of blocking synchronization mechanisms. We start with a review of related work on techniques that make synchronization mechanisms more efficient in Section 3.1.

At the same time, changes to these low-level mechanisms often pose trade-offs at design level and involve contradicting concerns. Our goal is to *improve the average-case performance*, but *without sacrificing the worst-case behavior* too much. In Section 3.2, we first introduce our *system model* and discuss the overhead of system calls and relative costs of CPU Instructions. Also, we present *requirements for predictability* and define the necessary *metrics of efficiency*.

In Section 3.3, we analyze applicable real-time protocols, identify common building blocks, and discuss potential shortcuts.

Next, in Section 3.4, we will then analyze the blocking synchronization mechanisms, and decompose them into their basic building blocks as well. With this, we define a generic model of blocking synchronization mechanisms.

State-of-the-art approaches to improve efficiency include *futexes* and other mechanisms to avoid system calls in common use case scenarios. Section 3.5 analyzes how futexes in Linux provide a fast path for blocking synchronization mechanisms. We also discuss predictability problems in the Linux kernel implementation that prevent the use of Linux futexes in safety-critical environments.

Section 3.6 compares different approaches to blocking synchronization mechanisms and identifies an alternative approach for an efficient synchronization mechanism. The resulting *monitor* approach is based on non-preemptible critical sections in user space with according suspend and wake-up mechanisms.

For non-preemptible critical section in user space, we then discuss state-of-the-art mechanisms for preemption control in Section 3.7.

We also discuss the constraints of low-level wait and wake-up mechanisms with direct addressing in Section 3.8.

Finally, Section 3.9 summarizes the analysis for efficient synchronization mechanisms in user space and their building blocks, namely *deterministic futexes* and *non-preemptive busy-waiting monitors*.

3.1 Related Work

Efficient synchronization mechanisms often exploit a *fast path for the common case* or *lazily* postpone the costs of some actions into the future in the hope that the won't be needed. We discuss related work on the topic.

3.1.1 Futexes and Fast Synchronization Mechanisms

The observation that system call overheads are expensive is well known. Several approaches were already proposed to improve the efficiency of synchronization mechanisms in historical systems: Keedy describes atomic *test-and-increment* and *decrement-and-test* operations on the *ICL 2900* computer to implement *P* and *V* operations on positive semaphores values without the need to call into the operating system kernel [Kee77]. A similar approach is described with *Benaphores* in *BeOS* [Sch96]. Birrell et al. describe an optimization for mutexes, condition variables, and semaphores in the Taos operating system to only call the operating system kernel when there is contention or a thread is blocked on a condition variable [BGHL87]. For synchronization in a Java virtual machine, Bacon et al. proposed *Thin Locks*, based on atomic operations for uncontended cases, with a fall-back to OS provided synchronization primitives [BKMS98]. Similar approaches where the kernel is entered only on contention are used by *Critical Sections* in Windows [RSI12].

Futexes extend these prior approaches as a generic *compare-and-block* mechanism. They were first introduced in Linux to implement POSIX thread synchronization objects in user space [FRK02, DM03, Dre11], and then later refined for real-time use cases using PIP [HG09]. Over time, scalability issues were addressed and discussed [BN14, Bro16]. Zuepke et al. presented approaches for deterministic futexes with FIFO ordering based on doubly-linked lists and look-up by thread ID [Zue13], and futexes using an index-based wait queue look-up [ZBL15]. Pizlo described an approach resembling futexes using cascaded locks and hashed wait queues in user space for fine-grained locking and condition variables in the WebKit browser [Piz16].

Another technique is to share data between user space and kernel to prevent system call overheads: Wisniewski et al. describe *scheduler-conscious* synchronization mechanisms in user space that share information across the application-kernel interface to prevent preemption by the kernel [WKS95]. Linux's vDSO [1] and L4's user-level TCB [LW01] map a page into user space to share information, such as the current thread's control block and IPC arguments (L4), or current CPU and system time (Linux). Spliet et al. evaluated the use of PCP, MPCP, and FMLP$^+$ for futexes in the context of LITMUS$^{\mathrm{RT}}$, a Linux-based testbed for real-time scheduling experiments [SVBD14]. The implementation uses an index-based wait queue look-up and a *lock page* shared between user space and kernel. The lock page contains a bitmap of acquired locks.

Besides futexes, Bershad et al.'s *restartable atomic sequences* for processor architectures that do not provide atomic operations [BRE92] describe a similar *optimistic* technique that favors a fast path at the cost of having to perform more work elsewhere. Here, when the kernel detects that a thread is interrupted inside an atomic sequence, it changes the instruction pointer of the thread to *restart* the atomic sequence once the thread resumes. This technique simplifies the implementation of atomic operations at the cost of extra checks during interrupt and exception handling.

But even light-weight locking mechanisms still cause overheads on multi-processor systems due to the required atomic operations and synchronization barriers, see Section 2.1.8. With *biased locking*, Kawachiya et al. presented a techniques to overcome these costs in the context of a Java Virtual Machines (JVM) [KKO02,Kaw05]. Exploiting thread locality, the JVM reserves a lock for a thread frequently using it. When a second thread appears, it breaks the bias, and the JVM falls back to conventional locking.

3.1.2 Lazy Techniques and Optimization

In the context of the L4 microkernel, Liedtke presented *lazy scheduling* [Lie93]. In synchronous microkernel systems like L4, threads often call servers that reply quickly, causing frequent removal and addition of threads to the ready queue. With lazy scheduling, threads remain in their place on the ready queue when they are blocked in IPC calls. Any blocked threads will be finally removed from the ready queue when the scheduler is invoked the next time, for example when the current time slice runs out.

However, Blackham et al. reported that this optimization to the IPC path had a negative impact on WCET analysis in seL4, an L4 variant with a formal proof of correctness [KEH$^+$09], as a scheduling operation has to handle $\mathcal{O}(n)$ blocked threads in the worst case, and lazy scheduling was removed. Instead, the seL4 developers used a different trick internally known as *Benno scheduling*, where an IPC operation switches directly to the target thread instead of placing the thread on the ready queue first. As result, the currently executing thread is not kept on

[1]Linux manual pages, `http://man7.org/linux/man-pages/man7/vdso.7.html`.

the ready queue in seL4 [BSC+11, BSH12]. The same technique is used in PikeOS to optimize fast priority switching.

In the context of operating systems, *lazy FPU switching* is another well-known technique. On a context switch, the operating system skips saving of *floating point unit (FPU)* registers and simply disables access to FPU registers for the next thread, based on the assumption that the next thread will not use FPU registers and the control flow switches back to the previous thread soon. If the next thread accesses FPU registers, the hardware will generate an *FPU unavailable exception* and the operating system will finally perform the context switch of the FPU registers. This technique optimizes the average context switch time when threads use the FPU rarely, at the cost of an extra exception when FPU registers are actually accessed.

3.2 Settings and Requirements

3.2.1 System Model

In this thesis, we consider an operating system that supports real-time scheduling for time-critical applications and optionally best-effort scheduling for other non-time-critical applications. We assume that different applications are hosted, and that some kind of resource partitioning is available to prevent threads to over-commit on resource usage. This is typical for complex systems in the automotive industry and for avionics. As such systems can be *updated* after deployment, the exact number of partitions and threads per partition might not be known at design time or initial integration time. To simplify the timing analysis when updating the system, the operating system should ensure that adding new partitions and threads has no impact on the timing analysis for existing partitions. This concept is known as *incremental certification* in avionics [WP09]. We simply assume that the system has m processors and that there are *at most* N threads in the system, i.e. N is a reasonable upper bound.

As thread scheduling algorithm, we assume *partitioned fixed-priority (P-FP) scheduling*, which is also mandated by both automotive (AUTOSAR) and avionics (ARINC 653) OS standards and used by real-time POSIX applications. Here, in an *offline* step, the *system integrator* at first assigns real-time threads to the available processors (or time partitions), as long as the processors (or time partitions) have remaining capacity left, and then assigns a static *priority* to each of the threads, as described in Section 2.1.4. The fixed assignment of threads to processors reduces the complexity of multiprocessor scheduling to the well known problem of single processor scheduling with its analysis techniques. For convenience, we further assume that higher priority values also reflect a higher scheduling priority. The scheduler will always select the thread with the highest priority as current thread. When threads share the same priority, they will be scheduled in FIFO order. Further, thread scheduling is *preemptive*, that is, an arriving higher priority thread interrupts the currently executing lower priority thread. Also, to prevent preemption, threads can temporarily raise their effective scheduling priority at runtime above higher priority threads, for example while entering critical sections.

For the internal design of the operating system, we assume a *threaded kernel* model, i.e. threads executing inside the kernel have their own dedicated stack and can block or wait at any time, while ISRs have *run-to-completion* semantics and simply use the current thread's stack as well. We can further consider ISRs as a class of highest priority threads that can always preempt all other threads. An incoming interrupt can be handled in its ISR, or the ISR wakes up a normal thread for further interrupt handling (*threaded interrupts*).

For synchronization mechanisms in user space, we assume the relevant synchronization mechanisms described in Section 2.2. Further, we assume that the operating system supports one or more real-time locking protocols to address priority inversion problems.

For internal synchronization inside the operating system, we assume a fine-grained locking model using spinlocks (busy-waiting) and mutexes (blocking), as described in Section 2.3. We further assume that critical sections with spinlocks are not preemptive, and critical sections that can be accessed by both ISRs and threads must be protected by disabling interrupts. Also, the operating system uses *inter-processor interrupts* to notify remote processor cores about rescheduling updates.

We further assume that the operating system provides means to address interference problems of shared hardware, for example by partitioning shared caches or providing a cap on a processor's bandwidth on a shared memory controller. Techniques are described in Section 2.1.9. For the remainder of this thesis, we only focus on *interference at the software level inside the operating system* due to sharing of data between partitions. We assume that the operating system avoids any unwanted interference between unrelated partitions, e.g. prohibits (even read-only) access to other partitions' resources and has no *false sharing* of data in caches. This addresses concerns about termination of atomic operations.

Regarding safety and security aspects in the system, we define the following convention. The general system design of a mixed-criticality system as described in Section 2.1.7 already implies a system design that is *restrictive by default*, i.e. access to resources must be explicitly granted. Therefore, any aspects that impact the functional behavior of a software component are considered relevant for safety. This includes any *direct effects*, such as modification of data in a partition or change to state in the kernel that alters the behavior of the partition, and any *indirect effects*, such as the temporal impact of operations. We expect that the WCET of all operations is bounded and known to the system integrator. As relevant for security, we define any side effects which can be exploited for unintended communication (covert channel) or to infer secret information of others (side-channel attack). In short, safety covers all aspects that *manipulate* data. For security, only aspects with an *observing* character remain. As we mainly deal with safety-critical systems, this simplification is acceptable in the context of this thesis.

3.2.2 Relative Costs of CPU Instructions

As a rule of thumb, we can assume that atomic operations on data in the L1 cache need *one order of magnitude* more time than simple ALU operations. For example, Al Bahra measures about 15 cycles for an atomic CAS operation on an Intel Core i7-3615QM [Al-13]. The overhead is mainly due to the guarantee of atomicity of CAS on the x86 architecture.

Similarly, we assume that system calls take at least *two orders of magnitude* more time than ALU operations. Soares and Stumm describe a system call overhead of around 150 cycles on an earlier Core i7 generation [SS10]. Ousterhout observed for contemporary processor architectures of the 1980s that *[. . .] operating system performance does not seem to be improving at the same rate as the base speed of the underlying hardware* [Ous90]. If we assume 16 MHz clock speed for the MIPS R2000 CPU in Ousterhout's paper, the presented system call overhead for `getpid` would be about 300 cycles. For current Intel CPUs, Elphinstone and Heiser note that the best-case IPC performance of seL4 is around 300 cycles on a Core i7-4770 [EH13]. The IPC operation is a non-trivial example of a system call, but we can consider that the authors applied lots of optimization, as their results for other architectures show, but still, at least half of the costs here is overhead.

Calls into the operating system kernel are expensive, but not only because of the synchronous trap and overheads to save and restore registers, but also because of the negative impact on performance due to pipeline flushes [SS10] and mitigation against processor design flaws such as *Meltdown* [LSG+18] and *Spectre* [KHF+19].

While we can assume that processor design flaws like Meltdown will be fixed in future processor generations, Spectre-like attacks on branch prediction by data cache side channels are expected to stay [MST+19]. As mitigation, an operating system must flush the branch predictor state on entry to supervisor mode and on context switches. Therefore, we cannot assume that the overhead for system calls improves soon.

Summarizing, for this thesis, we can assume that ALU operations are much faster than atomic operations, and atomic operations are much faster than system calls including operating system overheads:

$$t_{ALU-operation} \ll t_{atomic-operation} \ll t_{system-call}.$$

3.2.3 Requirements for Predictability

As a general rule, synchronization mechanisms for real-time systems that target certification must be predictable, i.e. have a WCET that is *analyzable* and *bounded* in the first place. Likewise, the synchronization mechanisms must properly address *interference problems* caused by sharing of resources. This includes shared memory, namespaces, and related data structures, e.g. global identifiers for processes. We thus require *absence of interference* for private mechanisms not shared between partitions, and we require at least *bounded interference* for any shared mechanism, as any shared resources can be accessed by all involved partners in parallel. These general requirements help to reduce efforts in software certification following avionic standards, as they allow to analyze independent software components in isolation.

We will now discuss these requirements *RQ1* to *RQ9* in detail. The requirements will later guide the design and the implementation of the synchronization mechanisms. In general, a certifiable design requires more scrutiny of corner cases and often sacrifices best-case performance for robustness and determinism.

The first two requirements *RQ1* and *RQ2* address general properties of correctness and robustness. The next four requirements *RQ3* to *RQ6* deal with properties for analyzability. The last three requirements *RQ7* to *RQ9* discuss long-running operations that should be preemptible. The requirements were adapted from previous work of the author [ZK19].

***RQ1* Correctness:** Obviously, a synchronization mechanisms must be *correct* and guarantee its desired properties, e.g. mutual exclusion, fairness, or freedom of starvation.

***RQ2* Robustness:** This requirement calls for robustness against *unexpected failures* and *accidental misuse* (or even abuse) of a mechanisms. For example, the dynamic memory allocations are a source of unexpected failures, as dynamic memory allocations can fail at runtime. Accidental misuse is particularly important for waiting synchronization mechanisms with a timeout, where *second order effects* already become visible. For example, a malicious partition could let a number of threads wait on a timeout and then let the timeouts expire in a certain pattern so that the next timeout always expires shortly after the previous one was processed. The resulting storm of timer interrupts could then affect scheduling of unrelated threads.

***RQ3* Analyzability:** The specification and the design of synchronization mechanisms must be *unambiguous*. Also, the specification must be *complete* so that all corner cases can be rigorously tested. An implementation must then precisely follow the specification. It should not add another level of complexity that is not covered by the specification. Otherwise one does not get the necessary coverage in software testing. Analyzability also aims to *reduce unnecessary complexity*, as simpler implementations need less variables in an analysis. Also, having fewer dependencies on other components simplifies the WCET analysis and certification.

RQ4 **Bounded operations on queues:** Queues of various types will be used for different tasks, e.g. ready queues, timeout queues, or wait queues. We require that operations *on* the queues, i.e. *insert* and *remove* to add and remove nodes, and $find$, min, and max to locate nodes in a queue, must be bounded operations w.r.t. the number of nodes n in the queue. As we require that the specification is unambiguous, all queues must have a defined queuing discipline.

For FIFO- or LIFO-ordered (*last-in first-out*, a *stack*) queues, when the implementation uses doubly linked lists, all operations except $find$ are in $\mathcal{O}(1)$ time regardless of n. With further restrictions, e.g. when no arbitrary nodes need to be removed, even singly linked lists would be acceptable.

For queues ordered by a single key, such as priority-ordered queues, linked lists are often problematic, because *insert* takes $\mathcal{O}(n)$ time. Operations in $\mathcal{O}(n)$ time are acceptable for predictability *if* the upper bound is both known and small, but this is often not the case for the synchronization mechanisms exported to user space as described in Section 2.2. For large or unknown n, an implementation can better use *self-balancing binary search trees (BSTs)* as discussed in Section 2.1.6. Binary search trees, such as *red-black*, *AA*, or *AVL* trees, bound the complexity of their operations to $\mathcal{O}(\log n)$ time. Also, logarithmic complexity often allows to perform an analysis of an overall system without having detailed knowledge about all applications. For example, for wait queues, an upper bound for n (blocked threads) can be easily approximated from system limits, e.g. the amount of available memory limits the number N of threads in a system.

RQ5 **Bounded loops and retries:** Like for the queue operations mentioned above, all loops in an implementation of a synchronization mechanism must be bounded. When processing multiple elements in a loop, e.g. wake up of all blocked threads, the overall number of loop iterations is bounded by the overall number of threads. But this bound might not be acceptable in some situations, for example from ISRs. Consider an example where a large number of threads set a timeout to expire at the same time. In this case, the kernel's timer interrupt handler has to wake up a large number of threads from the timer ISR. Doing this with interrupts disabled might have an unacceptable impact on the interrupt response time. It might be better to process only a small number of threads and become preemptive before processing the next set of threads.

Another important aspect are *bounded retries of atomic operations.* For example, an atomic increment operation must first *load* a value from memory, increment the value in a register, and then execute a CAS or SC instruction, which can fail due to concurrent updates of the data in memory or arbitrary writes to the same cache line due to false sharing. Bounded retries are also relevant to all kinds of atomic fast paths and shortcuts to bypass a potentially expensive operation.

RQ6 **Interference by shared namespaces:** The use of global namespaces that are shared between partitions can cause interference problems due to internal shared locks to ensure consistency of the namespace. For example, one partition could (accidentally or deliberately) cause an unwanted slowdown to a second partition by frequently accessing an object in the namespace. In general, global namespaces should be avoided, but this is not always possible, e.g. because of compatibility reasons. Process creation in Unix must lock the global namespace for process IDs to allocate a new process ID. The only way to address these problems properly is to virtualize a global namespace for each partition and effectively make it a private one. If this is not possible, access to global namespaces should be handled at boot time only or restricted to privileged software that can be rigorously analyzed. If the interference cannot be prevented, *fine-grained locking* or a *preemptive design* can help to bound the time of interference.

RQ7 **Preemptive operations:** When an operation takes a potentially long time, e.g. an operation that wakes up a large number of blocked threads, the operation should be preemptive. A good practice for the implementation is to become preemptive after processing a small number of elements, e.g. after wake up of each thread. For example, in case of timeout expiration, this allows a higher priority thread to become immediately eligible for execution and to remove itself from a wait queue. This is a good compromise with respect to the worst-case time an operation holds internal locks. The exact number of non-preemptible steps can be fine-tuned when a timing analysis is available. But preemptive operations cause side effects that require mitigation, as we will see in the following requirement.

RQ8 **Termination:** We require that any preemptive operation must eventually terminate. This is not always the case. For example, a preemptible implementation of a synchronization mechanism with a wait queue bears the risk that an already processed thread reenters a wait queue again and causes unbounded loops. This requirement is related to the bounded loop requirement *RQ5*, however the emphasis here is not to define an upper bound of the loop a priori, but to prevent unwanted modification of wait queues.

RQ9 **Hidden transience:** Another side effect of preemptive operations is that the already processed threads may observe the processing thread or the synchronization object in a transient state. To affected threads, a preemptible operation should appear to be atomic. In the context of synchronization mechanisms, if threads follow the usage constraints properly, preemption should be hidden to threads on a wait queue and to other threads operating on the same synchronization object. For example, when using condition variables, an application shall lock the support mutex before calling `cond_broadcast`. Also, an implementation must handle corner cases where already processed threads access the same synchronization object again, e.g. when using barriers and a woken up thread completes early and needs to wait for the next round. In these case, this thread must not be added to the existing wait queue (this would also violate the requirement about termination), but to a new one.

3.2.4 Metrics to Evaluate Efficiency and Predictability

As stated before, we want to improve the average-case performance of synchronization mechanisms, but without sacrificing the worst-case behavior. This means that we can effectively measure the average-case performance in a benchmark, and we can look at the worst-case factors that impact the WCET in an analysis.

For the benchmarks, we have two baselines to compare performance results to: (i) we can compare the performance of an implementation of a predictable synchronization mechanism against an optimized, but non-predictable implementation, such as Linux, or (ii) we can compare the performance of an optimized and predictable synchronization mechanism against a non-optimized, but still predictable variant. To compare the worst-case timing behavior, we can only use the predictable baselines. Comparing the predictable and non-predictable implementations would be like comparing apples to oranges. Still, the results show if an implementation is on par with other state-of-the-art mechanisms.

For benchmarking average-case performance, we can use typical hardware platforms that are also used in certified systems. This includes current desktop and embedded processors. We expect that the average execution time $\bar{e}$ of an optimized variant $\bar{e}_{opt}$ is smaller than that of the baseline variant $\bar{e}_{base}$. Note that these results will highly depend on the benchmark scenario.

However, comparing the actual WCET of two implementations using example scenarios is not a task we can easily do. We neither have the resources to create a test scenario to empirically assess a practical WCET bound, nor do we have the tools to model the WCET down to pipepine level to derive a theoretical WCET bound. Instead, we perform a timing analysis on an abstract level with symbolic execution times and present the WCET $e(n, m)$ as a formula for n threads, m processors, and constant terms of included critical sections $e_{CS,i}$.

In general, operations in an operating system kernel as described in Section 2.3 are not computationally complex, therefore we assume an actual WCET analysis to be dominated by data access and reduce the terms to the *individual number of accessed cache lines*. Our empirical results in [ZK19] have shown that cold-cache memory accesses dominate performance when traversing greater amounts of thread control blocks (TCBs) in an operating system kernel, as queue nodes are kept at the same offset within page-aligned TCBs, effectively causing trashing in the same cache set. Likewise, low-level system call code and context switching mainly comprise saving and restoring registers and therefore are dominated by data accesses as well, e.g. cache eviction on store and cache misses on load.

We further use the general observation that algorithms for queue operations with constant time $\mathcal{O}(1)$ are better than ones with logarithmic time $\mathcal{O}(\log n)$ and algorithmic time is better than linear time $\mathcal{O}(n)$ as rough guideline to evaluate approaches. This allows us to simplify an individual WCET term into a constant part t_{const} depending on n threads and m processors, e.g. $t_{linked-list} = t_{head} + n{\cdot}t_{per-node}$ for linked-list traversal, and then further replace the constant parts by the number of accessed cache lines, e.g. $t_{linked-list} = 1 + n$ in the previous example. This model is *not* accurate for a detailed WCET analysis and clearly neglects instruction cache fetches, but it should be sufficient to evaluate the *trends* when processing a large number of threads n and the interference by $m - 1$ other processors.

Using a timing analysis at such a level, we can now clarify how to evaluate an approach. We compare the polynomial WCET terms of an optimized variant e_{opt} to the baseline variant e_{base}. But as the polynomial WCET terms depend on two variables n and m, we cannot define a simple metric to compare the WCET terms. Therefore, we must discuss the differences individually.

3.3 Building Blocks of Real-Time Locking Protocols

We now analyze the different real-time synchronization protocols of Section 2.1.5 to decompose them into their key elements, identify the building blocks that must be supported by an operating system, and discuss potential fast paths.

Overview: As an overview, we briefly summarize the real-time synchronization protocols for P-FP scheduling described in Section 2.1.5 and categorize these protocols w.r.t. immediate access to uncontended resources, busy-waiting, blocking, and migration:

- *NPCS (non-preemptive critical sections)*: A thread disables preemption when accessing a local resource. Access is granted immediately; the thread never waits.
- *IPCP (immediate priority ceiling protocol)*[2]: A thread temporarily changes its priority to the ceiling priority of the local resource. No preemption by new arriving threads with lower priority. Immediate access, no waiting.

[2]We choose IPCP instead of PCP due to the simpler implementation.

- *PIP (priority inheritance protocol)*: A thread simply accesses a local resource and can get preempted. The thread inherits the priority of any higher blocked thread. Again immediate access and no waiting. Any further action is delegated to blocked threads on contention.
- *NPCS with spinlocks* uses non-preemptible FIFO spinlocks for global resources. No preemption, but busy-waiting before access.
- *MrsP (multiprocessor resource sharing protocol)* combines IPCP with FIFO spinlocks for global resources. Preempted lock holders are migrated to other spinning processors. Busy-waiting and migration.
- *DPCP (distributed PCP)*: A thread migrates to the resource's local processor and uses PCP with priority boosting locally. Immediate access is only possible when resource is accessed on the resource's local processor. Requires blocking and migration.
- *MPCP (multi-processor PCP)* uses priority boosting and PCP for access to global resources. Threads can get immediate access to uncontended resources due to immediate priority boosting, otherwise threads must block.
- *Multiprocessor PIP* with *local helping* and *migratory priority inheritance*: A thread immediately accesses a resource if it is free, otherwise the thread blocks. PIP and migration is applied by the blocked threads on contention.
- FMLP$^+$ uses priority boosting based on the time of the lock request and a FIFO-ordered wait queue on contention. Immediate access and blocking.

Similarities: When comparing these protocols, we see certain similarities. Most commonly, NPCS, IPCP, MrsP, DPCP, MPCP, and FMLP$^+$ temporarily disable preemption or change the priority of the requesting thread, and revert to the previous state after the thread releases the resource. The non-preemptibility of NPCS and MrsP can be alternatively implemented by using the highest possible priority in IPCP. Also, the priority boosting in MPCP and DPCP can be implemented as an additional priority band above normal priorities, or by splitting the normal priority range in two halves. Lastly, IPCP, DPCP, and MPCP require the definition of ceiling priorities for the resources.

DPCP, multiprocessor PIP, and MrsP require migration. DPCP requires this unconditionally, while PIP and MrsP require this only when blocked threads detect that the lock holder is currently preempted.

Of the blocking protocols, single- and multi-processor PIP, DPCP, and MPCP require a priority-ordered wait queue, while FMLP$^+$ requires a FIFO-ordered wait queue. The busy-waiting protocols use FIFO ordering.

PIP in any form gives immediate access to an uncontended resource and applies any protocol-specific mechanisms to resolve the priority inversion only on contention, i.e. in the context of blocked threads. However, PIP internally requires to build a *resource allocation graph* to correctly track the dependencies of nested critical sections and locate lock holders to apply priority inheritance to [ZUW20].

Categorization: In their work on futexes in LITMUS^{RT}, Spliet et al. categorized protocols such as PCP, MPCP, and FMLP^{+} as *anticipatory*, as they take actions to reduce priority inversion *before* issuing a lock request [SVBD14]. NPCS, IPCP, MrsP, and DPCP are also anticipatory protocols. In contrast, PIP is categorized as a *reactive* protocol, as it only becomes active on contended lock or unlock operations.

We introduce a second category whether a protocol provide *immediate access* to uncontended resources. All protocols besides DPCP fall into this category, as DPCP might first migrate the resource requesting thread to the target processor. Note that the PCP protocol described in Section 2.1.5 also does not fit into this category, as the protocol might refuse access to an uncontended resource due to the priority ceiling rules.

Identification of potential fast paths: PIP in any form fits well to futexes, as the protocol (i) allows immediate access to an uncontended resource, and (ii) is activated by the blocked threads on contention, and the blocked threads need to perform system calls for blocking and wake-up anyway. Also, migration in PIP in the multi-processor variants of the protocol also happens only due to contention. Therefore, no additional mechanisms are needed here.

For the anticipatory protocols NPCS, IPCP, MrsP, DPCP, MPCP, and FMLP^{+}, an efficient mechanism to manipulate the scheduling priority of the currently executing thread in user space would allow to implement the fast path for uncontended `lock` and `unlock` operations completely in user space.

This applies partly for DPCP as well, but only if the requested resource is on the current processor. For this, a thread must know on which processor it is currently executing on. Otherwise, migration must be handled by the operating system kernel.

Likewise, MrsP requires migration of preempted lock-holders on contention. This additionally requires a robust mechanism to detect lock-holder preemption in user space to prevent superfluous spinning.

3.4 Building Blocks of User-Level Blocking Synchronization

In Section 3.4.1, we now analyze the blocking synchronization mechanisms described in Section 2.2 from a high-level point of view. We discuss similarities and identify the underlying *elementary synchronization mechanisms* that compose all user-level blocking synchronization mechanisms. We then classify the elementary synchronization mechanisms and provide a generalized form that hides differences. In Section 3.4.2, we dig deeper into the implementation level inside an operating system kernel to identify the basic low-level building blocks and discuss their interworkings and implications.

3.4.1 High-Level Analysis and Generalization

Overview: At start, we briefly summarize the blocking synchronization mechanisms of Section 2.2 w.r.t. their type, semantics, blocking and wake-up behavior, and queuing disciplines. The list does not include spinlocks and sampling ports. Both do not block.

- *Mutexes*: `mutex_lock` acquires the resource and blocks the caller with a timeout when the resource is contended. `mutex_unlock` releases the resource and wakes up at most one thread. Blocked threads are queued in priority (POSIX) or FIFO order (ARINC 653-specific and configurable). Support for different mutex types consists of additional error checks and a recursion counter. Support for IPCP and PIP.

- *Condition variables*: Require a locked support mutex. `cond_wait` lets the caller wait with a timeout. The function unlocks the support mutex before waiting. After waiting, the support mutex might be locked or not. `cond_signal` and `cond_broadcast` requeue one or all threads from the condition variable's wait queue to the wait queue of the support mutex. Spurious wake-ups are acceptable. Waiting threads are queued in priority order.

- *Reader-writer locks*: A `lock` operation acquires the resource in the requested mode and blocks the caller with a timeout. An `unlock` operation releases the resource and wakes up one writer or all reader threads. Blocked writers are queued in priority order, while blocked readers are unordered.

- *Counting semaphores*: Semaphores have an internal counter. `sem_wait` decrements the counter and blocks the caller with a timeout when the counter resource was not greater than zero. `sem_post` increments the counter and wakes up at most one thread. Waiting threads are queued in priority order.

- *Barriers* and *one-time initializers*: The operations allow the caller to wait. The last resp. first thread wakes up all waiting threads. There is no special order of waiting threads.

- *Queuing ports* and *buffers*: Classic *bounded-buffer* problem. Two blocking points if message queue is empty or full. A `send` or `recv` operation blocks with a timeout if the queue is full/empty; then performs a message transfer; then wakes up at most one thread. Blocked threads are queued in priority or FIFO order (configurable). The message transfer involves a data copy operation that must be serialized to ensure the FIFO ordering of messages.

- *Blackboards*: Barrier with message transfer. `bb_read` allows the caller to wait with a timeout. `bb_display` wakes up all waiting threads. There is no special order of waiting threads. Data copy operation are performed in parallel by the readers after wake-up. Multiple writers must be serialized.
- *Eventmasks*: Eventmasks are specific to a single thread. A `wait` operation allows the caller to wait. When one or more events are set and the thread is waiting for one of them, the waiting threads is woken up.
- *Events*: A `wait` operation allows the caller to wait with a timeout. When an event is set, all waiting threads are woken up. There is no special order of waiting threads.
- *Futexes*: `futex_wait` allows the caller to wait with a timeout. `futex_wake` wakes up a given number of waiting threads. `futex_requeue` requeues a given number of waiting threads from one futex wait queue to a second one. Typically use cases wake or requeue either one or all threads. Waiting threads are queued in priority order. Spurious wake-ups are acceptable. Supports two modes; non-mutexes and mutexes with PIP.

Similarities and categorization: As we can see, all mechanisms have one or two implicit wait queues, which have either a priority-ordered or a FIFO-ordered queuing discipline. Notifications on a condition variable or by `futex_requeue` requeue threads from one wait queue to another. All other wake-up operations process either one or all blocked threads. Queuing operations combine a wait and a wake-up operation. Waiting on a condition variable combines a wake-up (`mutex_unlock`) with a wait operation. All other mechanisms let the caller perform a wait or a wake-up operation, but not both at the same time. When a data copy operation is involved, we need additional synchronization to serialize the message transfers with FIFO ordering.

With this, we can now assign the blocking synchronization mechanisms to four categories:

- *CAT1*: Simple blocking mechanisms comprise either a wait or a wake-up operation.
- *CAT2*: Requeuing operations move waiting threads between wait queues.
- *CAT3*: Complex operations combine both wake-up and waiting. Only `cond_wait` is in this category.

- *CAT4*: *Bounded-buffer* synchronization handles two independent blocking points.

Problem reduction: Note that by using only semaphores, or mutexes and condition variables, or futexes, we can compose the remaining synchronization mechanisms from these elementary synchronization mechanisms (see text books, e.g. [HS08], or [Dre11] for futexes). Similarly, we can also compose the bounded-buffer synchronization of *CAT4* from elementary synchronization mechanisms (see text books again), so we will not further discuss the latter for now.

The complex operation `cond_wait` of *CAT3* can also be composed from `mutex_unlock` and a waiting primitive. However, to prevent *missed wake-ups* between both elementary operations, we need to carry over state information from before unlocking the mutex. For now, we denote such an operation as a *start waiting primitive*. The resulting sequence to compose `cond_wait` from is: *start waiting primitive* → `mutex_unlock` → *waiting primitive*.

Such a sequence can now for example be implemented by using an *eventcount* [Ree76, RK79]. Using an eventcount, the start waiting primitive reads a counter value. The waiting primitive checks if the counter value is still below a given value before blocking the caller, otherwise the eventcount must have been notified by an *advance* operation, which increments the counter and wakes all already waiting threads where the counter value exceeds the waiting value. Note that eventcounts are a simple blocking mechanism of the first category.

A similar technique is possible with futexes. Here, the *start waiting primitive* simply reads the futex value, and the blocking primitive implemented in the kernel blocks the caller only if the current futex value still matches the expected value. In this regard, futexes have a different semantic (*compare-if-equal*) than eventcounts (*compare-if-below*), but do not impose counter semantics.

The technique to split a waiting operating into two parts, into a first short non-blocking operation to start a blocking operation, and a second operation that eventually blocks, allows us to *interleave* different synchronizations mechanisms.

Summary: With this, we can compose the complex operations from primitives, and only the first two categories *CAT1* and *CAT2* remain, comprising the elementary synchronization mechanisms we further focus on.

Generalization: When looking at the different synchronization mechanisms, we see lots of differences, for example semaphores implement counter semantics, while eventmasks implement bitmasks of single events, but also many similarities, such as blocking the caller. We now try to *generalize* the operations. We first discuss an *abstract waiting operation* and then an *abstract wake-up operation.*

For generalization, we encapsulate the differences between the synchronization mechanisms by defining a *semantic operation*, which hides the exact semantics of each synchronization mechanism and operates on some internal synchronization-mechanism-specific state, which we denote as *semantic state*. For example, `sem_wait` decrements the counter of a semaphore, `mutex_lock` checks the current ownership of a mutex, or `event_wait` checks the state of an event. This *pre-wait semantic operation* then either succeeds and returns to the caller, or blocks the calling thread, e.g. on a negative counter for the semaphore, contention on the mutex, or if the event is cleared. Also, when a calling thread must wait, each synchronization mechanism requires a different queuing discipline. The pre-wait semantic operation enqueues the calling thread on a specific wait queue and suspends the thread in the scheduler.

After being woken up, a thread performs another semantic operation. This *post-wait semantic operation* first checks the *cause* of the wake-up (e.g. a wake-up operation, a timeout, cancellation for signals, etc.), might have to remove the calling thread from the wait queue, and evaluates the semantic state again. Depending on the synchronization mechanism, a waiting operation now usually completes successfully or unsuccessfully. However, in case of `cond_wait`, a timeout while waiting on the condition variable might require a second round of waiting to acquire the support mutex.

We can model a wake-up operation using the same technique. First, a *wake-up semantic operation* defines what needs to be done, e.g. `sem_post` increments the semaphore counter, `mutex_unlock` releases a lock, or `event_set` sets an event. And again, this semantic operation either succeeds and returns to the caller, or a defined number of blocked threads will be woken up, e.g. one for the semaphore or the mutex, or all waiting threads in case of the event. Then the semantic operation removes the given number of threads from the wait queue and wakes up the threads.

Lastly, we can also model requeue operations to notify condition variables this way as well. A *requeue semantic operation* checks if there are threads waiting on the condition variable and if the support mutex is locked. If the support mutex is locked, the operation requeues one or all waiting threads from the condition variable wait queue to the mutex wait queue. If the support mutex is not locked, the operation wakes up one thread to become the next mutex owner, and requeues the remaining threads. Note that such a requeue operation is not very generic, but specific to condition variables and mutexes, so one could generalize requeuing as *wake a number of threads* followed by *requeue another number of threads*. However, the author is not aware of any other use case of requeuing except for condition variables.

In summary, we can generalize waiting, wake-up, and requeue in blocking synchronization (CAT1 and CAT2) by defining a synchronization-mechanism-specific semantic operation, generic operations on wait queues, and suspend and wake-up in the scheduler. For waiting, we have defined two semantic operation, i.e. one before waiting and another one after waiting, while wake-up and requeue operations require one semantic operation before waking up or requeuing one or more threads.

Identification of potential fast paths: The description of the semantic operation already shows the potential to introduce shortcuts. The first steps of a semantic operation already check if a thread must block or if there are threads to wake up or to requeue. Section 3.1.1 on related work already explained how this works for counting semaphores. Also, the futexes of Section 2.2.11 show that a generalized form of this blocking decision can be moved to user space, as the kernel is only needed for blocking and wake-up/requeuing.

3.4.2 Mapping to Low-Level Building Blocks in the Kernel

To further formalize the generalized waiting, wake-up, and requeue operations (*CAT1* and *CAT2*) described in Section 3.4.1, we now take a look at a typical implementation in an operating system kernel using fine-grained locking. We compose the operations from their underlying building blocks: look-up mechanisms to internal kernel data-objects, preemption control and internal critical sections, FIFO- and priority-ordered wait queues, timeout handling, and suspension and wake-up in the scheduler.

For this, we assume that the operating system represents each synchronization object in user space by a related internal object in the kernel. The operating system supports an *indexing system* to look-up the kernel object. The kernel object and its associated wait queues are further protected by an internal spinlock. To prevent lock holder preemption inside critical sections, preemption must be disabled before acquiring any locks. The operating system further supports self-suspension in the scheduler to either block or wait with a timeout, and a related wake-up call. As low-level building blocks are interleaved with each other, we present them in the graphical notation depicted in Figure 3.1.

In the following, we discuss the implementation of a typical blocking synchronization mechanism comprising two operations, *wait* and *wake-up of one thread*. We omit the case of waking up all threads, as this can be implemented by a loop around the core functionality. We also omit a description of a *requeue* operation. A *requeue* operation, which moves one or many threads between wait queues, comprises the same building blocks as *wait* and *wake-up* operations, but requires no interaction with the scheduler. Like in Section 3.4.1, we do not assume any specific synchronization mechanism and abstract *semantic* differences away.

Wait: A `wait` operation comprises the following lower-level steps: first, it performs the pre-wait semantic operation and evaluates the specific blocking condition, which highly depends on the actual synchronization mechanism, and if this blocking condition yields true, the operation inserts the calling thread into a wait queue

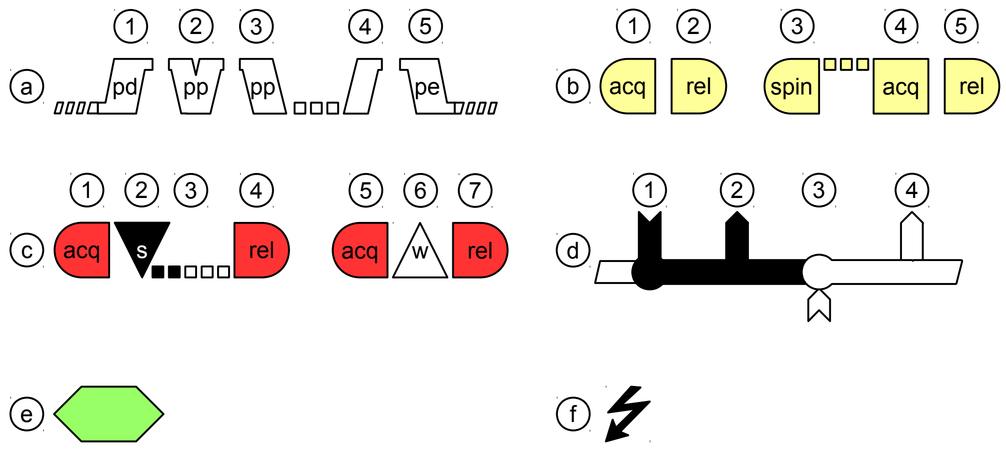

Figure 3.1: Graphical notation of building blocks.
The figure shows four sequences ⓐ to ⓓ which are read from left to right. The sequences comprise different operations or steps. denoted by ① to ⑦ for each sequence.
In the top left, ⓐ represents a sequence of preemption control operations: preemption disable (*pd*) ①, a preemption point (*pp*) without preemption ②, a preemption point with preemption ③, resumption ④ and preemption enable (*pe*) ⑤.
Next on the top right, ⓑ shows a sequence of lock and unlock operations: lock acquire without spinning (*acq*) ① and lock release (*rel*) ② operations, and a second sequence of lock with spinning (*spin*) ③ until acquiry (*acq*) ④ and lock release (*rel*) ⑤ operations.
Sequence ⓒ shows a suspend operation comprising acquiry of the scheduler lock ①, suspension (*s*) ②, eventual wake-up (change of color from black to white) ③, and finally scheduling and release of the scheduler lock ④. A wake-up operation comprises acquiry of the scheduler lock ⑤, wake-up of the target thread (*w*) ⑥, and release of the scheduler lock ⑦.
Similarly, ⓓ shows the modification to a wait indicator state variable over time: the variable is set to *waiting* state (black) ①, read ②, set to *ready* state (white) ③, and read again ④. Lastly, ⓔ shows an atomic operation, and ⓕ shows a system call.
Note that dots at the bottom indicate preemptible progress, while dots at the top indicate non-preemptible progress. All operations related to suspension are colored black, while wake-up uses white color. Critical sections are encoded by different colors: yellow for a kernel synchronization object, red for scheduling data.

---

of given order and then suspends the calling thread for a given time (timeout). After wake-up, the post-wait semantic operation checks if it was woken up by a timeout or any other unrelated condition, e.g. signal handling. In this case, the wake-up is a *spurious wake-up* and the calling thread must be removed from the wait queue, as it might be still enqueued. Finally, the operation returns with the status of the operation.

Figure 3.2 depicts a graphical representation of a successful generic `wait` sequence with all internal locking: At first, the generic `wait` operation disables preemption ① and acquires a spinlock protecting the internal kernel object and

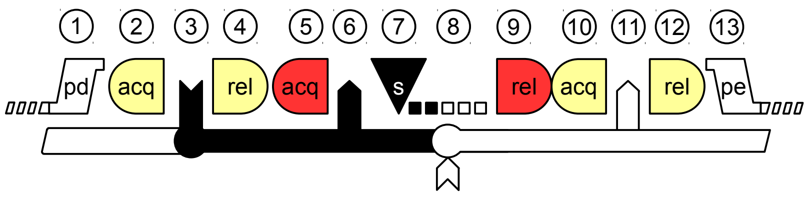

Figure 3.2: A successful generic wait operation over time.

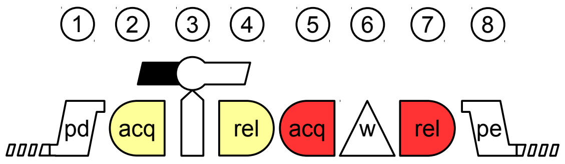

Figure 3.3: Related successful wake-up operation over time.

the wait queue ②. Inside the critical section, the pre-wait semantic operation checks the wait condition and decides to suspend the calling thread ③. The intention of the calling thread to block is indicated by the bar below the thread, which changes its color from white (ready) to black (waiting). This step is a *start waiting primitive*, as the thread can not block inside the critical section. The operation then releases the spinlock protecting the wait queue ④ and calls the scheduler to suspend the calling thread. The scheduler first acquires a spinlock protecting its scheduling data ⑤. Then the scheduler checks if waiting is still indicated (it is) ⑥, and then suspends the calling thread with a given timeout ⑦. After some time, the thread is woken up and put onto the ready queue again ⑧. It takes further time until the thread is scheduled again, and the scheduler operation completes ⑨. After suspension in the scheduler, the generic `wait` operation must acquire the spinlock protecting the wait queue again ⑩ and check if the thread is still enqueued on the wait queue or was properly woken up by another thread ⑪. Finally, the operation releases the spinlock ⑫ and enables preemption again ⑬ before returning to user space.

Wake-up: A related `wake` operation needs to perform the following lower-level steps: when a given wake-up condition is met, the `wake` operation removes the first waiting thread from the wait queue if the wait queue is not empty and then wakes up this thread.

Figure 3.3 shows this with all internal locking: at first, the operation disables preemption ① and acquires a spinlock protecting the wait queue ②. Then, the `wake` operation performs the wake-up semantic operation, decides to wake-up a thread, checks the wait queue, and removes the first thread. To wake up the thread, the operation first clears the wait indicator of this thread ③. The wait indicator of the thread is depicted above the thread. The color changes from black (waiting) to white (ready). Afterwards, the operation releases the spinlock protecting the wait queue ④ and calls the scheduler to finally wake up the waiting thread. The scheduler locks internal scheduling data ⑤, puts the thread back onto the ready queue ⑥, and releases the internal lock ⑦. The generic `wake` operation completes by enabling preemption again ⑧. The thread then returns to user space or is in immediately preempted by the woken up thread.

Preemption control: We see that both `wait` and `wake` operations are bracketed in `preempt_enable`/`preempt_disable` pairs. After releasing the first critical section (④ in both Figure 3.2 and 3.3), both `wait` and `wake` operations could also restore the previous preemption state, but then would have to immediately disable preemption again before acquiring the scheduler lock ⑤. Also, for the `wait` operation, these extra preemption points would be superfluous, as the thread anyway intends to suspend (before the scheduler's critical section) or was recently scheduled (after the scheduler's critical section). Similarly, an extra preemption point before the actual wake-up in the scheduler might introduce consistency errors (ABA problem).

Consecutive locking: We also see that the locks do not *nest*. The lock protecting the wait queue (yellow) and the lock protecting scheduling data (red) are taken *consecutively*. The *wait indicator* (line below or above the operations) is a variable that is only modified in the first critical section protecting the wait queue (change from white to black to indicate the start of waiting and change from black to white to indicate the start of a wake-up), thus protecting the *wait state* as well. In the `wait` operation, the wait indicator is then read by the scheduler before suspension. If the indicator has not been changed by a concurrent `wake` operation, the scheduler suspends the calling thread. The `wake` operation also follows this convention. It first changes the wait indicator in the critical section protected by the wait queue and wait state lock, and then calls the scheduler to put the thread onto the ready queue again. In general, the wait indicator is a *start waiting primitive*, as discussed in Section 3.4.1. For use in the kernel, a simple binary state (waiting / woken-up) is sufficient.

The benefits of taking locks consecutively instead of nesting is that the locks do not influence each other. It would also be possible to perform *hand-over-hand* locking, e.g. *lock wait queue* → *lock scheduler* → *unlock wait queue* → … → *unlock scheduler* for the `wait` operation, but this inflates the WCET of the outer critical section $e_{wait-queue}$ with the full pessimism of the inner critical section $e_{scheduler}$, i.e.

$$\begin{aligned} e_{hand-over-hand-locking} &= m \cdot (e_{wait-queue} + m \cdot e_{scheduler}) \\ &= m \cdot e_{wait-queue} + m^2 \cdot e_{scheduler} \end{aligned}$$

instead of

$$e_{consecutive-locking} = m \cdot e_{wait-queue} + m \cdot e_{scheduler}$$

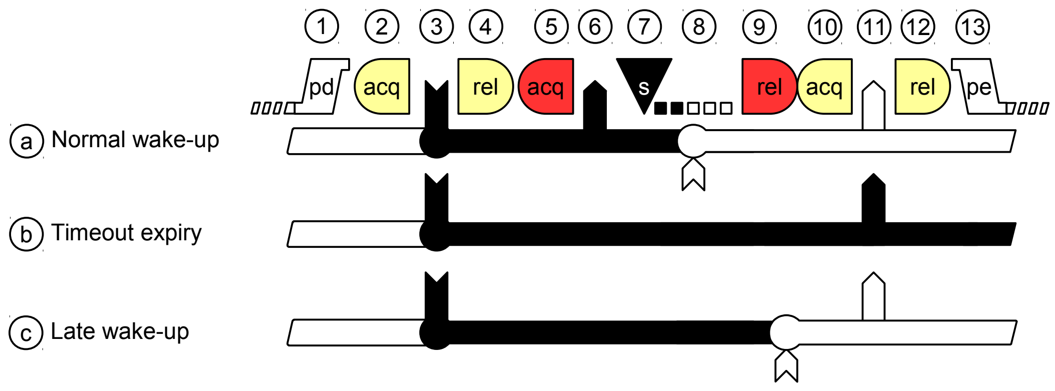

Figure 3.4: Wait operation with suspension in scheduler and ⓐ normal wake-up, ⓑ wake-up by timeout, and ⓒ late normal wake-up.

---

and therefore

$$e_{consecutive-locking} < e_{hand-over-hand-locking}.$$

This is a problem of all designs using *nested locking*.

The TOWTTOS problem: Note that after wake-up, the `wait` operation requires a second critical section (⑩ to ⑫ in Figure 3.4) for the post-wait semantic operation to handle wake-ups by timeouts or other OS-specific events. In this case, the thread *might* still be on the wait queue and the caller must remove itself from the wait queue (case ⓑ). However, note the emphasis on *might*: when a thread is woken up by a timeout at first and then a regular `wake` operation happens before the thread could acquire the lock of the wait queue again (case ⓒ), the thread was regularly woken up and already removed from the wait queue. The thread can recognize this and could change its return code to indicate a successful wake up. An appropriate reaction depends on the semantics of the synchronization mechanism.

One obvious idea to solve this problem could be to let the interrupt handler remove a thread from its wait queue. The interrupt handler would then have to lock the wait queue as well. But this does not really solve the problem. A parallel wake-up on another processor can still race between the time the interrupt handler releases the critical section protecting timeout data and acquires the critical section protecting the wait queue. Such a change would also require the programmer to temporarily disable interrupts before acquiring the wait queue lock. Also, this would have a rather huge impact on response time analysis of interrupt handling in general, without providing further benefits.

As far as we know, any system using fine-grained consecutive locking will observe this kind of *time-of-wake-up-to-time-of-scheduling (TOWTTOS) race* *conditions*[3] due to the use of multiple *independent* critical sections. The author of this thesis is not aware of any mechanism for fine-grained locking without using nested locks that prevents this kind of problems.

Summary: We have discussed that we can decompose complex user synchronization mechanisms into *elementary synchronization mechanisms*. We can further decompose these elementary synchronization mechanisms into wait and wake-up primitives where a *semantic operation* implements the semantic of a specific synchronization mechanism but the remaining steps follow the same blueprint. We have then further described the low-level building blocks in an operating system kernel. With this, we got a *generic model* of wait and wake-up primitives implemented in the kernel.

3.5 Futexes in Linux Revisited

In this section, we look at how the futexes presented in Section 2.2.11 work in detail. We extend the model and building blocks presented in Section 3.4 with a fast path.

Section 3.5.1 examines the conceptual approach of how futexes provide a fast path for blocking synchronization mechanisms, and Section 3.5.2 analyzes the futex implementation in Linux. Section 3.5.3 then discuss the shortcomings of futexes in general and the Linux implementation. Section 3.5.4 discusses safety and security aspects of futexes. Finally, Section 3.5.5 summarizes.

The analysis of the Linux implementation is based on Linux kernel versions 4.14.59-rt37 and 4.16.12-rt5 with real-time patches, Linux kernel version 5.5 without real-time patches, and glibc versions 2.24 to 2.31 for the user space part of the futex implementation.

3.5.1 Conceptual OS Implementation of Futexes

Following the discussion about the building blocks in Section 3.4, we see that the main difference between futexes and in-kernel synchronization is that futexes move the *pre-wait semantic operation* into user space and that futexes do not need a complex critical section to decide whether to block or not. Instead, the semantic operation is "compressed" into a single atomic operation in user space (depicted by the green hexagon in ① in Figure 3.5). This has certain benefits, as the futex operation in user space does not experience any lock holder preemption problems and a single atomic operation is faster than a full-blown critical section.

We will first discuss futexes using the non-PI mutex API as described in Section 2.2.11. Here, futexes split the pre-wait semantic operation into two parts:

[3]We deliberately named this problem after the *time-of-check-to-time-of-use (TOCTTOU) problem* [BD96].

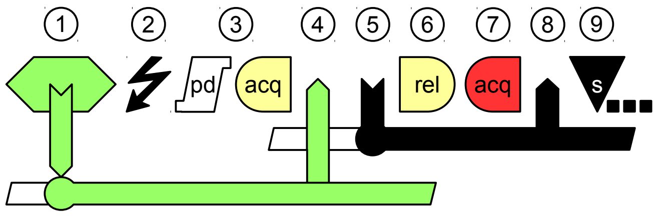

Figure 3.5: Consecutive locking and *nesting of wait indicators* at the beginning of a `futex_wait` operation until the thread is suspended in the kernel.

(i) a *first semantic operation* in user space using atomic operations on 32-bit variables, and (ii) a *second semantic operation* in the kernel comparing the futex value to prevent missed wake-ups. The first semantic operation implements the fast path for uncontended futex operation, and for this, it comprises both the modification of the semantic state and the decision whether to block by using atomic operations. At the same time, it acts as a start waiting primitive when needing to block in the kernel. Then the futex value in user space acts as a *wait indicator* for the critical section in the kernel, like in the consecutive locking scenario discussed in Section 3.4.2. For futexes implementing the PI mutex API, the semantic operation inside the kernel comprises additional steps besides checking the value, such as trying to lock the mutex for the caller.

Figure 3.5 shows this consecutive locking in an example. When waiting on a futex, a thread sets some specific value in the futex variable in user space as wait indicator (lower horizontal bar in green/white) ① and then calls the `futex_wait` operation in the kernel ②. Internally, `futex_wait` locks (and creates) a wait queue ③ and then evaluates the futex value in user space ④ before putting the thread to sleep. However, the step of putting the thread to sleep uses a second wait indicator in the kernel (upper horizontal bar) ⑤ before releasing the critical section protecting the wait queue ⑥ and suspending the calling thread in the scheduler as usual (steps ⑦ to ⑨).

The previous description has deliberately left out how in-kernel objects and wait queues are addressed. For this, we can assume two *indirection* approaches: the in-kernel object can be addressed either by an index or descriptor ID, or by the address of the futex value in user space (see also Section 2.3.1). For futexes, the original designers opted for the second approach [FRK02]. With this, futexes do not need prior registration of synchronization objects in the kernel. This property increases the flexibility of futexes and allows the kernel to have an arbitrary number of synchronization objects. However, this indirection mechanism requires additional efforts in the implementation.

We now have identified the additional building blocks of futexes: atomic operations on 32-bit variables in user space, and a split pre-wait semantic operation. We denote the synchronization-mechanism-specific encoding of the futex value in user space as *futex protocol*. The flexibility of a futex protocol in the first semantic operation is limited to 32-bit variables. The kernel side supports two different sets of semantic operations, one set of operations implementing *compare-and-block* semantics, the other set of operations implementing *PI mutex* semantics. Additionally, the kernel requires an address → in-kernel object indirection mechanism for futexes.

3.5.2 Linux Futex Implementation

We will now discuss both the user space implementation and kernel implementation of futexes in Linux. Note that the user space parts are implemented as part of the C library, while the kernel parts are implemented in the Linux kernel. In the following, we restrict the analysis of the user space implementation part to mutexes in both non-PI and PI forms as typical examples of the Linux futex API.

Data model: Besides the 32-bit futex data in user space, synchronization objects in Linux comprise additional data defining the characteristics of a specific synchronization object. For example, mutexes support a bunch of different modes and attributes (see Section 2.2.2), or barriers must encode the maximum number of waiting threads as well. Therefore, the 32-bit futex word is only a part of the synchronization-mechanism-specific data structure in user space.

In the kernel, futexes are managed by *hashed wait queues*. The Linux kernel hashes the futex address to derive the right hash bucket and an associated wait queue. Due to the hashing, this wait queue can be shared by threads waiting on different futexes. To identify the threads belonging to a specific futex, the Linux kernel stores the current futex address for each waiting thread as well. Therefore, we say "to address a wait queue" in the following, as there is no other in-kernel object associated with a specific futex.

Implementation of non-PI mutexes: Non-PI mutexes are intended for best-effort applications. The encoding of the futex value is as follows. A futex value of 0 indicates an unlocked mutex. A locked mutex is indicated by the thread ID (`TID`) of the lock owner. In Linux, thread IDs are globally (system-wide) unique identifiers of all threads in the system. Another important information in the futex value is the `WAITERS` bit. The `WAITERS` bit is the top-most (most significant) bit and indicates contention on the mutex.

`mutex_lock` evaluates the futex value in user space. The function tries to lock the mutex object for the caller from 0 to `TID` by using atomic CAS operations. When the CAS operation succeeds, the fast path completes. Otherwise the mutex must be locked. If not set, `mutex_lock` sets the `WAITERS` bit atomically. Then `mutex_lock` calls `futex_wait` to wait in the kernel. The current futex value is provided for the compare operation in the semantic operation in the kernel. Recall that `futex_wait` does not interpret the futex value, but just compares the current futex value to the given value as blocking condition. When the system call returns, `mutex_lock` restarts the sequence from the beginning again. However, this time the fast path CAS operation already sets the `WAITERS` bit, as there might be other waiting threads.

Conversely, `mutex_unlock` atomically exchanges the current futex value with 0 to release the mutex. If the previous futex value had the `WAITERS` bit set, `mutex_unlock` calls `futex_wake` to wake up one waiting thread.

Note that when a CAS-operation in user space fails, the whole sequence is restarted all over. Concurrent locking attempts from `mutex_lock` and `futex_lock` operations and false sharing is a cause of interference here.

Implementation of PI mutexes: PI mutexes are intended for real-time applications. The encoding of the futex value is the same as for non-PI mutexes. Here, the kernel fully understands the protocol encoded in the futex value.

`mutex_lock` performs the same fast path as for non-PI mutexes, but does not set the `WAITERS` bit if the fast path fails. Instead, the function directly calls `futex_lock_pi` in the kernel. In the kernel, `futex_lock_pi` evaluates the futex value again and either acquires the mutex for the caller, or sets the `WAITERS` bit and suspends the thread. In the latter case, the kernel creates a resource allocation graph and passes down the suspended thread's priority to the current lock owner.

`mutex_unlock` first evaluates the futex value. If the `WAITERS` bit is set, the function calls `futex_unlock` to handle situation and completes. Otherwise, it sets the futex value to 0 with a CAS operation. If the CAS operation fails, i.e. the `WAITERS` bit was set between reading the futex value and the CAS operation, the function also calls `futex_unlock` to handle contention. In the kernel, `futex_unlock` wakes the highest priority waiting thread, makes the thread the new lock owner and updates the futex value in user space to the new `TID`. If other threads are waiting on the futex, the kernel sets the `WAITERS` bit as well. The kernel also adjust the scheduling priority of the current thread as priority inheritance on this mutex ends.

Here, the CAS operation to set the `WAITERS` bit in the futex value moved from user space into the kernel.

Implementation of condition variables: The condition variable implementation in glibc is quite complex. In POSIX, `cond_wait` is a cancellation point, i.e. a thread can be synchronously cancelled (terminated) by other thread, while waiting in `cond_wait`. Cancellation also defines a cleanup mechanism to release any resources of the terminated thread. Due to changes in the POSIX standard w.r.t. condition variables[4], support for requeuing of waiters from the condition variable wait queue to the futex wait queue was removed in glibc version 2.25

[4]See `https://www.austingroupbugs.net/view.php?id=609`.

and not added again in later versions under the assumption that the number of threads waiting on a condition variable is usually small and thundering herd problems are rare[5]. Therefore, the condition variable implementation only uses `futex_wait` and `futex_wake`.

Wait queue hash table and look-up: We now focus on the kernel side of futexes. The futex implementation in the kernel uses a fixed-sized *hash-table* of *wait queue roots*. The kernel creates this array at boot time and allocates 256 wait queue roots per available processor. The hash table is shared between all processes in the system. To look up a wait queue for a futex, the kernel combines the futex address with internal (process- or file-system-specific) data as *hash key* and permutes the data into a unique index in the hash table in $\mathcal{O}(1)$ time. As mentioned in Section 2.2.11, Linux distinguishes between *shared* and *private* futexes. For shared futexes, the kernel uses the underlying physical address of a futex variable, while for private futexes, it can use the virtual address in user space. Wait queues are shared between futexes. Due to hash collisions, threads waiting on unrelated futexes may end up in the same wait queue.

Wait queue: The wait queue implementation is based on a *priority-sorted linked list* (plist), with 140 priority levels (100 real-time priority levels and 40 priority levels for *nice* levels from -20 to 19). The plist implementation comprises a doubly-linked list of the longest waiting threads for each used priority level, and a second doubly-linked list with all threads. Each wait queue root also contains an internal lock to ensure consistency of the wait queue (the plist). This wait-queue-specific lock is also taken when checking the expected futex value. For the internal lock, a kernel with the real-time patch uses a priority inheritance mutex. Without the real-time patch, the lock is realized as spinlock.

Waiting: For waiting operations, Linux prepares the futex key based on the futex address and optional timeout data, locks the shared wait queue in the hash table, and evaluates the futex value in user space. The kernel either compares a generic futex value or tries to lock a mutex for the calling thread and ensures that the `WAITERS` bit is set in the futex value. Then it inserts the thread into the ordered wait queue, unlocks the shared wait queue, and finally suspends the thread. Due to the priority-ordering in the plist data structure, insertion is the most expensive operation on the wait queue and takes $\mathcal{O}(p)$ time for p priority levels. After wake-up, the kernel removes the thread under the wait queue lock if the thread is still enqueued on the wait queue and performs necessary cleanups for timeouts. Removal only takes $\mathcal{O}(1)$ time.

[5]See `https://sourceware.org/bugzilla/show_bug.cgi?id=13165`.

Wake-up: For wake-up operations, the kernel iterates the wait queue in order under lock and wakes up the requested number of threads matching the target futex. Threads blocked on unrelated futexes are ignored. This takes up to $\mathcal{O}(n)$ time for n threads on the wait queue.

In the mutex case, only one thread is woken up. This thread is made the new owner of the mutex and the futex value is updated accordingly.

Requeue: For requeuing, Linux locks up to two wait queues in ascending order in the hash table. The kernel then iterates the source wait queue, and for threads matching the target futex, it wakes up a specific number of matching threads and requeues any remaining requested number of matching threads to the destination wait queue. Again, unrelated threads are ignored. If requeuing targets a different wait queue, the kernel removes threads from the source plist and adds them to the destination plist, preserving priority and FIFO ordering. Otherwise, threads remain in their current position in the plist and just get their futex key updated.

Priority inheritance protocol: The Linux kernel supports a priority inheritance protocol for `PTHREAD_PRIO_INHERIT`. For each PI mutex, the Linux kernel raises a lock holder's scheduling priority to the maximum priority of all blocked threads. Internally, the kernel maintains a *priority inheritance state (PI state)* data structure for each futex wait queue related to a PI mutex. Among other data that is relevant to the Linux kernel for life-time tracking of futexes, the PI state comprises a kernel-internal PI mutex that is the head of the queue of all threads lending their priority to the current lock holder. The PI state data is dynamically allocated on the first call to `futex_lock_pi` and freed on the last call to `futex_unlock_pi`. Therefore, this data structure is always associated with the associated futex rather than a blocked thread. The kernel supports nested PI mutexes.

Priority ceiling protocol: In POSIX, `PTHREAD_PRIO_PROTECT` is the immediate priority ceiling protocol. In Linux, this locking protocol is implemented in the C library by temporarily changing a thread's scheduling priority before locking a mutex and after releasing a mutex. Changing a thread's scheduling priority requires a separate system call.

3.5.3 Issues with Futexes in Linux

In the following, we identify and discuss issues with futexes in Linux that violate our requirements for predictability presented in Section 3.2.3. We start with particular issues in the Linux implementation and end with general issues of using futexes.

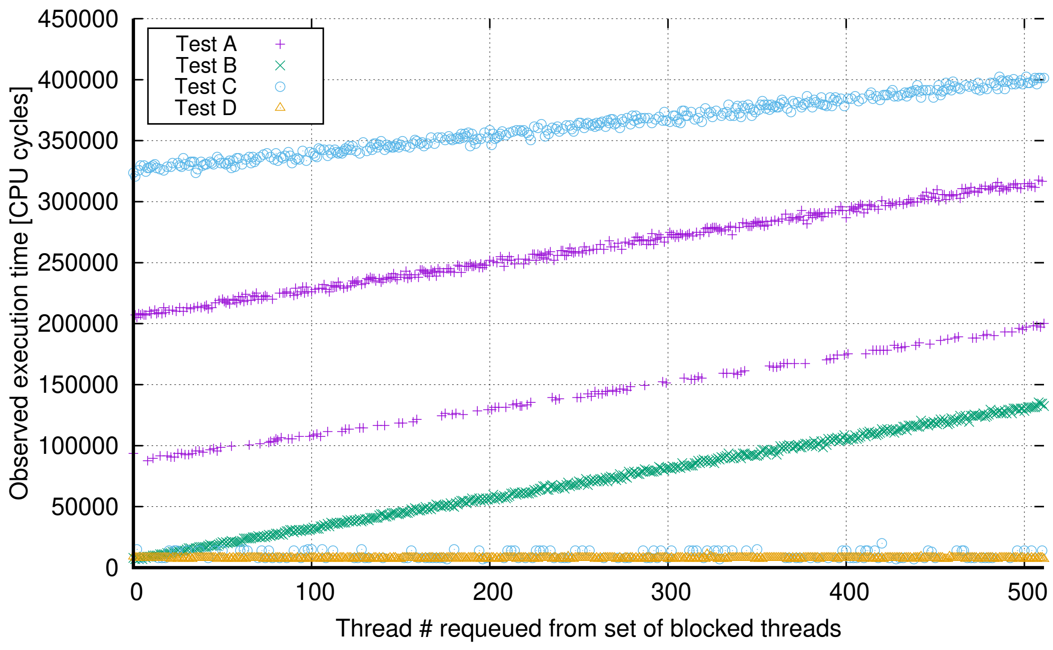

Figure 3.6: Observed execution times of `futex_requeue` operations by one process to requeue *one* of 512 blocked threads. In parallel, another process on another processor requeues *all* of its 512 threads using unrelated futexes. The four tests use different combinations of shared and non-shared hash buckets for source and destination futexes in the requeue operation: In test A, all futexes share the same hash bucket. In test B, the two processes use different hash buckets, but the source and destination futexes of their requeue operations end up in the same hash bucket. In test C, the source futex of one process uses the same hash bucket as the destination futex of the other process, and vice versa. In test D, all futexes use dedicated hash buckets. The measurements were performed on an i.MX6 *SABRE Lite* board with four Cortex-A9 ARM cores clocked at 996 MHz and Linux kernel 4.14.59-rt37. Note that outliers (due to the not always reliable synchronization in the tests) have not been removed. Diagram adapted from [ZK19].

Problem: Hash table with shared wait queues: By using shared wait queues, futexes of unrelated processes can share the same internal wait queue. Even worse, futex operations on wait queues are not preemptible. This directly violates *RQ6 (interference by shared namespaces)* and *RQ7 (preemptive operations)*, and indirectly harms predictability of futex operations. Consider an example system where trusted (real-time) and untrusted (non-real-time) applications co-exist. The untrusted applications can interfere with trusted ones by letting a large number of threads block on the same wait queue. In this case, the system integrator must assume worst-case behavior of untrusted applications.

We have demonstrated this effect in an experiment in our previous work [ZK19] by using a worst-case scenario comprising requeue operations on private (non-shared) futexes in parallel by two unrelated processes. The requeue operation is interesting here because two futex wait queues are involved, a source and a destination wait queue, and both source and destination wait queues can end up in the same hash bucket due to a hash collision. On a 996 MHz ARM processor, we observed an interference of up to 400 µs, while the average case was handled in 8 µs. We repeat the key results of the experiment in Figure 3.6. We point out that the scenario in this experiment is highly unrealistic in practice. We deliberately chose this setting in order to demonstrate potential worst-case scenarios, as `futex_requeue` locks two wait queues internally.

Problem: Resource exhaustion: A second problem is resource exhaustion in the kernel: Assume a resource-intensive (or a malicious) non-real-time process that allocates all kernel heap. In this case, even if a real-time application follows recommendations for POSIX real-time programming and pre-allocates all its resources, the allocation of a PI state data structure in a `mutex_lock` operation can fail. Note that PI state is used to track the blocked threads of a PI futex and the data structure exists as long as the according PI futex has contention, therefore it cannot be pre-allocated, as Linux does not offer an API for initialization or registration operation of futexes.

As a consequence, blocking on a mutex with contention can fail. This violates *RQ2 (robustness)*. The real-time application might be able to revert to a non-PI mutex for blocking, but it cannot apply PI to the lock holder anymore. This might lead to a priority inversion problem in the application. Even if we assume that this problem usually never happens in reality, the Linux futex API of `futex_lock_pi` specifies an error condition for memory allocation failures. Thus user applications must provide additional error handling code for this. Note that for testing reasons, Linux supports a fault injection interface to let arbitrary memory allocations of PI state data fail.

Problem: Unbounded atomic operations: The different futex protocols use atomic operations on the futex value. When a CAS operation fails, the implementation repeats the atomic sequence again. This can happen due to legitimate updates of the futex value or false sharing. But this make it hard to obtain an upper bound on the number of retries of the CAS operations for *RQ5 (bounded loops and retries)*.

Note that this is usually not a problem for well-behaving applications. In this case, the corner cases of the specific futex protocol must be considered to derive an upper bound for a correctly behaving operation. However, recall that `futex_lock` performs atomic operations on the futex value from the kernel to lock the mutex for the calling thread or set the `WAITERS` bit. If these CAS operations fail, the kernel simply starts over and evaluates the futex value again. Note that this provides an attack surface for a malicious application. By writing to the futex value or even accessing the same cache line in parallel, a malicious application can prevent progress in the kernel.

This problem can be mitigated by bounding the number of retries of the atomic operations in the kernel. The kernel could then return to user space with an error. In turn, the user space code could retry the operation again. This would at least remove the attack surface from the kernel.

Problem: Arbitrary priority boosting: In `futex_lock_pi` operations, the kernel cannot check whether or not the target thread currently holds a PI mutex in user space. A malicious thread can now use such an operation to apply a priority boost using the kernel's priority inheritance mechanism to *arbitrary* threads of *unrelated* processes in the system. This not only violates the assumptions of a strict process model where threads cannot alter the scheduling of threads in other processes: arbitrarily boosting a real-time thread's priority above others may effectively invalidate a previous scheduling analysis and may lead to priority inversion problems, and boosting of non-real-time threads may lead to starvation of lower priority real-time applications. Effectively, this violates *RQ2 (robustness)* and *RQ6 (interference by shared namespaces)*.

However, this problem is not specific to `futex_lock_pi`, but to Linux processing in general, as the Linux process model only provides weak separation between processes. As real-time scheduling often requires *root* permissions, a malicious thread could also invoke `sched_setscheduler()` to change the target thread's scheduling attributes directly. The impact of this arbitrary boosting can be mitigated by defining appropriate *control groups* with separate namespaces for processes and thread IDs, but this imposes an additional burden on the system integrator and on the validation of the configuration.

General issues: The following discussion is not specific to the futex implementation in Linux, but to futexes in general. They show *structural issues* with the futex approach.

The concept of using atomic operations for the semantic operations shows a limitation. Futexes protocols must always be "compressible" to a single 32-bit variable. The restriction to a single atomic word and the limited word size can become a serious barrier for adoption of more complex synchronization mechanisms. Note that using 64-bit atomic operations would be better here. However, when futexes were conceived, most of the available computers running Linux were 32-bit architectures without 64-bit atomic operations, but had at least atomic *fetch-and-add* or similar primitives [FRK02]. Therefore, futexes in Linux are limited to 32-bit variables.

The simplicity of futexes being a *compare-and-block* primitive also poses problems. Consider the case where the compare operation in the kernel fails. Then the `futex_wait` operation returns immediately, and the user space code has to handle this situation. This behavior is acceptable for a condition variable implementation where the futex word is changed on each wake-up operation, but it still causes spurious wake-ups, e.g. when two threads start call `cond_wait` while another thread intends to wake up only one thread with `cond_signal`. For other use cases where spurious wake-ups are not acceptable, the user space implementation must hide failed waiting attempts, e.g. by repeating the call to `futex_wait` again. And this could lead to unbounded loops.

A more general *compare* operations in the kernel could address parts of these problems. Unnecessary looping could be prevented if the kernel has more semantic information than the simple *compare-if-equal* comparison when evaluating the futex value. However, this would require a more complex API. Also, parameterized arithmetic operations can only test for limited patterns. In parts, the Linux API supports this with the `bitset` variants.

Even more flexibility could be gained if the kernel operation could actually *modify* the futex value. Then futexes would become a *compare-and-modify-and-block* mechanism. The Linux API supports this partly on the wake-up side with the `futex_wake_op` operation, see Section 2.2.11.

3.5.4 Safety and Security Aspects of Futexes

We briefly discuss safety and security aspects of futexes. In Section 3.5.3, we already identified temporal interference problems due to shared futexes and the problem of boosting arbitrary threads. Both problems are specific to the implementation in Linux. Still, we must discuss the side effects of intended or unintended changes to the semantic state exposed to user space, as this affects futexes in general.

For threads in the same process or same partition, we assume that these threads must trust each other anyway. A thread locking a mutex and then not releasing the mutex is considered an application error, regardless of the implementation of the mutex. Similarly, we can consider that all unintended modification of the futex value are application errors. As the futex value in user space is kept in memory of the partition, such errors are properly isolated to the affected partition.

For futexes shared between partitions, recall from Section 3.2.1 that the system design is restrictive by default and that access to a shared memory containing a futex value must be explicitly granted. Then we can compare the situation to the use of shared mutexes or shared semaphores. When sharing a blocking synchronization mechanism, the participants must trust each other and expect that the other side plays fair and does not mess up the synchronization. Therefore, any modifications to the futex value can be considered to be an application error again. The fault remains isolated in the affected partitions and cannot harm the kernel or impact unrelated partitions. Also, applications can still detect these errors, e.g. by blocking with a timeout.

3.5.5 Summary

The futex design in Linux strongly focuses on performance, as performance is critical in real-world server and desktop applications. The implementation was carefully tuned to avoid internal overheads where possible. The design is based on the assumption that contention is rare when using mutexes, and, if contention happens, only a small number of threads are blocked at all. This explains the design decision to use a hash table and linked lists, as the necessary operations take $\mathcal{O}(1)$ *amortized* time. Also, Linux developers are aware of the side effects of this design, as [BN14, Bro16] and the fault injection interface for PI mutexes show. In summary, futexes as implemented in Linux are efficient, but not predictable.

3.6 Alternative Approaches for Efficient Synchronization Mechanisms

We now discuss *alternative approaches* to futexes for efficient blocking synchronization mechanisms in user space. From the analysis of the futex building blocks in Section 3.5.1, we can see that the main technique for efficiency is to implement a fast path in user space to skip unnecessary overhead for system calls. We start our discussion by addressing the specific limitations of futexes, and derive a more generic model that uses *full* critical sections instead of just atomic operations.

The discussed techniques and Table 3.1 were presented in previous work of the author [Zue20].

Wider atomic operations: Recall that futexes implement the fast path by an additional semantic operation in user space, and that futexes "compress" the semantic state into a 32-bit value due to the dependency on 32-bit atomic operations. A first (and probably the most natural) approach to extend futexes is to consider to lift the limitation of having only 32-bit semantic state. For this, we could consider to use wider atomic operations. However, while atomic operations on 32-bit, 64-bit, or even 128-bit variables are often available on today's processor architectures, larger state or disjoint memory locations cannot be handled by simple atomic operations. Historical processor architectures, such as Motorola 68000 and Intel i860, allowed to modify two (Motorola 68000) or more words (Intel i860) atomically by holding the bus locked for a longer time [GH91], but such techniques have fallen out of favor for today's systems due to the high impact on scalability when using more than a few processor cores. Today, techniques like transactional memory are favored [HM93]. Transactional memory requires special hardware, but is not yet widely available on contemporary processors.

Spinlock-protected critical sections: Alternatively, a spin-based critical section for mutual exclusion might be helpful. Spinlocks can be implemented efficiently in user space. However, using spinlocks in user space introduces non-determinism due to lock-holder preemption problems, see Section 2.1.3. Therefore, we need to provide a mechanism for preemption control as well. With this, we get *non-preemptive spin-based critical section*, similar to the ones inside an operating system kernel, see Section 2.3.3. Note we leave open the question how preemption control can be implemented efficiently.

When using spin-based critical sections, the described futex system calls are still sufficient, but not optimal for waiting and wake-up in the operating system kernel. They do not cope well with the critical section, as the system call for blocking must be moved *outside* the critical section. In this case, the critical section prepares a 32-bit futex value to encode sufficient information to prevent missed wake-ups and other race conditions, and the actual call to `futex_wait` happens after the critical section is unlocked, like when using eventcounts [RK79]. Note that this problem does not apply to the non-blocking `futex_wake`, which can be called from within the critical section. But a wake-up operation *inside* the critical section can lead to preemption (kernel interaction) right *after* the critical section is released and preemption is enabled again.

Also, when using a spinlock-protected critical section to protect any state information in user state, the 32-bit futex value does no longer need to be the only value to block on. Instead, multiple futex values could be used, e.g. in the context of reader-writer locks, one futex value could address blocked writers, while a second futex value could address blocked readers. One could even think of making the futex value private to a blocking thread.

These considerations show that spinlock-protected critical sections in user space are promising, but futexes might not be the best mechanisms to support them.

Non-preemptive spin-based monitors: An even more radical approach is to move *more* steps from the kernel to user space. From the analysis of the building blocks in Section 3.4.2, we can see that the in-kernel implementation comprises the look-up of a kernel object, disabling preemption, internal locking, a semantic operation, and a wait queue operation before finally suspending or waking up threads in the context of the scheduler. Moving the scheduler operations into user space is unrealistic, as the kernel must be able to suspend, preempt, or wake up threads without interference from user space. But we can at least move *all other steps* up to the scheduler interaction into user space.

This effectively extends the previously discussed approach (spinlock-protected critical sections) with the handling of the wait queue and turns the approach into something which resembles a *non-preemptive spin-based monitor*. At the same time, this also enables a different approach to address the blocked threads in wake-up operations, as the wait queue is kept in user space. Threads can be addressed *directly* by their thread ID instead of indirectly via an in-kernel wait queue.

Comparison: Table 3.1 shows and compares the building blocks of three approaches for a generic blocking synchronization mechanism implemented as (i) a *baseline* approach using system calls, (ii) a *futex* approach like in Linux, (iii) the previously discussed *monitor* approach that handles the wait queue in user space. Comparable operations and data objects are placed in the same rows.

System call: The baseline implementation in the left column uses system calls to implement a blocking synchronization mechanism and shows the building blocks as discussed in Section 3.4.2. Here, user space code calls the kernel with an identifier to a kernel object. In turn, the kernel validates the identifier and retrieves the kernel object comprising all necessary data in the look-up step. Then the kernel disables preemption, locks the data in the kernel object, and performs a semantic operation, such as checking and modifying the internal state. At this point, the semantic operation decides whether the operation is completed or if a wait queue operation is needed. In the latter case, the kernel either enqueues the calling thread on the wait queue, or removes a waiting thread from the wait queue, depending on the desired operation. Then the kernel must lock internal

Table 3.1: Comparison of three implementations of a generic blocking synchronization mechanism. The upper part shows the layered operations from user space down to the kernel. A "•" marks the operations in the fast path when no blocking is needed, i.e. in the fast path, only the marked operations are performed. The lower part shows the associated data in both user space and the kernel. A "◇" denotes global data.

operations	**system call (baseline)**	**futex**	**monitor**
user space			• disable preemption
			• lock user space object
		• 1st semantic op. (atomic)	• 1st semantic operation
			wait queue operation
	• system call	system call	system call
kernel	• look-up kernel object	look-up futex wait queue	look-up thread
	• disable preemption	disable preemption	disable preemption
	• lock kernel object	lock wait queue	
	• semantic operation	2nd semantic operation	
	wait queue operation	wait queue operation	
	lock ready queue	lock ready queue	lock ready queue
			2nd semantic operation
	suspend / wake-up	suspend / wake-up	suspend / wake-up

data model	**system call (baseline)**	**futex**	**monitor**
user space	ID of kernel object	futex value (atomic)	user space object lock
		additional semantic state	semantic state
			wait queue
			◇ thread states
kernel	kernel object lock	wait queue lock	
	semantic state	address	
	wait queue	wait queue	
	◇ ready queue lock	◇ ready queue lock	◇ ready queue lock
	◇ thread states	◇ thread states	◇ thread states

scheduling data (ready queue lock) before it can finally suspend the calling thread or wake up a waiting thread.

Futex: A futex-based implementation with its building blocks as discussed in Section 3.5.1 is shown in the middle column. Compared to the system call approach, the main difference of the futex-based implementation is the 32-bit state variable in user space expressing the semantic state. Depending on the atomic operation on the variable, the semantic operation either succeeds immediately or requires a system call. In the kernel, the look-up of an associated in-kernel wait queue is based on the user space address of the atomic variable, but the following steps are similar to the baseline approach. The second semantic operation in the kernel is a check if the futex value has changed in the mean time or locks / unlocks a mutex on behalf of the caller.

Monitor: The right column shows the building blocks of a monitor using a non-preemptible spin-based critical section. The monitor-based implementation differs from both previously discussed approaches. From the data point of view, the user space object comprises a lock, semantic state, and a wait queue. From the execution point of view, the user space code disables preemption and locks the object before it evaluates the internal semantic state in the semantic operation. In the fast path, the semantic operation succeeds and the operation completes. In the slow path, a system call to the kernel is required to block or to wake up. For blocking or wake-up, the wait queue operation either adds the current thread to the wait queue or removes a thread from the wait queue and then in turn calls into the kernel. The kernel first validates the given thread ID and locates the target thread. Then the kernel disables preemption, locks the necessary scheduling data, and performs a second semantic operation. The second semantic operation is serialized by this last lock and detects parallel wake-up or suspend operations. If the semantic check succeeds, a suspend or wake-up operation takes place.

Discussion: When comparing the three approaches, we can see that the baseline approach and futexes have a similar overall structure, while the monitor approach moves the steps concerning wait queue management from the kernel to user space.

When comparing futexes and the monitor approach, the monitor approach allows more complex expressions to appear in the semantic operation, since the semantic operation is protected by a critical section. However, for comparable performance between futexes and monitor in the fast path, this critical section must be kept short.

From Table 3.1, we can draw the following conclusions.

First, a non-preemptive spin-based monitor in user space is a blank spot on the map in the design space of efficient blocking synchronization mechanisms. Especially, the building block to temporarily disable preemption is critically important for a well-performing implementation, as these operations should not use system calls. From the predictability point of view, we can use much a simpler mechanism to suspend and wake up threads in the kernel. This clearly simplifies a timing analysis of the operating system kernel. But at the same time, the overall complexity is not reduced, but just moved to user space.

Second, both the baseline approach and the futexes show a quite similar structure in the kernel. A *different futex design* that does not share wait queues could improve on of the Linux implementation and solve the predictability issues shown in Section 3.5.3.

3.7 Preemption Control Mechanisms

In this section, we analyze and compare different mechanisms for preemption control. We use these mechanisms to temporarily disable preemption while executing in a critical section to prevent lock holder preemption problems as discussed in Section 2.1.3. Due to their conceptual relation to real-time locking protocols, we use the term *protocol* for these mechanisms as well.

We start with an in-depth review on related work of different approaches to mitigate the lock holder problem in Section 3.7.1. Then we look at the implementation level of preemption control mechanisms inside operating system kernels in Section 3.7.2. This provides a good starting point to identify particular building blocks. We discuss these building blocks in Section 3.7.3. Section 3.7.4 discusses the details of the temporary non-preemption protocol in Symunix II, and Section 3.7.5 analyzes the fast IPCP implementation by Almatary et al.. Finally, Section 3.7.6 summarizes and discusses the results.

Our goal is firstly to design an efficient mechanism to effectively disable preemption for the current thread in user space, when possible without using expensive system calls. Secondly, the preemption mechanism should work well with the monitor approach discussed in Section 3.6.

3.7.1 Related Work on Lock Holder Preemption

In the following, we review related work on the lock holder preemption problem. As already stated in Section 2.1.3, measures comprise preventing or avoiding preemption in the first place, recovery from preemption, and mitigation of side effects of preemption. The first two categories describe *preemption-safe lock mechanisms* [MS98]. The last category addresses the problem at the level of scheduling, i.e. by scheduling threads that share locks at the same time in parallel, e.g. coscheduling [Ous82]. As we mainly consider P-FP scheduling, we cannot make assertions about the scheduling on other processors, and therefore we focus on the first two categories in the following discussion.

Mechanisms of the first category can *prevent* any preemption in the first place. Edler et al.'s *temporary non-preemption mechanism* in *Symunix II* disables preemption before entering a critical section and enables preemption again after leaving the critical section [ELS88]. This is an anticipatory mechanism, similar to ceiling-based real-time protocols, especially IPCP. From the usage point of view, this mechanism is easy to use and does not require any further information about the critical section or the number of involved CPUs. The mechanism is also robust as long as a thread follows the protocol and does not continue execution with preemption disabled. Lastly, the mechanism is optimistic in that preemption is delayed by at most the WCET of the critical section.

In contrast, the *two-minute warning* mechanism by Marsh et al. in *Psyche* [MSLM91] *avoids* preemption if the remaining time of the thread's current time slice is not sufficient to complete the critical section. Here, the user must at least have the number of concurrent threads and the information how long it will take to complete the critical section to use this mechanism correctly. This makes the protocol both hard to use and pessimistic: if the time budget for the critical section is too small, the thread observes lock holder preemption, if the time budget is too large, the thread unnecessarily yields.

Note that both mechanisms were originally designed for systems with quantum scheduling. They allow to shorten or extend the next quantum accordingly to increase fairness. For Marsh et al.'s approach to work at all, we need the exact time of the next preemption. However, this is impossible to achieve in an event-based system. Only Edler et al.'s approach works in other scheduling systems.

Mechanisms of the second category provide means to recover from the fact that a lock holder was preempted. We can further categorize the mechanisms as follows: (i) *passive* ones that prevent wasting of too many CPU cycles on the waiter side by changing from spinning to blocking; and (ii) *active* ones that try to solve the problem, either by pushing the preempted thread out of its critical section by yielding to the lock holder, or by helping the preempted thread otherwise to finish the critical section. Compared to the mechanisms for prevention and avoidance, we can see that these mechanisms here require an action on the side of the spinning threads, but not on the side of the preempted threads. Note that the passive mechanisms simply cope with the fact that the lock holder is preempted. These mechanisms were primarily designed for best-effort systems. However, the active mechanisms differ. We can categorize them as *reactive* mechanisms, similar to the real-time protocols in Section 3.3. Also note the similarity of yielding CPU time to a preempted lock holder here to non-recursive PIP.

We do not further consider both approaches of the second category. The passive approaches just prevents further busy-waiting (best-effort), and the active approaches inflate the WCET of critical exceptions too much. Explained in detail, this means that yielding to a preempted lock holder quickly degrades to a scenario similar to a futex-based PI-mutex where a higher priority waiting thread donates its current priority to the preempted lock holder. And if now more than one thread is blocked, the other threads will continue spinning.

With this, the only practical remaining approach is to temporarily disable preemption while busy-waiting for a lock and during a critical section. However, a mechanism to disable preemption also requires a safeguard mechanism if applications abuse this. This might not be the case for real-time applications except in error cases, as they are usually well designed, but we also want to support best-effort applications. But there is a discrepancy to consider: when the safeguard mechanism triggers and preempts a thread, the *correctness* of a best-effort application is usually not affected, as other threads simply continue spinning for a longer time and the situation clears up when the preempted thread is scheduled again. However, for a time-critical application, the preemption might be problematic.

3.7.2 Preemption Control inside OS Kernels

In operating systems for single processor systems, we see the pattern to disable interrupts to achieve atomicity for actually non-atomic operations. For this, a single bit in a CPU register to encode the `enabled` or `disabled` state of a CPU to receive interrupts is sufficient. However, some hardware platforms provided multiple levels of interrupts and –naturally– operating system developers started to use these. For example, the *set privilege level (spl)* instruction of the *PDP-11* architecture with 8 interrupt levels influenced the mechanism for interrupt handling in early and later Unix systems [MNNW14]. A similar mechanism is the *interrupt request level (IRQL)* in the Windows NT kernel family [RSI12].

Sometimes, the instructions to control the interrupt level were costly, so software workarounds were used instead and protocols to temporarily disable preemption emerged, again supporting multiple interrupt levels. Operating system implementers also optimized these mechanisms a lot. The protocols to disable preemption were typically placed below interrupt privilege levels in the priority space, and above user-level priorities. This shows the semantic relationship of both mechanisms, and we can map thread priorities and interrupt priorities into a single unified priority space. We can even push this relation further with *threaded interrupts* in *Solaris* and map all interrupts to threads [KE95], or the other way around and use the interrupt controller as scheduler as in *Sloth* [HLSS09].

The preemption control mechanisms in Linux[6] works as follows. Each thread has two preemption related variables, a preemption counter `preempt_counter` and a flag variable `need_resched` that indicates a pending preemption request.

Listing 3.1: Preemption control mechanisms in the Linux kernel

```
void preempt_disable()
{
    preempt_count++;
}

```

[6]The Linux kernel supports different preemption modes, depending on compile-time configuration. Here, we consider a preemptible kernel.

```
 6  void preempt_enable()
 7  {
 8      preempt_count--;
 9      if ((preempt_count == 0) && need_resched) {
10          cond_resched();
11      }
12  }
```

The preemption counter is incremented each time when preemption is disabled (line 3). Conversely, enabling preemption decrements the counter (line 8). The counter becomes zero again when the last (outer) nested preemption request completes. And if `need_resched` is set as well, the scheduler is invoked (line 10).

Linux supports nesting of disabled preemption requests. This helps in the context of library functions that use nested spinlocks. The Linux implementation does not support different priority levels. Instead, Linux supports different *classes* of services, so-called *softirqs*, which run in the context of interrupt handlers, e.g. timer handling and network packet processing. Processing of these is controlled by changing different parts of the counter, e.g. incrementing and decrementing by $0x100$.

Note that the protocol exposes a short race condition: when a thread is preempted *after* decrementing the preemption counter and testing the counter for zero and checking the `need_resched` flag (line 9), but *before* calling `cond_resched` (line 10), the thread still calls `cond_resched` after preemption.

Also note that inside an operating system, unlike to user space, there are usually no safeguards necessary. Code executing in the operating system is trusted and (usually) designed and optimized for latency, therefore critical sections are short. Still, the Linux kernel provides a watchdog-based safeguard to print a warning ("soft lockup detected") when a thread executes non-preemptively in the kernel for too long time and eventually reboot the system after a configurable time.

3.7.3 Building Blocks of Preemption Control Mechanisms

Section 3.7.2 presented how preemption control mechanisms are usually implemented inside an operating system kernel. The in-kernel mechanisms are protocols for use in kernel space, which is trusted code and does not require any safeguards.

We now define the following terminology. A protocol to disable preemption needs to have at least two *preemption levels*. We define that the lowest preemption level means *preemption enabled* or *base priority*, and the highest preemption level means *preemption disabled* or *elevated priority*. With this, we can map single bit protocols into the priority world.

The current thread is said to be *interrupted* when an asynchronous interrupt, e.g. a timer interrupt or a device interrupt, happens during a critical section.

We denote a *scheduling event* when the interrupting ISR or the current thread in its critical section wakes up another thread. Note that a scheduling event may wake up a thread with (i) a higher priority than the current thread, (ii) a priority between the base priority and the elevated priority, or (iii) a lower priority than the current thread. In case (i), the current thread must be preempted immediately. In case (ii), the current thread is expected to be preempted *after* the critical section. We also say that *preemption is pending.* In case (iii), the current thread is not expected to be preempted. When more than one thread is woken up, the scheduling priority of the highest priority thread must be considered.

From this, we can derive the building blocks of preemption control mechanisms for user space. A protocol starts at the low base level (preemption enabled), and *temporarily increases* the preemption level to a higher level to disable preemption. And the end of the critical section, the protocol *reverts* to the previous state. And when preemption is pending, the protocol calls the scheduler.

The protocols may use a single bit or multiple priority levels to control preemption. The protocols should support nesting of requests. Single bit preemption-disable protocols can be easily extended to support nested requests by adding a counter. To support nesting in multi-level protocols, the previous priority state or level can be kept in a local variable on the caller's stack.

Finally, a protocol should preferably not require system calls to change the preemption level. The protocol should optimize for short critical sections. This increases the likelihood that the calling code will effectively not be preempted anyway.

3.7.4 Temporary Non-Preemption in Symunix II

We will now analyze the *Symunix II* protocol [ELS88]. Edler et al. implemented a preemption-disable mechanism with two levels for user space with a forced preemption watchdog as safeguard mechanism. The protocol was originally designed for a system with quantum scheduling (best-effort), so the protocol does not support different scheduling priority levels.

The protocol uses two variables shared between user space and kernel. Both variables are initially set to zero. With the first variable `not_preempt_now` ≥ 1, user space tells the kernel its wishes not to be preempted. The kernel uses the second variable `preempt_pending` $\neq 0$ to notify user space about a pending preemption request. Only in this case, the user space code needs a system call (`sys_yield`) to finally preempt the current thread.

The implementation of the protocol is straight forward, as Listing 3.2 shows:

Listing 3.2: Preemption control mechanisms in Symunix II

```
void preempt_disable()
{
    not_preempt_now++;
```

```
 4 }
 5
 6 uint32_t preempt_enable()
 7 {
 8     not_preempt_now--;
 9     uint32_t pending = preempt_pending;
10     if ((not_preempt_now == 0) && (pending != 0)) {
11         preempt_pending = 0;
12         sys_yield();
13         return pending;
14     }
15     return 0;
16 }
```

In the timer ISR (Listing 3.3), which handles the time slice scheduling, the kernel sets `preempt_pending` to either 1 (pending, line 8) or 2 (overdue, line 13), allowing user space to detect the situation:

Listing 3.3: Timer implementation in the Symunix II kernel

```
 1 kernel_timer_isr()
 2 {
 3     ...
 4     if (not_preempt_now == 0) {
 5         // OK to preempt now
 6     } else if (preempt_pending == 0) {
 7         // preempt later
 8         preempt_pending = 1;
 9         // do not preempt
10     } else /* preemption already pending */ {
11         // safeguard time elapsed
12         // indicate forced preemption
13         preempt_pending = 2;
14         // OK to preempt now
15     }
16     ...
17 }
```

The protocol follows the same principles as the in-kernel protocol of Linux discussed in Section 3.7.2. The first protocol variable `not_preempt_now` is a preemption counter, and the second protocol variable `preempt_pending` indicates a pending preemption request. The protocol shows the same small race condition in lines 10 to 12 in Listing 3.2. Also, the protocol supports nesting, but is not priority-based. However, we can easily adapt the protocol to priority scheduling if we set the `preempt_pending` accordingly on scheduling events that wake up higher priority threads, and replace `sys_yield` with `sys_preempt`.

The safeguard mechanism is also shown here. When a thread remains in preemption disabled state for more than a full time slice, the second timer interrupt will enforce preemption.

A benefit of this protocol is that it requires no system call on average as long as the thread is not interrupted by a timer interrupt. In the unlikely latter case, the protocol requires exactly one system call.

3.7.5 Fast IPCP Implementation by Almatary et al.

Almatary et al. presented a protocol to implement IPCP (the immediate priority ceiling protocol) and DFP (the deadline floor protocol) in user space [AAB15]. The variant with IPCP is for FP scheduling and supports nested requests, while the variant using DFP is for EDF scheduling and does not. In the following, we will discuss only the IPCP variant. We first analyze the non-nested case and then the nested case.

The protocol uses three variables shared between user space and kernel. The first variable `new_pri` refers to the elevated scheduling priority, the second variable `to_raise` indicates that an elevated priority is active, and the last variable `start` protects the priority restore operation. Additionally, the thread must know its base priority `base_pri`. Initially, `new_pri` is set to `base_pri`, and both `to_raise` and `start` are false.

In an acquire operation in the non-nested case, user space code sets both `new_pri` and `to_raise`, and during the release operation, it additionally sets `start` to indicate the start of a priority restore operation:

Listing 3.4: Fast IPCP by Almatary et al., non-nested case

```
void acquire_non_nested(prio_t prio)
{
    new_pri = uprio;
    to_raise = true;
}

void release_non_nested(void)
{
    start = true;
    if (to_raise) {
        to_raise = false;
    } else {
        sys_set_prio(base_pri);
    }
    start = false;
}
```

On a scheduling event, the kernel synchronizes `new_pri` with the effective in-kernel priority of the thread if `to_raise` is set and `start` is not, as Listing 3.5 shows:

Listing 3.5: Kernel part to handle the non-nested case

```
kernel_sched_event(<...> new_thr)
{
    ...
    // update priority if necessary
    if (to_raise && !start) {
        to_raise = false;
        kernel_set_prio(new_pri);
    }
    ...
    if (current_thr->prio < new_thr->prio) {
        // OK to preempt now
    }
    ...
}
```

Here, `current_thr` refers to the current thread and `new_thr` refers to another thread becoming ready. If `start` is set, the kernel knows that the current thread is in a release operation and already has left its critical section. In this case, no further actions are needed.

By using a dedicated variable to indicate the current execution of a release operation, the protocol effectively solves the race condition visible in the Linux kernel (Section 3.7.2) and in Symunix II (Section 3.7.4). For the non-nested case, the protocol requires no system calls on average and only one system call in case of a scheduling event during the critical section.

Also, the protocol assumes correct behavior of threads, so a safeguard mechanism is not provided. However, unlike the Linux and Symunix II protocols that disable preemption, this protocol only increases the scheduling priority of a thread. If one considers a configurable process-specific upper bound of the scheduling priorities of all threads in a process, the effect of an ill-behaving thread would be limited to its process and can be mitigated.

For the nested case, the protocol introduces another shared variable `level` to track the nesting level. In the following implementation, we use the same interface as the previous two protocols for clarity:

Listing 3.6: Fast IPCP by Almatary et al., nested case

```
prio_t prio_raise(prio_t prio)
{
    prio_t prev = new_pri;
    new_pri = prio;
    level++;
    if ((level > 1) && !to_raise) {
        sys_set_prio(new_pri);
    } else {
        to_raise = false;
    }
    return prev;
}

void prio_restore(prio_t prev)
{
    level--;
    start = true;
    new_pri = prev;
    if (to_raise) {
        if (level == 0) {
            to_raise = false;
        }
    } else {
        sys_set_prio(base_pri);
    }
    start = false;
}
```

And the kernel part in Listing 3.7 changes accordingly:

Listing 3.7: Kernel part to handle the nested case

```
 1 kernel_sched_event(<...> new_thr)
 2 {
 3     ...
 4     // update priority if necessary
 5     if (to_raise && (!start || (level > 0))) {
 6         to_raise = false;
 7         kernel_set_prio(new_pri);
 8     }
 9     ...
10     if (current_thr->prio < new_thr->prio) {
11         // OK to preempt now
12     }
13     ...
14 }
```

The kernel now updates the in-kernel priority to the priority in user space on all scheduling events (line 5). This leads to the problem that user space code needs a system call in each `prio_restore` operation in the worst case when the scheduling event happened at the deepest nesting level. But if no scheduling event happens during the nested critical sections, no system calls are necessary.

Lastly, the nested protocol is not easy to understand. The authors use model checking to verify the correctness of their approach.

3.7.6 Discussion

Both the Symunix II protocol discussed in Section 3.7.4 and the fast IPCP protocol discussed in Section 3.7.5 can be used to implement short non-preemptive critical sections in user space on a single processor system. For IPCP, one must simply use a priority set to the maximum of the scheduling priorities of all involved threads. Both protocols optimize for the case that critical sections are short and that the temporary changes to the preemption state or the scheduling priority do not last long, and the code will effectively not be preempted in most situations anyway.

Compared to a baseline approach that would use two system calls to implement `preempt_disable` and `preempt_enable`, both protocols get rid of system calls in the average case and require at most one system call in the non-nested case. The system call is only needed at the end of a critical section, when a previous preemption state or priority is restored, to finally preempt the thread. In the nested case, the Symunix II protocol requires at most one system call for all nested critical sections, while the fast IPCP protocol requires one system for each critical section.

Also, the fast IPCP protocol synchronizes the priority in the kernel *unconditionally* on a scheduling event. However, this will be problematic for a wake-up operation in the monitor approach discussed in Section 3.6. Then we would always need a system call when enabling preemption again, even if the thread woken up has a lower priority than the base priority of the caller. This is less a problem for the Symunix II protocol, as it always operates on the base priority.

With this, we can summarize that both protocols support uninterrupted critical sections without the need for system calls. Also, both protocols support nesting. For a real-time system, the fast IPCP protocol by Almatary et al. would be a good candidate, as it integrates into priority scheduling. However, if we consider the nested case, the Symunix II protocol would be better, as it requires only one system call in the worst case.

And we can conclude that there is still room for improvement. A better protocol for preemption control should integrate nicely into priority scheduling and should require a system call only when kernel interaction is really needed, i.e. preemption is pending.

3.8 Low-Level Wait and Wake-up Mechanisms

In this section, we analyze and compare low-level mechanisms to suspend and wake up threads. From the discussion about the monitor approach in Section 3.6, we discussed to move the concept of a wait queue into user space and we concluded that a simpler scheme to suspend and wake up threads with direct addressing of the threads would be helpful.

In his textbook on operating systems, Tanenbaum describes `sleep` and `wake-up` primitives when introducing the *producer-consumer* problem [Tan09]. The `sleep` primitive suspends the current thread in the operating system, and the `wake-up` primitives wakes up one (sleeping) thread by a thread ID or a reference to the thread.

However, in their simple form, the primitives expose a race condition due to the problem of *missed wake-ups*. This problem is solved by introducing the *"wake-up waiting" switch* by Saltzer [Sal66]. This is a flag that records a `wake-up` event in case the target thread is not yet sleeping. When the flag is set, `sleep` clears the flag and returns immediately. With this, the mechanism is now *commutative*, and the order of the two operations no longer matters for correct signalization, like in `sem_wait` and `sem_post` for binary semaphores.

In Section 2.3.4, we described the related `sched_wait` and `sched_wakeup` functions of the low-level scheduler API used inside an operating system kernel. Here, a different technique that uses a *wait indicator* flag prevents missed wake-up problems. The flag is set by a *start waiting primitive* (see also the building blocks of blocking synchronization in Section 3.4.1) inside a critical section, and evaluated by the final waiting primitive `sched_wakeup` again. If the flag changed in the mean time, the thread was successfully signaled. The *split* of the waiting primitive into two operations follows the design of Reed's *eventcounts* [Ree76], but the *state* encoded in the wait indicator flag still follows Saltzer's simpler wake-up waiting flag with two states instead of a counter.

The split of the waiting primitive into two operations is a key technique that allows to block *outside* of a critical section. We can introduce the same split for the wake-up side as well: the *start wake-up primitive* modifies the wait indicator inside a critical section, and the *wake-up primitive* actually wakes up the thread in the kernel. With this, the resulting sequence to wake up a thread is for example: *lock critical section* $\rightarrow$ *start wake-up primitive* $\rightarrow$ *unlock critical section* $\rightarrow$ *wake-up primitive*. As a wake-up operation is not blocking compared to a waiting operation, we do not technically need to decouple the wake-up operation. However, the benefit of this approach is that it helps to keep the critical section short, as any system calls are now outside the critical section and do not inflate the WCET of the critical section.

However, a wait indicator using just two states can cause *spurious wake-up* problems. Consider a situation with two threads executing concurrently on two processors, as Figure 3.7 shows. The first thread A is a waiting thread and performs two waiting operations back to back. The second thread B wakes up the waiting thread from the first blocking point, but is delayed after signaling the wait indicator. Now, thread A performs the first wait operation and sets the wait indicator to indicate waiting ①. The thread leaves the critical section and performs the system call to wait in the kernel ②. In the mean time, thread B

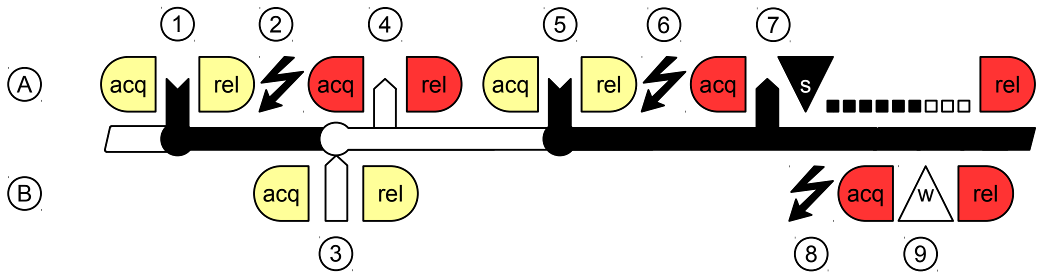

Figure 3.7: Spurious wake-up problem with simple wait indicators. Thread A performs two wait operation back-to-back. Concurrently, thread B performs a wake-up operation for the first wake-up, but the wake-up system call is delayed, e.g. by an interrupt. Now thread B spuriously wakes up thread A in its second wait operation.

---

performed a wake-up operation and signaled the wait indicator (3). In turn, the waiting system call of thread A sees the signaled wait indicator and does not block thread A (4). Thread A returns from the system call and performs the second wait operation. It sets the wait indicator to waiting state again (5), calls the kernel (6), and successfully blocks (7). In the mean time, thread B was delayed before it could finish the wake-up operation. Now B continues, calls the kernel (8) and spuriously wakes up thread A (9).

This particular problem can be solved by adding a check of the wait indicator to the wake-up operation. However, the solution would not scale if we add another thread C to the game that sets the wait indicator to signaled state before B calls the kernel. Instead, a wake-up operation should only succeed for the *matching* wait operation. This problem can be solved in multiple ways, e.g. by adding a barrier on the waiting side to check that the wake-up side has finished the wake-up operation, or by using a unique key for matching wait and wake-up operations. Such a unique key can be generated by a *sequencer* [RK79]. Conceptually, the sequencer is a simple up-counter. A `ticket` operation increments the counter and returns the current value. With this, a start waiting operation can start a new waiting round by incrementing the wait indicator. Conversely, a start waiting operation increments the counter for each wake-up. The system calls for waiting and wake-up then only succeed if the wait indicator is still at the expected value of the operation.

With this, we have identified the building blocks for low-level wait and wake-up operations. By using a wait indicator, we can effectively split both the wait and the wake-up operations into two phases, a first *start* phase that modifies the wait indicator variable, and a second phase that requires a system call for blocking or wake-up. The wait indicator is modified like a sequencer in the first phase, and checked like an eventcount in the second phase. This prevents both missed wake-up and spurious wake-ups problems.

3.9 Analysis Summary

We will now summarize the results of the analysis chapter.

The main insight is that *efficient* synchronization mechanisms prevent unnecessary system calls, as these have a high constant overhead (Sections 3.1.1 and 3.2.2). We analyzed blocking synchronization mechanisms, decomposed them into their building blocks, and derived a generic model of these mechanisms (Section 3.4). The key technique for a fast path is to provide a *semantic operation* in user space that decides whether a thread has to block or not. Our analysis of futexes in Linux (Section 3.5) has shown that futexes implement the semantic operation by atomic operations. We also evaluated the design space for alternatives and proposed spin-based critical sections in user space for the fast path operations (Section 3.6). However, spin-based critical sections in user space need an additional mechanism to temporarily disable preemption (Section 3.7) or they risk lock holder preemption and lock wait preemption. We additionally discussed low-level wait and wake-up mechanisms different than futexes for the spin-based critical sections (Section 3.8).

We analyzed real-time locking protocols that could help to address determinism issues and improve analyzability (Section 3.3). Most of the protocols unconditionally disable preemption or change the scheduling priority of the calling threads. As an efficient technique to implement preemption control or changes to the scheduling priority, we have identified that changes to specific state information of a thread can be implemented by variables shared between user space and the kernel instead of system calls (Section 3.7). The key technique here is *laziness* under the assumption that a critical section does not see contention and is not interrupted: user space code temporarily indicates a state change in a flag, and reverses the change before the kernel notices as fast path. In the slow path, additional synchronization with the kernel is needed, which requires a system call.

Another insight is that for *predictability*, the implementation of a synchronization mechanism must fulfill certain requirements and must be analyzable in the first place. To address the problems specific to mixed-criticality systems, we additionally require that synchronization mechanisms shared between different processes or partitions must especially address interference problems. Also, long running operations should be preemptible and their remaining non-preemptible elements should require at most $\mathcal{O}(\log n)$ time (Sections 3.2.1 and 3.2.3). We also discussed techniques to decouple critical sections by using consecutive locking in systems using fine-grained locking and the related TOWTTOS problem (Section 3.4.2) and key techniques that prevents missed and spurious wake-up problems (Section 3.8).

In general, we identified futexes and non-preemptive spin-based monitors as two promising techniques emerged to implement blocking synchronization mechanisms with a fast path in user space.

The case for more deterministic futexes: Futexes offer a generic way to blocking synchronization (Section 2.2.11). Futexes have the special property to be *registration-free*, i.e. synchronization objects do not need to be known to the kernel prior use. With this, the number of futex-based synchronization objects user space code can use is only limited by the available memory. In a system with n threads, at most n threads can wait on a futex, either on the same futex, or on n different futexes. Internally, the kernel creates the necessary wait queues on demand and destroys them after use.

However, the futex implementation in the Linux kernel has some drawbacks w.r.t. determinism (Section 3.5), in particular a hash table with shared wait queues, and a dependency to heap allocations when using priority inheritance. Summarized, Linux futexes are efficient, but not predictable. We will address these problems in Section 4.1. The presented design of *deterministic futexes* improves on the Linux implementation.

As further alternative, we evaluate the impact of changing one of the key properties of Linux futexes, namely creation of wait queues on demand. As wait queues are created and destroyed dynamically, the Linux kernel needs an indirection mechanisms to locate threads waiting on the same futex. The futex address in user space provides a unique ID. Therefore, removing the need to dynamically create wait queues on demand opens up new perspectives to address one of the pain points of the Linux implementation, the shared hash table. In such a design, the wait queues must be statically assigned to synchronization objects prior use. This clearly deviates from futexes in Linux, but retains the property of using atomic operations on futex words in user space and the higher-level synchronization mechanisms known for Linux. We will discuss such a design of *static futexes* in Section 4.2. The design is additionally focused on statically configured embedded systems.

Non-preemptive spin-based monitors: The second promising technique are non-preemptive spin-based monitors that manage a wait queue in user space and directly address threads (Section 3.6). Using a spin-based critical section gives more flexibility to implement a fast path and according wait queue operations than futexes with their limitation to atomic operations on 32-bit variables. Also, spinlocks can be implemented in user space already and do not depend on system calls. However, spinlocks are prone to lock holder preemption and lock waiter preemption problems, so we need efficient mechanisms for preemption control (Section 3.7). Further building blocks for the monitors are simplified blocking and wake-up system calls (Section 3.8).

In general, the monitor approach seems promising. However, the author of this thesis is unaware of any synthesis of these building blocks to construct arbitrary blocking synchronization mechanisms. And there are certain conflicting approaches to consider. Especially, the efficiency of the preemption control mechanisms is based on the fact that the calling thread is *not* preempted; however, blocking synchronization naturally requires to suspend or wake up threads. Therefore, we must design a preemption control mechanism that better interacts with wake-up requests during a critical section. We will address the preemption control mechanism in Section 4.4.1 and the resulting monitor approach in Section 4.4.2.

Chapter 4

Design

In the design chapter, we present three different approaches to fast synchronization mechanisms, two futex-based ones and one based on monitors with spinlock-protected critical sections in user space.

The first three sections discuss the futex-based approaches. Section 4.1 presents a *deterministic futex* design that provides a higher level of predictability at the kernel level than futexes in Linux, while providing a similar feature set. Section 4.2 presents *static futexes*, a design for statically configured systems with a subset of the features found in Linux. Section 4.3 discusses the design of higher-level synchronization mechanisms on top of futexes.

The next two sections present the second approach, light-weight *non-preemptive busy-waiting monitors.* The approach combines both *efficient IPCP implementations* as preemption control mechanism and *waiting in spin-based fine-grained locking* to turn futexes *inside-out.* Section 4.4 discusses the synthesis of these mechanisms into monitors. And Section 4.5 shows how to build higher-level synchronization mechanisms from monitors.

4.1 Deterministic Futexes

In this section, we present a futex design that addresses *some* of the determinism issues of futexes in Linux discussed in Section 3.5. The presented design focuses on the main problem of predictability, the hash table with shared wait queues.

In 2011, the author of this thesis developed a first futex implementation for PikeOS, supporting only a subset of the futex operations in Linux. Wait queues were FIFO-ordered and implemented as doubly-linked lists. Instead of using a hash table to address wait queues, the kernel kept the thread ID of the first waiting thread of a wait queue next to the futex value in user space. This allowed the kernel to locate the wait queue in $\mathcal{O}(1)$ time, as the look-up of threads by their IDs use radix trees. This *original* design was presented in a workshop paper in 2013 [Zue13].

Over time, priority-ordered wait queues were added to the futex implementation in PikeOS. Also, the PikeOS kernel design changed from a model using a global kernel lock to fine-grained locking. In 2018, the author of this thesis redesigned the kernel implementation in PikeOS to address the predictability and interference issues of the original design. This *redesigned* futex design is the focus of the following sections. It was first presented in a workshop paper [ZK18] and later in a full conference paper [ZK19].

The presented *deterministic futexes* follow the general design principles of futexes in Linux, i.e. a futex is a 32-bit atomic variable in user space, synchronization objects do not need registration, and the kernel provides compare-and-block, wake-up, and requeue primitives for general futexes and futex-based mutexes. However, the presented futex design deviates from the Linux futex API at some critical points and does not provide all interfaces, so it is not 100% compatible to the Linux futex API and also not suitable as a replacement of the futex implementation in the Linux kernel.

We start the discussion of the general design of deterministic futexes in Section 4.1.1 with the general structure of the underlying data structures, explaining key techniques, the design of the particular operations, and the available real-time protocols in the following sections.

4.1.1 Design Considerations

A futex design that aims for predictability needs to fulfill the requirements presented in Section 3.2.3. The main source of interference and non-determinism of the futex implementation in Linux is the hash table with shared wait queues that keep threads waiting on both private (per-process) and shared (global) futexes in the same data structure.

Obviously, a more predictable futex design should separate private futexes and global futexes into distinct sets. Private futexes can be kept in per-process data structures, and only shared futexes should use a shared data structure. This directly follows from *RQ6 (interference by shared namespaces)* to reduce unnecessary interference. Also, a more predictable futex design should not share wait queues between different futexes, but use *dedicated wait queues* for each futex. This follows from *RQ3 (analyzability)*.

Note that futexes do not require prior registration of synchronization objects. To *look-up a wait queue*, the kernel uses the address of a futex object in user space. *RQ3 (analyzability)* rules out solutions like hash tables for this. We use a binary search tree (BST) instead. As the binary search tree addresses the wait queues by their address, we name this data structure the *address tree*. Using BST satisfies *RQ4 (bounded operations on queues)* and bounds all operations to look-up a wait queue and to dynamically insert new wait queues and remove empty wait queues to logarithmic complexity.

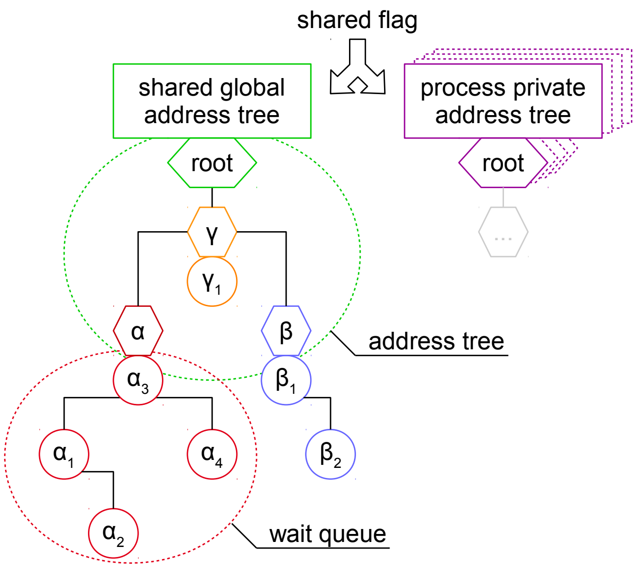

Figure 4.1: Futex architecture using nested BSTs, *address trees* and *wait queues*. The `shared` flag selects between the shared global address tree or a process private address tree. The shared global address tree comprises wait queues (hexagons) α, β, and γ. Address trees are ordered by futex addresses as keys. Wait queue α comprises four waiting threads (circles) α_1 to α_4. Wait queue β comprises two threads β_1 and β_2. Wait queue γ comprises one thread γ_1 only. Wait queues are ordered by thread priority. Threads α_3, β_1, and γ_1 are wait queue anchors of their wait queues. BST node data is kept in the TCBs of the involved threads. Figure taken from [ZK19].

For *RQ1 (correctness)*, we have to consider the following functional requirements of the operating system standards: POSIX requires that threads with the highest scheduling priority must be woken up first [IEE17]. This requires to use a priority-ordered wait queue. However, ARINC 653 and other (non-POSIX) purposes also require FIFO ordering. We implement the wait queue as BST again. A BST can support both priority and FIFO ordering and satisfies *RQ4 (bounded operations on queues)*.

From this follows that a predictable design should use the following architecture: (i) distinguish between global and private futexes, (ii) use the right *address tree* to locate the wait queue with the futex address as look-up key, and (iii) keep blocked threads in a dedicated per-futex *wait queue*. Figure 4.1 shows an overview of the architecture.

For locking in the presented design, we first considered to use hierarchical fine-grained locking with a lock for the address tree and dedicated locks for each wait queue. The kernel would first lock the address tree, locate a wait queue, lock the wait queue, and then unlock the address tree again. The hierarchical order in which locks are taken prevents deadlocks. However, an empty (and locked) wait queue cannot be removed safely from the address tree without holding the address tree lock. The kernel would have to unlock the wait queue first, then lock the address tree, and finally lock the wait queue again. But this *re-locking* exposes races, as the now re-locked wait queue may no longer be empty due to concurrent insertion. We assume that a solution can be found, e.g. using a lock-free look-up mechanism in the address tree to locate wait queues, but it is questionable if such an approach would be able to provide the required level of predictability, especially *RQ5 (bounded loops and retries)*. We decided against fine-grained locking and use a *single lock* that protects both the address tree and all its associated wait queues instead.

A design that uses single lock for both data structures is fine for most futex operations that operate on just one thread. In this case, all operations are already properly bounded for predictability. But operations that address more than one thread, i.e. `futex_wake` or `futex_requeue`, pose a problem. An artificially restriction to process only a small number of threads would solve the problem (this just increases the upper bound), but finding the right limit is hard and this would seriously cripple the flexibility of futexes. Instead, we implement these long-running operations in a preemptive way and follow *RQ7 (preemptive operations)*.

At first glance, such an architecture seems trivial to implement. One could simply allocate a new wait queue data structure on demand and insert it into the address tree. However, recall that *RQ2 (robustness)* demands not to use dynamic memory allocations. Instead, all data related to futex management must be kept inside the *thread control block* (TCB) of the blocked threads. Also, preemptible operations are tricky. Recall the requirements for *RQ8 (termination)* and *RQ9 (hidden transience)*.

4.1.2 Binary Search Trees

From a BST implementation, we require the standard operations $find$, min/max, $insert$, and $remove$, and additionally *root* and *swap*. Nodes in the BST use three pointers: two for the left and right child nodes, and a third one to the parent node. The *root* operation locates the root node of the BST from any given node in $\mathcal{O}(\log n)$ time. The *swap* operation allows to swap a node in the tree with another node outside the tree in $\mathcal{O}(1)$ time without altering the order in the tree. Lastly, the BST implementation requires a *key* to create an ordered tree. The key may not be unique, e.g. threads with the same priority are allowed to exist in the tree. We require FIFO ordering of nodes with the same key.

4.1.3 Address Tree Management

Separate address trees roots: Shared and private futexes are kept in different address trees, as Figure 4.1 shows. Like in Linux, the user specifies in a `flags` argument for each futex operation if futexes are shared or private. Shared futexes are kept in a global tree shared among all processes, while private futexes are kept in the process descriptor of each process. The root of an address tree is called *address tree root*.

Address tree: Address trees are ordered by increasing futex addresses as *key*. To look up a wait queue by its futex address, we designate *one* of the blocked threads in a wait queue as *wait queue anchor*. The anchor thread then holds the root pointer to the wait queue. Hence, the address tree consists of wait queue anchors as nodes.

Futex key: For shared futexes, the kernel uses the *physical address* of the futex as key; and for private futexes, the kernel uses the *virtual address* as key. We use the fact that futex variables in user space are naturally aligned 32-bit integers. As the last two bits of a futex address are always zero, the kernel uses them to encode further information.

Open/close state: We define that a wait queue is *open* if threads can be added to it, i.e. new threads can block on a futex, and a wait queue is *closed* if new threads cannot be added. The kernel encodes the state of a wait queue in its key. An open wait queue has the lowest bit *set* in the key, for a closed wait queue the bit is *cleared*. By clearing the open bit, the kernel can change a wait queue from open to closed state without altering the structure of the address tree.

Drain ticket: For closed wait queues, we also define a *drain ticket* attribute. The drain ticket determines the age of a closed wait queue. The drain ticket is drawn from a global 64-bit counter that is incremented each time a wait queue is closed. This counter should not overflow in practice.

Closed wait queues: We do not allow open wait queues with duplicate keys, as each key relates to a unique futex in user space. But multiple *closed* wait queues with the same key may exist. The closed wait queues become FIFO-ordered due to the ordering constraints in the BST when changing a wait queue from open to closed state. The closed wait queues also have increasing drain tickets that help us to distinguish older from newer closed wait queues. We later exploit this mechanism to wake up and requeue threads in a preemptible fashion.

Summary: This design for the address tree allows us to perform look-up, insertion, and removal of wait queues in $\mathcal{O}(\log n)$ time. Changing a wait queue from open to closed state needs $\mathcal{O}(1)$ time. Thus, the design fulfills *RQ4 (bounded operations on queues)*.

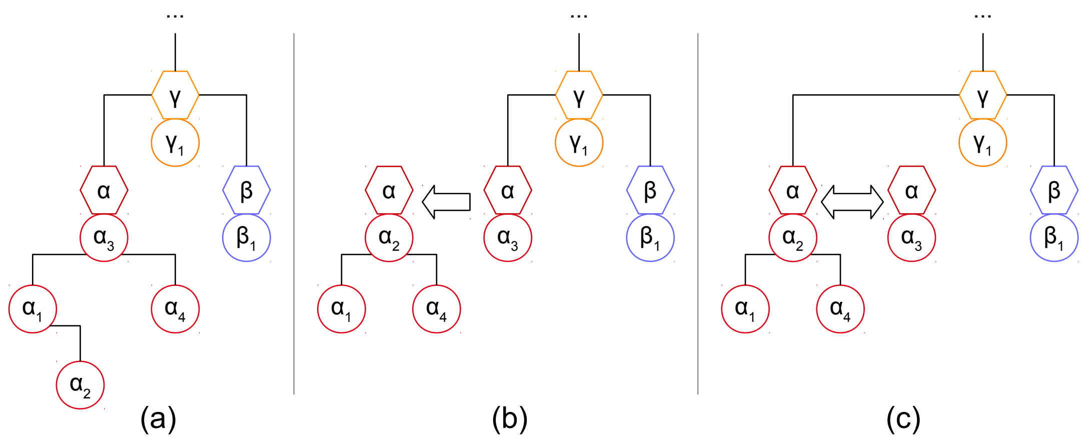

Figure 4.2: Wake-up of wait queue anchor thread. Step (a) shows thread α_3 as wait queue anchor thread. Thread α_3 is then removed from its wait queue. After rebalancing of the BST, thread α_2 becomes the new wait queue anchor. In step (b), wait queue state information is copied from α_3 to the new anchor α_2. In step (c), nodes α_2 and α_3 are swapped in the address tree, making α_2 the new wait queue anchor. Thread α_3 is no longer referenced afterwards. Figure taken from [ZK19].

4.1.4 Wait Queue Management

Wait queue anchor: As stated before, the wait queue anchor thread is an arbitrarily chosen thread that holds the root pointer of the wait queue and other wait queue attributes. We refine this now and define that the thread being the *current root node* of the wait queue is to be used as anchor. If the current root node changes due to rebalancing in the wait queue tree, the kernel first copies all wait queue attributes from the old anchor to the new anchor thread, and then *swaps* the old anchor thread in the address tree with the new current root node without altering the structure of the address tree. Figure 4.2 shows this in an example. Using the root node thread as anchor for a wait queue is not mandatory, as any node in the wait queue would do. But this simplifies the implementation when threads must be woken up for other reasons, e.g. timeout expiry.

Creation and destruction of wait queues: When a thread blocks on a unique futex address, the kernel creates a new wait queue in open state and inserts it into the address tree with this first thread as anchor. Note that this does not involve allocation of memory, as the pointers comprising the wait queue are kept in the blocked thread's TCB. Similarly, the wait queue is implicitly destroyed when the last thread (that again must be the anchor) is woken up. The kernel then removes the wait queue from the address tree.

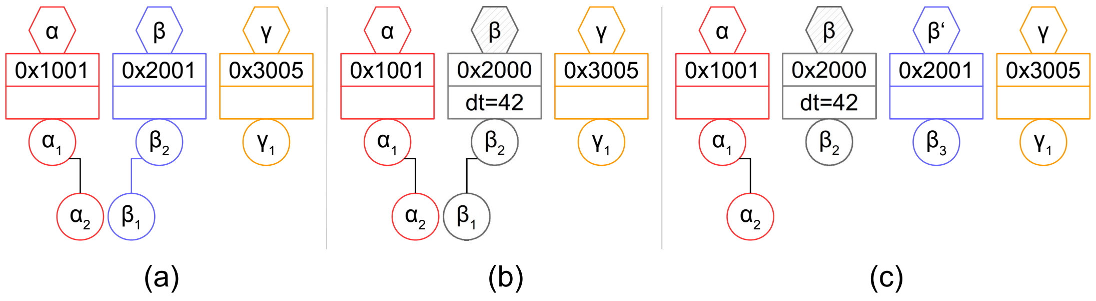

Figure 4.3: Preemptible draining of wait queue β. Step (a) shows three wait queues α, β, and γ with their futex keys. The lowest bit in a futex key indicates an open wait queue. In step (b), wait queue β is closed and the lowest bit in the key is cleared. The wait queue is assigned a drain ticket of 42. In step (c), while the previous wait queue β is emptied preemptively, a new wait queue β' is inserted with the former open key. Figure taken from [ZK19].

Insertion and removal in wait queues: The kernel supports both FIFO- and priority-ordered wait queues. The user specifies the queuing discipline in a `flags` argument for each futex operation. To support both modes with the same data structure, the kernel defines a waiting priority for each thread. In FIFO mode, the waiting priority is set to 0 for all waiting threads. In priority mode, the waiting priority reflects the scheduling priority of a thread. The kernel inserts threads in waiting priority order into an existing wait queue, with FIFO order on tie. Also, the kernel removes the thread with the highest priority first in wake-up and requeue operations.

Removal of an arbitrary node, e.g. on a timeout, requires to find the associated wait queue root to rebalance the tree afterwards. The kernel does not look up the wait queue in the address tree in this case, as it might have been set to closed state in the mean time. Instead, the kernel simply traverses the wait queue tree to the root node to locate the wait queue anchor and remove the thread. This is also necessary when a thread's scheduling priority changes while the thread is blocked. In this case, the kernel first removes and re-inserts the thread with an updated waiting priority.

Summary: This design allows to perform all internal operations on wait queues in $\mathcal{O}(\log n)$ time and fulfills *RQ4 (bounded operations on queues)*. From this, we can now construct all futex operations that operate on one thread.

4.1.5 Preemptible Operation

We now discuss preemptible operations on multiple threads: The overview of futex operations in Section 2.2 shows that in all cases targeting multiple threads, the kernel always wakes up or requeues *all* threads in a wait queue. Therefore, we restrict the implementation of `futex_wake` and `futex_requeue` to operate on either *one* or *all* threads, but not on an arbitrary number, as Linux allows. This simplifies the operations, as we simply can close a wait queue and then process all its threads preemptively.

Therefore, when these operations target all threads, the kernel sets the wait queue to closed state first, draws a unique *drain ticket*, and saves the drain ticket in the anchor node in the wait queue. A closed wait queue can no longer be found by other operations. This prevents already woken or requeued threads from re-entering a wait queue again, as Figure 4.3 shows in an example.

Then the kernel wakes up or requeues one thread after another, but provides a preemption point after handling each thread. After each preemption, the kernel must look up the closed wait queue again. If multiple closed wait queues with the same key are found, the stability in the BST ensures that nodes are ordered by increasing drain ticket numbers. The kernel then continues to perform its operations as long as the drain ticket number is less than or equal to its originally drawn drain ticket. If the drain ticket number of a node is less than the originally drawn ticket, the wait queue relates to an *older*, but still unfinished operation. In this case, the operation drains older wait queues on behalf of other threads as well.

Summary: Since at most $n - 1$ threads can be blocked before a draining operation starts and a drain ticket is drawn, the upper limit of steps to complete a `futex_wake` or `futex_requeue` operation is therefore n. This design satisfies *RQ7 (preemptive operations)* and *RQ8 (termination)*, but not *RQ9 (hidden transience)*.

4.1.6 Interference of Shared Futexes

One important aspect is to prevent unnecessary sharing of futexes. Like Linux, the presented design supports both private and shared futexes. Private futexes are always kept in a process-specific address tree root and therefore do not cause interference with other processes. However, shared futexes use a global address tree root and cause interference. Therefore, we suggest that shared futexes require an explicit capability to prevent applications from causing interference at all and to simplify the analysis.

Shared futexes require an analysis of *all* applications that potentially use shared futexes to determine the WCET of futex operations. If we assume that the system has m processors, the kernel uses fair *ticket spinlocks*, and m independent processes currently using sharing futexes, then a process can observe $m - 1$ concurrent–but preemptive–futex operations in the worst case. Shared futexes are therefore sub-optimal with respect to *RQ6 (interference by shared namespaces)*. Still, the global interference in bounded.

4.1.7 Summary

We presented the design of a predictable futex implementation at the kernel level. The design provides a subset of the features and interface of the Linux implementation, but also provides new features not available in Linux, such as FIFO-ordered wait queues. An implementation of the user space parts can follow the implementation in a C-library in Linux.

In particular, the presented futex design does not support the *priority inheritance protocol* (PIP) for mutexes as mandated by POSIX. Support for PIP must be implemented at kernel level.

As an alternative to PIP, POSIX describes the *immediate priority ceiling protocol* (IPCP) for mutexes. A `mutex_lock` raises the calling thread's scheduling priority to the defined ceiling priority of a mutex before locking the mutex and lowers the scheduling priority after unlocking. IPCP can be implemented efficiently with the protocols described in Section 3.7 and Section 4.4.1. For a mutex-based critical section without contention, this would requires no system call overhead, similar to uncontended futex operations.

4.2 Static Futexes

This section describes a futex design named *static futexes* which is especially suitable for statically configured systems. Nevertheless, the results are portable to dynamic systems as well.

In a statically configured system, all resources are known at compile time and do not need to be allocated at runtime. For example, in OSEK and AUTOSAR, a whole system is described in a *system configuration*, and code generators create fixed size arrays for all resources at compile time or link time. The resources can be indexed efficiently by their position in the arrays [OSE05]. Because of this, kernels for statically configured systems tend to be much simpler than kernels for dynamic systems.

Note that allocating wait queues statically changes a main property of futexes that futexes are *registration-free*. The resulting design clearly deviates from futexes in Linux.

Section 4.2.1 starts with a discussion on general design considerations for static futexes. Section 4.2.2 then presents the design of static futexes in the context of the AUTOBEST research kernel.

Static futexes were first presented in a conference paper by the author on AUTOBEST [ZBL15].

4.2.1 Design Considerations

Recall the discussion on futexes in Linux in Section 3.5. When using futexes, the kernel is involved only in the case of contention. In this case, the kernel allocates a wait queue on demand when the first thread starts to wait, and frees the wait queue when the last thread was woken up. Internally, a futex object's wait queue is referenced by hashing the futex user space variable's address. For futexes shared between different address spaces, Linux uses the physical address of the futex variable for hashing rather than its virtual, per address-space address.

For an implementation of futexes in statically configured systems, we lift the requirement that futexes require no prior registration in the kernel. Instead, all futex-based synchronization objects must be known at system integration time. Then the futex wait queues can be statically allocated and referenced by consecutive indices. In a more dynamic scenario, this relates to allocation of all wait queues during the initialization phase of an application.

Consequently, neither address hashing nor dynamic creation of wait queues are necessary. Such a static design then easily satisfies the requirements for predictability of Section 3.2.3, especially *RQ2 (robustness)*, due to the lack of dynamic allocation at runtime.

Still, there are many degrees of freedom to consider in the design:

- **Indexing wait queues:** We already mentioned that the futex wait queues can be effectively addressed by their array index. To satisfy *RQ6 (interference by shared namespaces)*, private futexes should not be shared. This can be achieved by giving a partition access to specific ranges in the array. Similarly, sharing of futexes can (and should) be explicitly configured. Note that when using a shared futex, one side is often only using wait operations, and the other side performs only wake-up operations. This also allows to use different namespaces for wait and wake-up operations on shared futexes or in general.

- **Futex types:** The type of a synchronization object (e.g. mutex, condition variable, or generic blocking) can be provided in runtime in the API or hard-coded at system configuration time. Both the waiting side and the wake-up side must agree on the futex type.

- **Queuing discipline:** The queuing discipline defines if a wait queue uses priority or FIFO ordering. The queuing discipline is a property of the waiting side. It can be set at runtime or at system configuration time.

- **Futex value in user space:** A futex value in user space and a compare value in the API of the waiting operation are still needed to handle missed wake-ups and for *RQ1 (correctness)*. The futex *address* can be provided at initialization time in the API or hard-coded at system configuration time.

- **Wait queue implementation:** In a static system, the maximum number of waiting threads is known at compile time, so requirements *RQ3 (analyzability)* and *RQ4 (bounded operations on queues)* can be easily satisfied. This even allows to use linked lists with $\mathcal{O}(n)$ time for insertion in priority-ordered wait queues.

- **Timeouts and cancellation of waiting:** When timeouts are supported, waiting operations can be cancelled. Cancelling a waiting operation needs extra care, as spurious wake-ups become an intended feature.
- **Handling of multiple threads:** Wake-up and requeue operations on multiple threads can either process a given number of threads in the API, or *all* threads. See Section 4.1.5 for the related argument in the deterministic futex implementation.
- **Requeue operation:** The requeue operation can be optimized to ignore "naked notifies" and to only handle requeuing from condition variables to mutex transitions, or even be omitted if no condition variables are used[1].
- **Locking architecture:** Dedicated wait queue objects for each futex allows fine-grained locking at wait queue level. However, often a coarse lock is sufficient as well, especially if the scope of the threads waiting on the wait queue is restricted to a subset of the available processors. Also, using a single lock for all futexes in a partition simplifies the implementation of requeue operations.
- **Non-preemptible operations:** A non-preemptible design is much simpler than a preemptible design. With a known maximum number of waiting threads, requirement *RQ5 (bounded loops and retries)* is satisfied, and the problems described in *RQ7 (preemptive operations)*, *RQ8 (termination)*, and *RQ9 (hidden transience)* can effectively be prevented.

With this, a system design can carefully consider the trade-offs.

4.2.2 Futex Operations for ARINC 653 in AUTOBEST

In AUTOBEST [ZBL15], the author opts for the following detailed design for static futexes.

AUTOBEST is a partitioning microkernel that supports both AUTOSAR and ARINC 653. The kernel supports multi-processor systems, but partitions are always assigned to a single processor. The kernel provides an efficient implementation of IPCP in user space.

Only ARINC 653 partitions use futexes. The ARINC 653 standard does not define condition variables, therefore we do not provide a requeue operation. The standard also does not mandate a static configuration for synchronization

[1]Note that a static design allows fine-grained tailoring of *all* futex operations. We did not explicitly mention this because the description tries to cover a design that is feature-wise comparable to a normal futex implementation.

mechanisms used internally by a partition. Therefore, we keep most of the design considerations described in Section 4.2.1 flexible and allow to configure futexes at runtime. The system configurator must provide a sufficient number of futex wait queues to an ARINC 653 partition.

For the futex operations in the kernel, we settle on a non-preemptible design. We use linked lists for the wait queue implementation. Recall that ARINC 653 allows both priority- and FIFO-ordered wait queues and the queuing discipline is configured at runtime.

Our intended use case for futexes covers both partition internal synchronization and queuing ports. The queuing port use case requires access to a wait queue from different partitions. Therefore, we consider the operations on a wait queue as a *two-ended directed communication channel*: one side *waits* for the other side to send a notificating *wake-up*. This allows us to grant both ends to two different partitions without knowing where the other end resides.

We then design the futex API accordingly. In the API, futex wait queues are addressed by partition-specific indirection tables that point to the right wait queues in the kernel. The indirection tables are generated from the system configuration. The waiting side then defines the queuing discipline and can only block with timeout, but never wake up threads. The notificating side can only wake up one or all threads, but never block. Note that "normal" futexes have both ends granted at the same time.

4.3 Higher-Level Synchronization Mechanisms based on Futexes

In this section, we present the design of higher-level synchronization mechanisms based on futexes. The discussed mechanisms are compatible to the deterministic futexes presented in Section 4.1 and the static futexes presented in Section 4.2. The mechanisms are usually implemented in user space and require the futex interfaces presented in Section 2.2.11 from the operating system kernel. Therefore, the main focus is on *RQ1 (correctness)*, *RQ3 (analyzability)*, and *RQ5 (bounded loops and retries)* of the requirements of Section 3.2.3.

For each presented synchronization mechanism, we *briefly* summarize the key aspects of the design and the used futex protocol, i.e. the encoding and rules of the futex value in user space. We present mutexes (Section 4.3.1), condition variables (Section 4.3.2), counting semaphores (Section 4.3.3), barriers (Section 4.3.4), and one-time initializers (Section 4.3.5). We conclude with a design for ARINC 653 queuing ports and an overview of other ARINC 653 synchronization mechanisms (Section 4.3.6).

The descriptions of the ARINC 653 queuing ports is partly taken from the author's paper on AUTOBEST [ZBL15].

4.3.1 Blocking Mutexes

Mutexes as described in Section 2.2.2 can be abstracted upon the generic futex interface or on the specialized futex interface for mutexes, both presented in Section 2.2.11. Using the generic interface leads to the problem that after waiting for a mutex once, the user space code does not know if there are still other threads waiting in the kernel, so an implementation always has to set the `WAITERS` bit for correctness (see Section 3.5.2 and [Dre11]). Therefore, we use the specialized futex interface with `futex_lock` and `futex_unlock` in the following.

The protocol follows the description of PI mutexes in Linux in Section 3.5.2.

Protocol: For a mutex, both user space and the kernel must understand this protocol. The futex value comprises two pieces of information: the thread ID (`TID`) of the current lock holder or 0 if the mutex is free, and a `WAITERS` bit if the mutex has contention.

In the fast path (no contention), both `mutex_lock` and `mutex_unlock` try to atomically change the futex value from `0` to `TID` and vice versa, without calling the kernel. On success, both operations must also provide memory barriers with *load-acquire* (lock) and *store-release* (unlock) semantics.

If `mutex_lock` finds an already locked mutex, the function calls the kernel to suspend the calling thread on the futex. The `futex_lock` operation in the kernel checks the futex value again and tries to either acquire the mutex for the caller if it is free, or, if not, atomically sets the `WAITERS` bit in the futex value to indicate contention, and suspends the calling thread. On successful return from `futex_lock`, the calling thread is the new lock owner.

Conversely, if `mutex_unlock` detects that the `WAITERS` bit is set, it calls the kernel to wake up a waiting thread. If no threads are waiting, `futex_unlock` sets the futex value to `0`, or wakes up the next waiting thread and makes it the new lock owner by updating `TID` in the futex value. The kernel also sets the `WAITERS` bit again if other threads are still waiting.

Implementation considerations: In a minimal implementation, the mutex data comprises just the 32-bit futex value. However, the mutexes described in Section 2.2.2 support different real-time protocols and handle different modes. This requires additional data that is usually kept in adjacent fields in a mutex object.

The user space code also needs access to the thread ID of the calling thread. This information is usually kept in thread local storage (TLS), so a thread can obtain its `TID` without a system call.

So far, the protocol follows the description for Linux in Section 3.5.2. To prevent the problem of unbounded atomic operations identified in Section 3.5.3, we must additionally limit the number of retries in the kernel. Therefore, we let the kernel retry the atomic operation a few times before returning to user space with an error. The design space would also allow to let the kernel fail immediately and let user space restart the operation. Similarly. the user space code could limit the number of lock attempts to bound the overall execution time.

Note that `futex_unlock` does not need an atomic CAS operation to update the futex value in user space. A simple store will do.

Summary: The design shows that atomic operations are problematic w.r.t. WCET analysis. The presented design provides an upper bound for the loops in the kernel (and solves the problem identified in Section 3.5.3), but the loops in user space are still unbounded. This violates *RQ5 (bounded loops and retries)* of Section 3.2.3.

4.3.2 Condition Variables

The condition variables presented in Section 2.2.3 use a *requeue* operation when signaling waiting threads. We define that a requeue operation requires a generic wait queue as source and a mutex wait queue as destination. For this, the kernel must remember the futex types internally and check for compatible wait queues. This removes unnecessary corner cases and limits requeuing to happen only once (unlike Linux).

A system designer must also decide whether the waiting side or the waking side specifies the requeue target. Providing the mutex for requeuing in `cond_wait` has the benefit that the signaling side does not need to know the support mutex. This is necessary for "naked notifies" where the caller does not need to have the support mutex locked when signaling the condition variable. We further assume this design.

We also assume that the kernel requeues threads preemptively and processes one thread at a time as described for deterministic futexes in Section 4.1 to satisfy *RQ7 (preemptive operations)*. In this case, the requeue operation must check for each thread if the target mutex is currently free, and then potentially lock the mutex for the processed thread. Otherwise, the kernel must ensure that the `WAITERS` flag is set properly

Protocol: In the presented protocol for condition variables, the futex value in user space represents a counter that is incremented on each wake-up operation. The kernel does not care about the protocol when suspending threads on a condition variable, so the generic *compare-equal* semantics apply for blocking.

For waiting, we follow the basic mechanism of an eventcount described in Section 3.8. The `cond_wait` operation reads the condition variable's counter value, unlocks the associated mutex, and then calls the kernel to block on the condition variable with an optional timeout. Here, `cond_wait` passes the futex address of the mutex to the kernel as well. In turn, `futex_wait` checks if the current counter value still matches the previously read value before blocking the caller.

For signaling, both `cond_signal` and `cond_broadcast` increment the counter and call the kernel to requeue either *one* or *all* blocked threads from the condition variable's wait queue to the mutex's wait queue. In the kernel, `futex_requeue` then preemptively processes one thread at a time and checks the futex value as described. Additionally to requeuing, `futex_requeue` must clear any pending timeouts of requeued threads. The processed threads will effectively wait for the mutex with an infinite timeout.

After wake-up, `cond_wait` needs to check the cause of the wake-up: if the caller was requeued, the condition variable must have been signaled, and the caller already owns the mutex. Otherwise, if the *compare-equal* step in the kernel failed or the timeout expired, the caller was not requeued to the mutex's futex and the function needs to lock the mutex again. Note that in the case the comparison of the futex value failed, the calling thread cannot know if the notification was for it or for another thread already waiting in the kernel. Inadvertently, `cond_signal` wakes up multiple threads in this case. The protocol exposes spurious wake-ups to prevent missed wake-ups problems.

Implementation considerations: Due to internal mutex acquire operation, requeue operations require atomic operations on user space values, which must be bounded for WCET analysis. The strategy described in Section 4.3.1 for the mutex protocol can be used here as well. If a certain number of atomic operations failed, the currently processed thread can be woken up with an error. The thread then must lock the mutex with a new system call. Note that dropping support for naked notifies could help here, as the kernel then only needs to set the `WAITERS` bit and could do this non-atomically, as the futex value of the mutex must contain the `TID` of the calling thread.

Another problem are the spurious wake-ups in `cond_wait`. In this case, the thread must apply for the mutex again with an additional system call. The alternative to directly move the thread to the mutex wait queue when the comparison of the futex value fails could be considered an option for efficiency, but this does not solve the correctness problem that the notification might not have been for the thread. Also, it requires the kernel to handle the full complexity to lock the mutex again and still requires the fallback in user space if the maximum number of retries on the atomic operations is reached. Similarly, we could consider to requeue the thread to the mutex when the timeout expires. This would push the complexity of requeuing into the kernel's timer handling.

The presented `cond_wait` operation exposes a race condition due to an ABA problem that may result in a *missed wake-up*. Missed wake-ups are normally prevented by the kernel comparing the futex value, but if—between the time `cond_wait` unlocks the mutex in user space and the time the kernel checks the futex

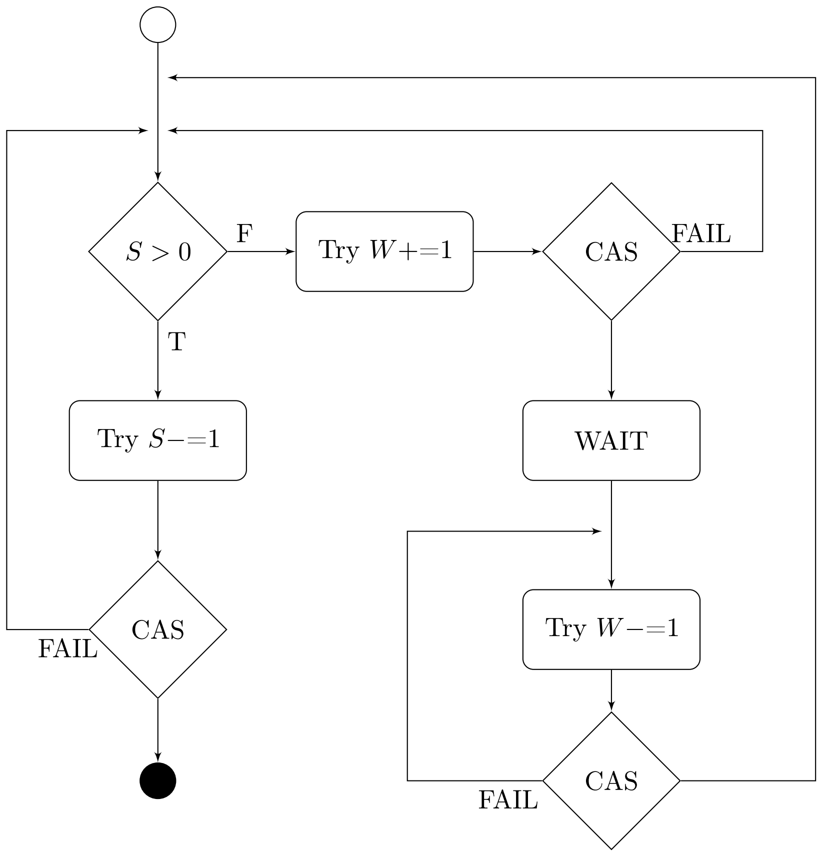

Figure 4.4: Blocking on semaphore (P operation) in a futex-based semaphore protocol. The futex value encodes two counters: S is the current semaphore count, and W shows the number of pending P operations and potentially blocked threads.

value—exactly 2^{32} wake-up operations are performed, the futex value overflows to exactly the same value and the check would erroneously succeed. However, this problem is unlikely to appear in practice.

Summary: We have assumed a preemptive design of the requeue operation in the kernel that satisfies *RQ7 (preemptive operations)*. We can observe that support for "naked notifies" violates *RQ5 (bounded loops and retries)*. Also, the design shows race conditions leading to missed wake-ups or spurious wake-ups. The overflow of the futex value leading to the missed wake-up problem is *improbable for a real-time system*, as the system must be heavily overloaded by higher priority threads consuming all CPU time for a very long time for the race condition to trigger, therefore we do not consider this to be a problem for *RQ1 (correctness)*. However, the spurious wake-ups are unpleasant, but acceptable in domains like POSIX.

4.3.3 Counting Semaphores

Counting semaphores were discussed in Section 2.2.5. In the following, we consider that a P operation with a timeout is used, e.g. `sem_wait` in POSIX, and a traditional V operation.

Protocol: A first idea for a futex protocol is to encode the semaphore counter in the futex value. Threads would then wait in a P operation when the counter is zero. But a V operation would then never know if there are threads waiting and always has to call into the kernel to wake a potentially waiting thread. Therefore, a performing protocol must encode *both* the current counter value of the semaphore *and* information on the number of blocked threads. For this, the obvious idea to express the number of waiting threads as negative counter value like in Dijkstra's *T.H.E* system [Dij68] does not work correctly due to spurious wake-ups of futexes.

Instead, we define that the futex value comprises two counters. A counter S encodes the current semaphore count. The second counter W encodes the number of pending P operations. Now the value of the semaphore counter S never becomes negative and always represents the number of available resources. At the same time, the number W of *potentially* blocked threads is known.

A P operation as shown in Figure 4.4 first evaluates the semaphore counter S. If non-zero, the operation tries to atomically decrement the semaphore counter and succeeds. If S is zero, the P operation increments the waiter counter W and then blocks. After wake-up, the operation decrements the waiters counter W again and restarts the sequence.

Conversely, a V operation always increments the semaphore counter S and wakes a waiting thread when W indicates potentially blocked threads.

Implementation considerations: An implementation of the P operation has to take care of multiple issues for spurious wake-ups. Firstly, the comparison in the futex system call can fail when another waiting thread arrives and also increments the waiting counter W. Secondly, a later P operation can *steal* a resource before a signaled thread had the chance to acquire it. The signaled thread then must wait again.

Note that due to the stealing, the P operation might need to call `futex_wait` multiple times. But then it is not longer robust when using *relative timeouts*, as the kernel then potentially waits for the relative timeout again. This is not a problem when the timeout is encoded as an *absolute expiry time.*

Summary: In a futex-based design of counting semaphores, the kernel parts do not require any loops or atomic operations on the futex value. Instead, the protocol exposes unbounded loops and spurious wake-ups, which is bad for *RQ3 (analyzability)* and *RQ5 (bounded loops and retries).* Also, relative timeout values are problematic for *RQ1 (correctness).*

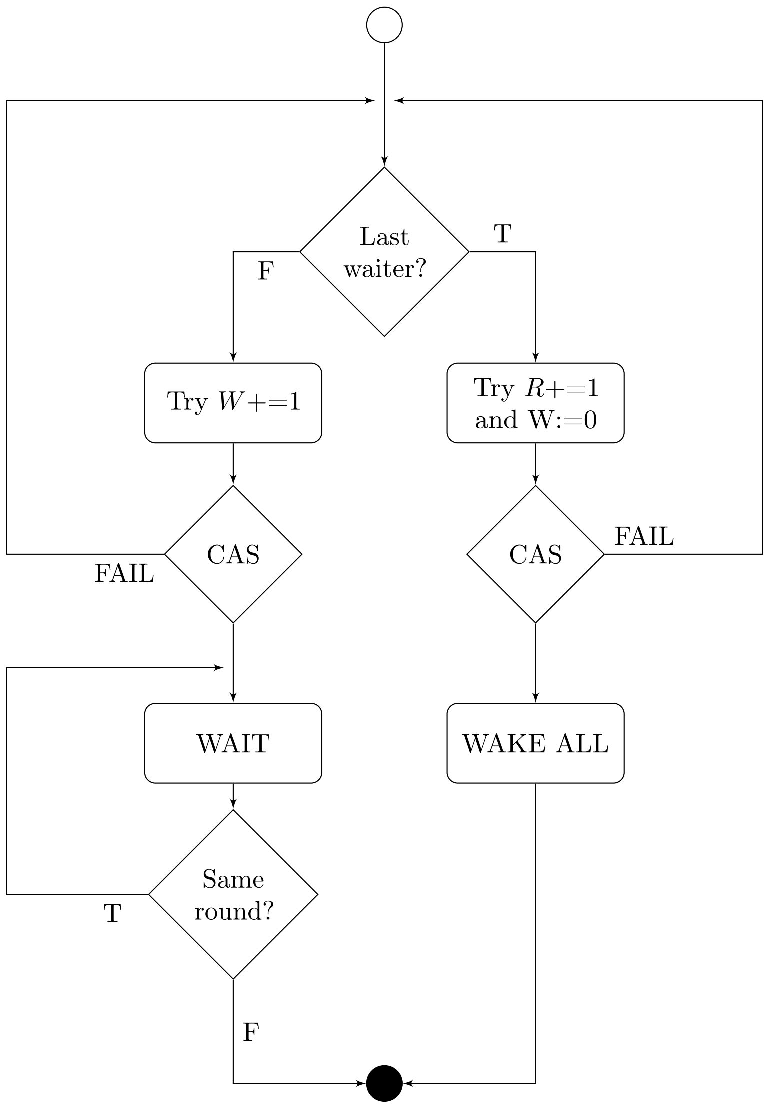

Figure 4.5: Futex-based barrier operation. The futex value encodes two counters. W tracks the number of currently waiting threads. R contains the current waiting round.

4.3.4 Barriers

Barriers were described in Section 2.2.6. A barrier is initialized with a given number of threads to wait for. Threads reaching the barrier wait until the last thread arrives. The last thread wakes all threads waiting on the barrier. Waiting on a barrier uses an infinite timeout.

POSIX allows that *more* than the given number of threads can call the barrier concurrently. This could cause a race condition when a thread of the next round arrives and waits before the last thread of the previous round could wake up the waiting threads.

Protocol: The barrier protocol encodes two counters in the 32-bit futex value to prevent uncertainty on the current waiting round. We further assume both counters are 16 bit wide. The first counter W tracks the number of already arrived and waiting threads. This value is incremented atomically each time a thread reaches the barrier. The second counter R encodes the current waiting round. This value is incremented by the last thread reaching the barrier. The last thread also resets the waiting counter W and wakes up all waiting threads. Additionally, we must keep the given number of threads that must reach the barrier next the futex value. Figure 4.5 shows a flowchart of the protocol in use.

Implementation considerations: A barrier implementation must handle spurious wake-ups due to the *compare-equal* condition in the kernel when other threads arrive and increment the number of waiting threads W in the futex value at the same time a thread calls `futex_wait` to block in the kernel. Note that the number of times this can happen is bounded by the number of threads waiting on the barrier. The spurious wake-up due to an increment of the round counter R is intended and effectively prevents missed wake-ups.

Also, the implementation shows an ABA problem when the round counter R is incremented 2^{16} times after a thread was woken up, but not scheduled. We consider that this problem does not happen in reality, however 2^{16} increments have a higher likelihood for failure than the 2^{32} increments in the condition variables of Section 4.3.2. As mitigation, the number of waiting threads W can be reduced to free more bits for the round counter R.

Summary: Like the semaphores discussed in Section 4.3.3, we have to manage two counters consistently and place them into a single futex value, but this causes failures in the *compare-equal* check in the kernel due to changes in the futex value of the part unrelated to the blocking condition. Still, the number of retries is bounded for *RQ5 (bounded loops and retries)*.

4.3.5 One-time initializers

One-time initializers were presented in Section 2.2.7. The interface comprises a single function that accepts a callback function as parameter. The callback is invoked only once on the first invocation of the function. While the callback is executed, other threads wait on an internal barrier. When the callback returns, waiting threads are released. In so far, one-time initializers are barriers in reverse.

Protocol: The protocol for one-time initializers can be modeled as a state machine with three states: *not initialized* → *initialization in progress* → *initialized.*

The first thread reaching the barrier changes the state from *not initialized* to *initialization in progress* and invokes the provided initialization callback. After

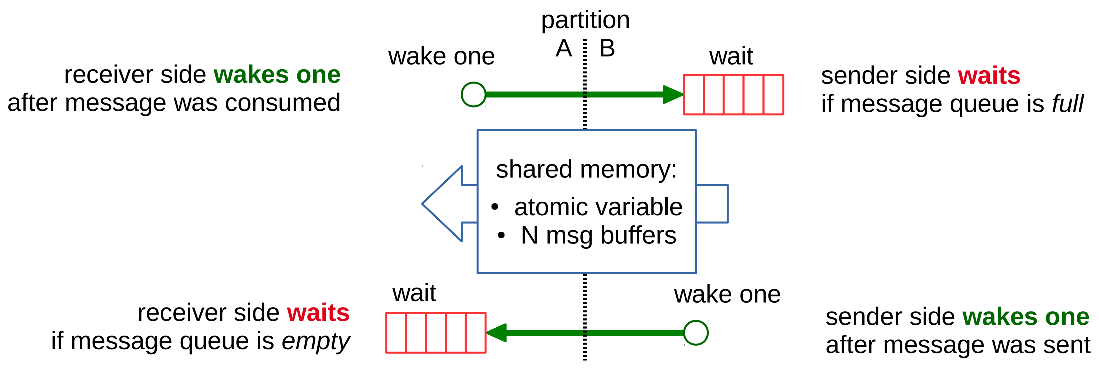

Figure 4.6: Queuing ports in AUTOBEST using static futexes [ZBL15].

the initialization completed, the first thread sets the state to *initialized* and wakes all waiting threads. Threads arriving while in state *initialization in progress* must wait on the barrier futex. Threads arriving in *initialized* state simply continue.

Implementation considerations: If the comparison in the `futex_wait` system call fails, the initialization must be already completed, so this is not to be considered a spurious wake-up.

Alternatively, one could model the *initialization in progress* state as two states to further distinguish pending waiters, but this would make the protocol more complex and would require atomic operations by the waiting threads to change the state.

Summary: The presented protocol using a state machine approach fits well to the futexes. Spurious wake-ups do not hurt here. All requirements are satisfied.

4.3.6 ARINC 653 Synchronization Mechanisms

The ARINC 653 blocking synchronization mechanisms listed in Section 2.2 comprise mutexes, counting semaphores, queuing ports, buffers, blackboards, and events. Mutexes and semaphores follow the designs described in Sections 4.3.1 and 4.3.3. Events and blackboards use protocols similar to the barrier protocol described in Section 4.3.4 that encode a waiting round and the current state of the event or if data is available in the blackboard. We only discuss the designs of queuing ports and buffers in the following.

Protocol: The design of queuing ports is specific to the static futex design in AUTOBEST described in Section 4.2.2 and requires that one futex value in user space controls *two* wait queues.

A queuing port comprises a shared memory segment containing a ring buffer and a 32-bit control word as futex value, and two static futex wait queues with different directions, as Figure 4.6 shows. The futex variable encodes the position of read and written buffers as well as two bits for full and empty conditions and is updated from both sides atomically using atomic CAS instructions.

The first static futex wait queue on the sender's side is used to wait on a *queue-full* condition, while the second static futex wait queue on the receiver's is used to wait if the queue is empty. On a send operation, the sending partition first checks for an available empty slot in the ring buffer, copies the message into the ring buffer, updates the control word, and wakes the first receiving thread waiting on the wait queue, if any. If there is no empty slot available in the ring buffer, the sender waits on its own wait queue for a free entry in the buffer. The receiver side follows the same protocol to wait for incoming messages.

This protocol works the same for ARINC 653 buffers. However, a normal futex can be used instead. Since both sending and receiving ends are in the same partition, and a buffer has threads waiting *either* on an empty buffer *or* on a full buffer, and not on both conditions at the same time, a single wait queue is sufficient.

Note that the presented protocol does not aim to improve efficiency, as it always uses a system call to wake up the other side.

Implementation considerations: An implementation must ensure atomicity of the message transfers for *RQ1 (correctness)*. Otherwise, a low priority thread could be preempted during a message transfer and subsequent operations on the queuing port or buffer could observe the unfinished message. If threads of a partition always share a single processor, like in AUTOBEST, an implementation could simply raise the calling thread's scheduling priority above all other threads during message copy operations. Otherwise, a blocking mutex must be used to serialize the copy operations.

Spurious wake-ups due to a failed *compare-equal* check in the kernel can lead to violations of the message ordering and therefore *RQ1 (correctness)*. The thread with the failed compare check can then in turn *steal* the message intended for the regularly woken up thread. This problem is similar to stealing described in Section 4.3.3 for the semaphore protocol, however the timeout problem is not applicable to ARINC 653 as all timeouts use absolute expiry times.

Summary: The presented design for queuing ports use a feature of static futexes to decouple the futex value in user space from a wait queue in the kernel. The design uses a shared ring-buffer with two wait queues. Buffers can use normal futexes with a similar ring-buffer structure.

4.4 Non-Preemptive Busy-Waiting Monitors

Recall the analyses in Sections 3.6, 3.7, and 3.8: We suggested to use a spinlock-protected critical section in user space instead of using an atomic operation like in futexes and also to manage the wait queue in user space. To prevent unwanted preemption of spinlock-based critical sections, we suggested to use a fast IPCP protocol. We further discussed that a thread cannot wait inside a spinlock-protected critical section, but must wait outside. Similarly, on the wake-up side, a thread should be able to wake up threads outside the critical sections as well. Lastly, with wait queues managed in user space, the wake-up operation should address waiting threads by their thread ID.

We now present a design for light-weight *non-preemptive busy-waiting monitors*. Its building blocks comprise a fast IPCP implementation as preemption control mechanism, which we introduce in Section 4.4.1. We then present a mechanism for *waiting in spin-based fine-grained locking* in Section 4.4.2. Lastly, we combine the different mechanism to become light-weight *monitors* in Section 4.4.3.

4.4.1 Efficient IPCP Protocols

In this section, we present two protocols for a fast implementation of IPCP in user space. The two protocols were first introduced in detail in a workshop paper [ZBK14] and discussed in later conference papers [ZBL15, Zue20] of the author.

Recall the analysis of preemption control mechanisms in Section 3.7. The protocol by Edler et al. for Symunix II [ELS88] discussed in Section 3.7.4 uses two variables shared between kernel and user space to temporarily disable preemption. The two variables follow the logic of an in-kernel implementation such as Linux as discussed in Section 3.7.2. Obviously, the Symunix II protocol just disables preemption and therefore does not integrate well into priority scheduling, but requires at most one system call in the nested case, regardless of the nesting level.

In contrast, the protocol by Almatary et al. [AAB15] discussed in Section 3.7.5 allows to change the priority in user space by using *three* shared variables. The first presented protocol UPRIO|KPRIO is a conceptual extension of the Symunix II protocol to change the priority instead of just disabling preemption. It just uses *two* protocol variables.

The UPRIO|KPRIO supports nested critical sections, but then shows the same pessimism as Almatary et al.'s protocol w.r.t. system calls in the nested case. Unlike the Symunix II protocol, these priority-based protocols require one extra system call for *each* nesting level in the worst case. The second presented protocol UPRIO|NPRIO improves on the nested case and requires a system call only when preemption is needed.

In the listings in the following sections, the protocol variables shared between user space and kernel are kept in the thread local storage of each thread to satisfy concerns about *RQ2 (robustness)*. The macro `SELF` provides an accessor to the specific variables of the current thread, as shown in Listing 4.1.

Listing 4.1: Shared variables of the UPRIO|KPRIO protocol

```
typedef struct {
    <...>
    prio_t uprio;
    prio_t kprio; // resp. nprio
    <...>
} thread_t;
#define SELF ((thread_t*)<...>)
```

The UPRIO|KPRIO Protocol

The UPRIO|KPRIO protocol uses two variables shared between user space and kernel. Both variables represent scheduling priorities. The first variable `uprio` is set in user space and reflects a thread's current scheduling priority in user space. The second variable `kprio` is the thread's priority in the kernel and is set by the kernel. The main idea of the protocol is that user space code temporarily increases `uprio` at the beginning of a critical section and reverts the change at the end of the critical section. Nested requests keep the previous priority on the caller's stack. During a scheduling event, when the kernel observes a change in `uprio`, it also updates `kprio` accordingly. If the user space code detects that the kernel has observed the priority change, the current thread calls into the kernel to preempt itself.

The user space part of the implementation in Listing 4.2 follows the basic sequence of the Symunix II protocol:

Listing 4.2: Fast IPCP in UPRIO|KPRIO

```
prio_t prio_raise(prio_t prio)
{
    prio_t prev = SELF->uprio;
    SELF->uprio = prio;
    return prev;
}

void prio_restore(prio_t prev)
{
    SELF->uprio = prev;
    if (SELF->uprio < SELF->kprio) {
        sys_preempt();
    }
}
```

The kernel part of the implementation in Listing 4.3 follows the steps already discussed in Section 3.7.5 for Almatary et al.'s protocol.

Listing 4.3: Kernel part of Uprio|Kprio

```
 1 kernel_sched_event(<...> new_thr)
 2 {
 3     ...
 4     prio_t uprio = min(SELF->uprio, max_prio);
 5     SELF->kprio = uprio;
 6     if (uprio < new_thr->prio) {
 7         // OK to preempt now
 8     }
 9     ...
10 }
```

The kernel first synchronizes its internal scheduling priority `kprio` on a scheduling event (lines 4 and 5), and preempts the current thread if necessary (lines 6 and 7). Here, `new_thr` refers to another thread becoming ready.

In this protocol, each scheduling event that happens during a critical section causes a synchronization of `kprio`, regardless of the priority of the new threads that become ready. This can lead to a superfluous system call at the end of a critical section if the new thread has a priority below the original priority of the thread, but this is required for *RQ1 (correctness)*. And for a nesting level of l critical sections, we might need up to l system calls in the worst case. This is the same as in Almatary et al.'s protocol. However, the Uprio|Kprio protocol has a simpler implementation. The presented protocol does not require atomic operations, just regular loads and stores to the protocol variables. Also note that the kernel bounds the user space priority to a maximum priority `max_prio`.

The Uprio|Nprio Protocol

Note that the Uprio|Kprio protocol presented in Section 4.4.1 is not optimal w.r.t. efficiency on the required system calls for preemption at the end of a critical section as the kernel synchronizes the elevated priority *unconditionally* on a scheduling event, without considering the priority of threads that became ready.

One technique to improve this is shown by Almatary et al.'s protocol discussed in Section 3.7.5. In the non-nested case, Almatary et al.'s protocol compares the *base priority* of the current thread to the priorities of other threads to decide whether to preempt the current thread at the end of its critical section. But this only works in the non-nested case, and not in the nested case when multiple intermediate priorities exist.

To solve this problem efficiently, we need to export relevant information on other threads that became ready in the mean time. In the Uprio|Nprio protocol, the second variable `kprio` is therefore replaced by `nprio`. In `nprio`, the kernel provides the priority of the *second highest priority thread* on the ready queue, i.e. the *next thread to be scheduled* if the current thread would block. This requires that the currently executing thread is *not* kept on the ready queue (recall *Benno scheduling* of Section 3.1.2).

The implementations of `prio_raise` and `prio_restore` in user space follow their counterparts of the Uprio|Kprio protocol, as Listing 4.4 shows:

Listing 4.4: Fast IPCP in Uprio|Nprio

```
prio_t prio_raise(prio_t prio)
{
    prio_t prev = SELF->uprio;
    SELF->uprio = prio;
    return prev;
}

void prio_restore(prio_t prev)
{
    SELF->uprio = prev;
    if (SELF->uprio < SELF->nprio) {
        sys_preempt();
    }
}
```

The key difference of the protocol is in the kernel part while handling scheduling events:

Listing 4.5: Kernel part of Uprio|Nprio

```
 1 kernel_sched_event(<...> new_thr)
 2 {
 3     ...
 4     ready_queue_add(new_thr);
 5     ...
 6     prio_t nprio = ready_queue_next()->prio;
 7     SELF->nprio = nprio;
 8     ...
 9     prio_t uprio = min(SELF->uprio, max_prio);
10     if (uprio < nprio) {
11         // OK to preempt now
12     }
13     ...
14 }
```

In Listing 4.5, the kernel puts the new thread `new_thr` on the ready queue (line 4) and then updates `nprio` to the priority of the highest priority thread on the ready queue (lines 6 and 7). If `uprio` is less than `nprio`, the current thread is preempted immediately (lines 9 to 12). Otherwise the same check in `prio_restore` takes care of preemption later.

The Uprio|Nprio protocol improves on the other protocols as it now reduces the system calls to preempt the current thread to the required minimum to still satisfy *RQ1 (correctness)*. The steps in the presented protocol relate exactly to the steps in a baseline implementation and system calls to change the scheduling priority of a thread, but with the semantic decision whether to preempt the current thread or not moved to user space. For *RQ2 (robustness)*, the kernel must additionally bound the user space priority to the maximum priority `max_prio`. Note that the protocol does not require atomic operations or any loops that could cause issues for *RQ3 (analyzability)*.

Lastly, the Uprio|Nprio protocol works well with nested priority changes or when waking up other threads inside the monitor approach discussed in Section 3.6.

4.4.2 Light-Weight Blocking for IPCP

We now present a light-weight blocking mechanism that provides the aforementioned properties and interacts properly with the fast IPCP implementation of Section 4.4.1.

The mechanism was first presented in a conference paper of the author [Zue20]. The description of the mechanism is also mostly taken from this paper [Zue20].

Non-preemptive critical sections: With the fast IPCP implementation presented in Section 4.4.1, we can now realize non-preemptive critical sections in user space.

Listing 4.6: Example non-preemptive critical section in user space

```
 1 spin_t example_lock;
 2
 3 void example_cs(<...>)
 4 {
 5     prio_t old_prio = prio_raise(max_prio);
 6     spin_lock(&example_lock);
 7     <...>
 8     spin_unlock(&example_lock);
 9     prio_restore(old_prio);
10 }
```

Listing 4.6 shows a simple example of such a non-preemptive critical section. A thread first raises its user space scheduling priority to `max_prio` to become non-preemptive, then acquires a spinlock. The thread's previous priority is kept in the local variable `old_prio`. At the end of the critical section (lines 8 and 9), the thread unlocks the spinlock and restores its previous scheduling priority. Note that this sequence does not need any system calls in the fast path. The system call to preempt the thread in `prio_restore` is only needed when in the meantime another thread with a priority higher than `old_prio` became ready.

Waiting: We now extend the critical section with a blocking mechanism that interacts properly with the fast IPCP implementation and the spinlock-protected critical sections. We also decide to manage the wait queue of blocked threads in user space. Our goal for efficiency is to reduce the number of system calls. Additionally, we address the requirements for predictability of Section 3.2.3.

We can make the following considerations for a waiting operation:

- Suspending the current thread needs help by the kernel. This requires a system call in any case.
- Waiting *after* the call to `prio_restore` could trigger unnecessary preemption, as the calling thread is going to suspend itself anyway. Waiting at `max_prio` is advisable.
- Waiting *inside* the spinlock-protected critical section causes problems, as other threads would be unable to acquire the spinlock. A system call to suspend a thread must then unlock the spinlock in user space from the kernel.
- Waiting *outside* the spinlock-protected critical section must prevent *missed wake-ups*, as the spinlock-protected critical section protects any internal state w.r.t. blocking.
- A thread's scheduling priority at wake-up time should reflect its original priority. When a thread is woken up at `max_prio`, it would execute only to the point where it restores its original priority and then causes unnecessary context switches if other medium priority threads are ready.
- Spurious wake-ups, e.g. timeouts, might require a second critical section *after* waiting to remove the current thread from the wait queue again. It is advisable when the thread is executing at `max_prio` after wake-up.

With these constraints, we can now design a waiting mechanism.

To prevent missed wake-ups, we use a *compare-and-block* mechanism in the kernel, similar to futexes or the wait indicator described in Section 3.8. Inside the spinlock-protected critical section, a thread evaluates internal state and decides to block. For this, it prepares a state variable. The state variable encodes a waiting condition and is changed by a wake-up operation. Then the thread unlocks the critical section in user space and calls the kernel to suspend. The kernel reads the state variable again and, if the current value matches the previous value, suspends the thread.

An alternative would be to call the kernel from inside the critical section and let the kernel unlock the critical section before suspending the calling thread. This would prevent any race conditions and ambiguity with parallel wake-up operations. However, the kernel would then need to know the exact semantics of the spinlocks in user space to unlock the spinlock for the caller. Also, this would tightly couple the user space critical section to a critical section in the kernel due to nested locking, which we identified as anti-pattern in Section 3.4.2.

We opt to unlock the spinlock in user space instead. This keeps the implementation in the kernel simple.

To address the problems of the scheduling priority at wake-up time, we temporarily *drop* the priority while waiting. While still executing at `max_prio` in user space, a thread calls the waiting system call to wait at a lower priority `wait_prio`. The kernel then temporarily sets the thread's priority to `wait_prio` while waiting. When the thread is woken up again, it will be enqueued at `wait_prio` on the ready queue. And when the thread is eventually scheduled, the kernel increases the scheduling priority back to `max_prio`, and then returns from the waiting system call. The obvious choice for `max_prio` is to use the thread's base priority, `old_prio`.

Note that we can easily achieve this by using the following trick in a fast IPCP implementation using the UPRIO|NPRIO protocol of Section 4.4.1. The kernel lets the thread wait at `wait_prio`, but leaves `uprio` unmodified while waiting. Note that `prio_raise` sets `uprio` to `max_prio` before waiting, so the thread will be effectively running at `max_prio` again after waiting.

Then the thread in user space can either lock the spinlock again, or leave the IPCP-protected critical section and restore its previous scheduling priority. As most likely no other threads with a higher priority will be ready at that moment, no system call will be needed.

Wake-up: For a wake-up operation, we can discuss similar considerations as for waiting:

- Waking up a waiting thread needs a system call to the kernel in any case.
- The wake-up system call must address a blocked thread directly by its thread ID (because the wait queue is kept in user space).
- Waking a thread up *after* calling `prio_restore` could cause unnecessary delays due to preemption of the current thread. Again, doing the wake-op operation at `max_prio` is advisable.
- The wake-up system call could happen *inside* the spinlock-protected critical section or be deferred *after* unlocking the spinlock. In the latter case, a system call to wake up a thread *outside* the critical section must prevent *spurious wake-ups.*

- Waking up a thread with a priority higher than oneself causes preemption when restoring the previous priority.

The wake-up mechanism addresses these constraints as follows.

We opt to wake up a thread *outside* the spinlock-protected critical section, but still running at `max_prio`. To prevent *spurious wake-ups*, we use same technique of a state variable as in the waiting operation. User space code passes the address and expected value of the state variable in user space to the system call, and the kernel compares the state variable to the expected value and only unblocks the thread on a match. This constitutes a *compare-and-unblock* mechanism and solves the spurious wake-up problems discussed in Section 3.8.

To prevent unnecessary system calls for preemption in `prio_restore`, we *fuse* the system call for wake-up with the system call for preemption. The resulting system call first wakes up a blocked thread by its thread ID, then unconditionally restores the calling thread's original priority `old_prio`, and eventually preempts the caller, if necessary.

System calls: We show the prototypes of the system calls:

Listing 4.7: Waiting and wake-up system calls

```
err_t sys_wait_at_prio(uint32_t *ustate, uint32_t cmp,
                        timeout_t timeout, prio_t wait_prio);

err_t sys_wake_set_prio(tid_t tid,
                         uint32_t *ustate, uint32_t cmp,
                         prio_t new_prio);
```

For the waiting mechanism `sys_wait_at_prio`, we follow the basic idea of the futex wait operation in Linux, i.e. the kernel suspends the calling thread with an optional timeout (`timeout`) if the content of a given state variable in user space (`ustate`) matches a compare value (`cmp`). As wake-up operations directly reference waiting threads by their thread IDs, the kernel can simply suspend the thread without using any internal wait queue. Additionally, the kernel *temporarily* changes the priority of the thread to `wait_prio` while waiting. Any variables of the UPRIO|NPRIO protocol in user space remain unchanged. When the waiting thread is woken up, it will be enqueued on the ready queue at `wait_prio`. However, when the thread is eventually scheduled, the priority of the `uprio` variable takes precedence.

The wake-up mechanism references a waiting thread by its thread ID (`tid`). A wake-up operation only succeeds if the address of the state variable in user space (`ustate`) matches the address for waiting, and if the current content of the state variable matches the compare value (`cmp`) of the system call. After performing the wake-up operation, the thread's priority is set to `new_prio`, and is possibly preempted. This time the protocol variables of the UPRIO|NPRIO protocol are updated.

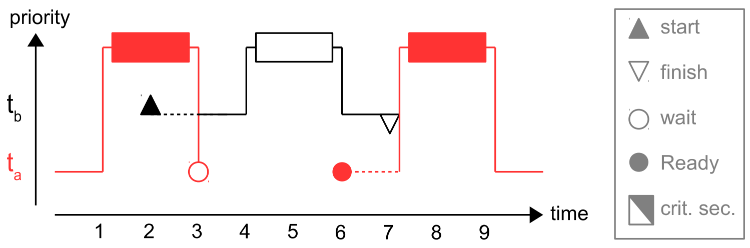

Figure 4.7: Example scenario of waiting and waking up in a critical section (CS) protected with IPCP. Inside the CS, thread t_a decides to wait, and effectively waits at its original scheduling priority after leaving the CS. Eventually, thread t_b enters the CS and wakes up t_a when leaving the CS. When t_a is scheduled again, it's priority is immediately raised again. Thread t_a then briefly enters the CS again and restores its original scheduling priority afterwards.

An implementation of the kernel primitives is shown in the paper [Zue20].

Interaction of blocking and wake-up with IPCP: Figure 4.7 shows an example of the interaction of the waiting and wake-up mechanism with IPCP for non-preemptive execution. The example comprises two threads t_a and t_b. Thread t_a performs a critical section with a wait operation, and thread t_b performs the corresponding wake-up operation.

At time $t = 1$, the thread t_a raises its user space scheduling priority (`uprio`) from its original scheduling priority to `max_prio` and then successfully locks the spinlock of the critical section. While t_a is inside the critical section, thread t_b becomes ready at time $t = 2$, but is not immediately scheduled due to the elevated priority of t_a. Inside the spinlock-protected critical section, thread t_a decides to wait and in turn prepares a state variable for waiting. At time $t = 3$, thread t_a unlocks the spinlock and calls `sys_wait_at_prio` with its original scheduling priority as `wait_prio` parameter. After successful comparison of the state variable, the kernel suspends thread t_a at `wait_prio`. As a result, thread t_a is effectively waiting at its original scheduling priority. But note that the `uprio` variable of t_a remains unchanged and still contains `max_prio`.

Because t_a suspends at $t = 3$, t_b can now run. At time $t = 4$, thread t_b raises its scheduling priority to `max_prio` and then successfully locks the spinlock. Inside the spinlock-protected critical section, the thread signals the waiting thread's state variable as a logical wake-up. This prevents a parallel state check by a waiting operation to succeed. Then, after unlocking the spinlock-protected critical section at time $t = 6$, thread t_b calls `sys_wake_set_prio` and passes the necessary data of the waiting thread (t_a in the example) and its original scheduling priority as parameters. The fused system call first wakes up the target thread t_a as described, and then restores the original scheduling priority of t_b. As t_a has a lower priority than t_b, thread t_b is not preempted and continues until it finishes at $t = 7$, The thread t_a is woken up at $t = 6$ and finally scheduled at $t = 7$. Once the thread is scheduled, its priority is immediately raised again, as the thread's `uprio` is still set to `max_prio`. Back in user space, t_a then locks the spinlock again. Since the management of waiting queues is performed in user space, a thread must check for spurious wake-ups (e.g. timeouts) which may require the thread to remove itself from the wait queue. At time $t = 9$, thread t_a releases the spinlock and restores its original scheduling priority without needing a system call for preemption.

Summary: The key technique for waiting is to move the wait system call outside the spinlock-protected critical section and couple the state of the critical section with a wait indicator. A temporary priority change to the original priority of the thread without further interfering with the IPCP protocol allows to preserve the original scheduling order of priority-based scheduling while waiting.

For the wake-up side, the key technique is to fuse the wake-up system call and the system call to synchronize the user space priority and preempt the current thread into a single system call. Unlike futexes, the presented mechanism allows to wake-up only one thread, which must be addressed by its thread ID.

The example shows that the IPCP mechanisms and the presented wait and wake-up mechanisms can fully mimic the behavior of non-preemptive execution inside an operating system kernel. The only difference is that user space cannot disable interrupts. The presented design follows the guidelines identified in the analysis in Section 3.4.2 and decouples the critical section in user space from any critical section in the kernel. Also, the techniques to reliably handle waiting and wake-up in decoupled critical sections follow the discussion in Section 3.8.

Regarding the requirements for predictability of Section 3.2.3, the overall design approach satisfies *RQ3 (analyzability)*. The technique to check a state variable in both the wait and wake-up operation helps *RQ1 (correctness)* and *RQ2 (robustness)*. The presented kernel mechanisms do not contain wait queues or atomic operations on user space variables which could interfere with *RQ4 (bounded operations on queues)* and *RQ5 (bounded loops and retries)*. The kernel does not need to modify any state variables in user space either. The only critical mechanism is the look-up of a thread by its thread ID in the kernel. When a global namespace is used, the mechanism could expose problems with *RQ6 (interference by shared namespaces)*.

4.4.3 Monitor Synthesis

The building blocks discussed in Section 4.4, namely preemption control by IPCP (Section 4.4.1), spinlocks, wait queues in user space, and the wait and wake-up primitives (Section 4.4.2), can now be used to construct a *non-preemptive busy-waiting monitor*.

Monitor operations: We say a thread *enters* a monitor to gain *exclusive access* to some internal state protected by the monitor. Within the monitor, a thread can decide to *wait* on condition variable or to *notify* waiting threads. Afterwards, a thread *leaves* the monitor again.

Waiting on a condition variable effectively comprises enqueuing the thread in a wait queue, leaving the monitor, blocking in the kernel, and entering the monitor again after waiting. When handling spurious wake-ups, e.g. timeouts, the operation must remove the thread from the wait queue.

For notification of the condition variable, a thread removes a blocked thread from the condition variable's wait queue and wakes up the blocked thread when leaving the monitor. Postponing the wake-up to the point where the current thread leaves the monitor is acceptable, as the woken up thread cannot re-enter the monitor before the spinlock is unlocked.

Wait indicator protocol: For waiting and wake-up in the monitor, we need a *state variable* for the wait and wake-up mechanisms of Section 4.4.2. The state variable acts as a *wait indicator*, similar to synchronization mechanisms inside a kernel (see Section 2.3). We will use a *dedicated* state variable (`ustate`) for each thread, and not one shared variable as in futexes. We keep the variable in each thread's TLS segment. The state variables must only be modified in the critical section protected by the monitor.

One important aspect is to handle the state variable containing the wait indicator correctly to prevent both missed wake-ups and spurious wake-ups. For the protocol, we draw from *eventcounts* and *sequencers* [RK79]. We will use `ustate` as a counter that is incremented *before* a thread suspends or is woken up. This prevents spurious wake-ups due to *ABA-problems* when two waiting operations follow each other back to back as described in Section 3.8.

Like a *sequencer*, the increments of a thread's `ustate` variable in user space order the particular wait and wake-up operations of the related thread. The waiting operation in the kernel follows *eventcounts*. As we use a dedicated per-thread counter, a wait operation will observe at most one additional increment from the corresponding wake-up operation. With this, the *compare-equal* condition for blocking in the kernel is sufficient to detect missed wake-ups. The increment before waking up a thread also follows *eventcounts*. As any further waiting attempt would increment `ustate` again, the *compare-equal* condition for unblocking in the kernel is sufficient to prevent spurious wake-ups.

Summary: We presented a design of a *non-preemptive busy-waiting monitor* in user space with *Mesa-style blocking condition variables* [BFC95] for systems with fixed-priority scheduling.

The monitor's building blocks comprise preemption control by IPCP (Section 4.4.1), spinlocks (Section 2.2.1), wait queues in user space, the wait and wake-up primitives (Section 4.4.2), and the wait indicator protocol discussed in this section.

The monitor operations combine sequences of common operations similar to the low-level building blocks of Section 3.4 to decouple the monitor critical sections from the system calls.

4.5 Higher-Level Synchronization Mechanisms based on Monitors

Based on the building blocks of non-preemptive busy-waiting monitors presented in Section 4.4, we now discuss the design of related higher-level synchronization mechanisms based on monitors.

We start with a discussion on the general design of monitor-based synchronization mechanisms and their data model in Section 4.5.1. We then present a design for blocking mutexes in Section 4.5.2 and related condition variables in Section 4.5.3. We also present low-level blocking primitives with a monitor interface in Section 4.5.4. The choice depends on the length of the critical sections. to protect the waiting condition, i.e. mutexes for long critical sections or spinlocks for short ones.

The description is partly taken from [Zue20].

4.5.1 Data Model of Monitor-Based Synchronization Mechanisms

We define a common data model for all further higher-level synchronization mechanism based on monitors.

Data model: In general, the *internal state* of a higher-level synchronization mechanism comprises: (i) a spinlock to serialize access to internal data, (ii) one or more wait queues, and (iii) further state information specific to the synchronization mechanism.

Additional data must be provided for each waiting thread. This data comprises at least one node element for the wait queues. Note that a thread can only wait on *one* wait queue at a time in the blocking synchronization mechanisms discussed in Section 2.2, therefore space for one wait queue node is sufficient and can be shared by all instances of the synchronization mechanisms in the designs presented in this thesis. This data can be kept in an arbitrary data segment specific to a thread, e.g. on the thread's stack or in thread-local storage (TLS). In the following description, we consider that the per-thread data is kept in TLS. This keeps the design simple.

As described in Section 4.4.3, we use a dedicated state variable (`ustate`) for each thread and also keep the variables in each thread's TLS segment. The state variables are also part of the internal state data of a monitor. State variables are only modified in the critical sections of the related synchronization objects in user space and only when a thread is waiting on a condition variable.

This comprises the full state data. Except for the spinlocks, atomic operations are not required, as all state data is protected by a spinlock.

As the wait queue is managed in user space, it must be able to handle an application-specific number of threads, and not an arbitrary number of threads. Therefore, an application can use a data structure that suits the application requirements best, e.g. doubly-linked lists for a smaller number of threads, or binary search trees for a greater number of threads.

Summary: The presented data model follows the blueprint of typical in-kernel synchronization mechanisms as discussed in Section 3.4.

When looking at the requirements for predictability of Section 3.2.3, avoiding atomic operations especially addresses concerns of *RQ3 (analyzability)* and *RQ5 (bounded loops and retries)*. The responsibility to manage a wait queue in user space also moves any complexity from the kernel to the user space. Especially *RQ2 (robustness)* and *RQ4 (bounded operations on queues)* now depend on application-specific bounds and behavior. And the protocol for the wait indicator ensures *RQ1 (correctness)*.

4.5.2 Blocking Mutexes

The presented blocking mutexes follow the described interface in Section 2.2.2. We use the monitor operations presented in Section 4.4.3 and the data model of Section 4.5.1 to construct blocking mutexes as higher interface from monitors.

Protocol: The data representing a blocking mutex comprises an internal spinlock, the thread ID of the current lock owner, and one priority-ordered wait queue.

A `mutex_lock` operation enters the monitor and checks the current owner of the mutex. If the mutex is currently free, it makes itself the new owner and leaves the monitor. Otherwise, `mutex_lock` adds the calling thread to the wait queue and waits with a given timeout. On wake-up, the thread is either the new mutex owner or not. If not, `mutex_lock` removes the thread from the wait queue before leaving the monitor.

A `mutex_unlock` operation enters the monitor and checks the wait queue. If the wait queue is empty, it marks the mutex as free and leaves the monitor. Otherwise, it removes the first waiting thread from the wait queue, put the new thread's ID as mutex owner, and leaves the monitor with an according wake-up operation.

Note that wake-ups due to failed *compare-equal* checks of the state variable always mean that the mutex value was passed to the woken-up thread. In this case, the design prevented a missed wake-up, and the thread can immediately continue.

Summary: The presented mutex design shows that blocking mutexes follow the basic blueprint given in Section 4.4.3 and in Section 4.5.1. An implementation does not need to handle any unintended wake-ups, nor does it need to handle any loops critical for WCET analysis.

In the best case, locking and unlocking a mutex takes no system call. On contention, `mutex_lock` needs one or two system calls (the second one to handle a pending preemption request). `mutex_unlock` always needs one system call.

4.5.3 Condition Variables for Blocking Mutexes

We now discuss condition variables for the mutexes presented in Section 4.5.2. Condition variables were first discussed in Section 2.2.3.

Condition variables are wait queue objects, so the internal state of a condition variable synchronization object comprises a wait queue.

However, for internal synchronization, we also require a spinlock to keep the wait queue consistent. We use the spinlock of the support mutex to protect the wait queue as well. One possible way to obtain the spinlock of the support mutex is to change the API of the notification functions so that the support mutex is always passed as additional parameter. Another possibility is that the condition variable object includes a reference to the support mutex. The first call to `cond_wait` registers the support mutex in the condition variable object. Note that this requires explicit ordering of memory accesses and memory barriers in the notification operations to properly detect an uninitialized wait queue.

Protocol: A `cond_wait` operation enters the monitor of the mutex, enqueues the calling thread on the wait queue of the condition variable, unlocks the mutex, wakes up any new mutex owner, and then waits with the given timeout. After wake-up, a thread can be in three different states:

- The thread is the owner of the mutex. It was successfully signaled and requeued to the mutex. In this case, `cond_wait` leaves immediately.
- The thread is still enqueued on the condition variable wait queue. This happens when the timeout expires. In this case, `cond_wait` removes the thread, inserts it on the mutex wait queue, increments the wait indicator, and then waits on the mutex.
- The thread is already enqueued on the mutex wait queue. This happens when the timeout expires *after* the thread was notified. The notification

operation never tells the kernel to clear the timeout. This can also happen when the *compare-equal* check in the kernel fails and the thread is requeued in parallel. In this case, `cond_wait` simply waits on the mutex.

To evaluate the difference between the last two states, `cond_wait` can check if the wait indicator was incremented. In any case, on return of final waiting, the thread must be the new mutex owner.

Both `cond_signal` and `cond_broadcast` need to know the spinlock of the support mutex. They enter the monitor and simply requeue the given number of threads (one or all) from the wait queue of the condition variable to the wait queue of the mutex. For each requeued thread, they must also increment the user state variable. To support "naked notifies" where the support mutex is currently unlocked, both notification functions check the current state of the mutex after requeuing. If the mutex is free, they make the first waiting thread the mutex owner and wake up the thread when leaving the monitor.

Summary: The protocol design is complex. However, similar complexity is required for an implementation in a kernel with fine-grained locking.

The implementation does not require any loops except for `cond_broadcast`. The loop is bounded to an application-specific maximum number of waiting threads for *RQ5 (bounded loops and retries)*. Also, as the loop is non-preemptive and protected by the spinlock, other threads cannot observe transient states, therefore this satisfies *RQ9 (hidden transience)*.

The design shows a small imperfection. Unlike `futex_requeue`, notification of a condition variable does not require a system call and thus does not clear any internal timeouts of the thread waiting on the condition variable. The implementation must handle an unintended wake-up when a timeout expires and re-apply for the mutex with an extra system call. Note that this does not change a thread's position on the mutex wait queue, so this has no impact on *RQ1 (correctness)*.

In the best case, both `cond_signal` and `cond_broadcast` require no system call. In the worst case, both functions require at most one system call, either for preemption, or to wake up a new mutex owner.

In contrast, `cond_wait` requires from one to four system calls: one system call to wake-up the next mutex owner, one system call to wait on the condition variable (mandatory), one system call to wait on the mutex after a spurious wake-up, and one system call for preemption when leaving the monitor.

4.5.4 Low-Level Monitor API

We now present another type of condition variables that interact directly with the non-preemptive critical sections of the internal spinlocks instead of the blocking mutexes presented in Section 4.5.2. Here, the monitor is the higher-level synchronization mechanism and provides mutual exclusion by busy-waiting and blocking condition variables.

The main difference to the mutex-based condition variables of Section 4.5.3 is that "naked notifies" are not needed here. We consider this to be an alternative to semaphores for low-level notification in situations where blocking synchronization mechanisms cannot be used, for example from user space interrupt handlers.

For the interface, we deviate from the standard condition variable interface presented in Section 2.2.3. Instead, we use a *monitor* interface comprising *enter*, *leave*, *wait*, *notify*, and *notify all* operations, as described in Section 4.4.3. Like in Java, the design uses a single wait queue embedded in the monitor. The design can be extended to support multiple wait queues.

Protocol: A monitor object comprises an internal spinlock and an internal wait queue. For efficiency when notifying a thread, we defer the actual wake-up operation to the point when leaving the critical section. For this, we keep a pointer to the thread with a pending wake-up in the monitor object.

An *enter* operation simply locks the internal spinlock. When multiple threads try to enter the monitor, they are serialized by the internal spinlock of the monitor. This relates to a FIFO-ordered *enter queue*.

A *leave* operation checks for a pending wake-up and wakes up the thread when leaving the monitor.

A *wait* operation unlocks the monitor, wakes up any pending thread, and waits with the given timeout. After wake-up, the operation locks the spinlock again and evaluates the thread's wait indicator. The thread can be in two states:

- The wait indicator is unchanged. The wake-up was spurious, e.g. the timeout expired. In this case, the thread is still enqueued on the wait queue. The *wait* operation removes the thread from the wait queue.

- The wait indicator was incremented. The thread was woken up, or the *compare-equal* check in the kernel failed before waiting. In both cases, the thread was properly notified.

The first *notify* operations registers the thread to wake up for later. This optimization works well until a second *notify* operation is needed, or a *notify all* operation processes a second thread. In this case, the operations must wake-up the previously pending thread first, but can keep the next thread to process as pending for later.

To support more than one wait queue, the wait queues can be moved into dedicated wait queue data objects. Then the interfaces of the *wait* and *notify* operations must be changed to work on the given wait queue objects.

Summary: The design is less complex than the mutex-based condition variables of Section 4.5.3, because requeue operations to the monitor wait queue are not needed here. This also simplifies the handling of wake-ups by timeouts. For the rest, a similar discussion as for the mutex-based condition variables applies here. The implementation does not require loops except for the *notify all* operation.

The presented design is optimized to perform an uncontended *enter* → *leave* sequence without system calls, and both *wait* and *notify* with one system call in the best case. It requires at most one system call for a *enter* → *leave* sequence, at most two for *enter* → *wait* → *leave*, and exactly one for *enter* → *notify* → *leave*. Note that more than one thread can be woken up, but this requires one additional system call for each thread. Also, the additional system calls happen from within the internal critical section, pulling the pessimism of the system call into the critical section. The design is optimized for the wake-up of only one thread.

Chapter 5

Evaluation

This chapter evaluates the designs presented in Chapter 4. For the evaluation, we discuss two different aspects following the metrics defined in Section 3.2.4. The first aspect is to evaluate the performance benefits of the presented designs in benchmarks. This shows that the designs are efficient. The second aspect is to discuss the impact of the designs on a WCET analysis. This evaluates the designs analytically and shows predictability.

We evaluate the designs on two different levels, i.e. kernel mechanisms and the resulting higher-level mechanisms. In both cases, we provide reasonable baselines to compare to. For the blocking kernel mechanisms presented in Chapter 4, i.e. deterministic futexes, static futexes, and the low-level wait and wake-up primitives used by monitors, we provide a comparison to a hash-based futex design similar to Linux. For the higher-level synchronization mechanisms presented in Section 2.2, we discuss analytical aspects only for mutexes and condition variables, as they cover the two main use cases of blocking synchronization mechanisms, mutual exclusion and event notification. Here, we additionally include a system-call-based implementation of mutexes and condition variables as baseline. Also, we compare the presented fast IPCP design to a pure system-call-based approach to change the scheduling priority.

For the performance evaluation, we use *microbenchmarks* to evaluate the fast paths of the approaches. As this would leave open how much real applications would benefit from futexes and monitors, we evaluate different implementations of mutexes under various degrees of contention in a research kernel. As a target hardware for benchmarking, we selected different ARM processors. In particular, we compare the performance on a *Freescale i.MX6Q* processor with four 32-bit ARM *Cortex A9* cores. Unlike x86 processors by Intel and AMD, the 32-bit ARM architecture provides simpler designs that target the lower end of the power usage and performance spectrum. However, due to differences in the hardware architecture, the system call overhead is not so extreme as compared to Intel processors as discussed in Section 3.2.2.

For the analytical evaluation, we cannot compare the *exact* WCET of the different mechanisms unless we have a specific application and with known upper bounds for a specific hardware platform and then feed these results into a WCET analysis. For this thesis, we want to have a comparison on an *abstract level*, so we define a *reference architecture* for the comparison for which we evaluate all the different synchronization mechanisms. The reference architecture comprises a kernel that hides the complexity that is not relevant for comparison, such as a scheduler implementation or the overheads of system calls and context switches, as these subsystems and building blocks of the kernel provide the same costs for all evaluated designs. The reference kernel uses fine-grained locking, however with a reduced complexity compared to Linux where the additional complexity is not necessary to compare the mechanisms. We think this is the right level for a system designer to assess the presented designs, as the particular building blocks behave similarly and the complexity of the remaining system is not necessarily relevant for an evaluation or impacted by these results.

We start with the evaluation of the efficiency of the designs. We first validate our assumptions on the specific costs of elementary synchronization and OS overheads for a range of ARM processors in Section 5.1. Then we show benchmark results for a selected set of the mechanisms in Section 5.2. For the evaluation on the analytical level, we present a worst-case timing model and discuss the costs of the designs in Section 5.3. We summarize the evaluation results in Section 5.4.

5.1 Validating Assumptions

We start the evaluation with a brief introduction to architectural overheads of 32-bit ARM processors. We first check if the relation of the costs claimed in Section 3.2.2, namely that ALU operations are much faster than atomic operations, and that atomic operations are much faster than system calls, still holds for the ARM architecture.

As testbed, we use a research RTOS named *Marron*. Marron provides static partitioning of OS resources with fixed-priority scheduling on each processor core. Marron further supports multiple address spaces (partitions), threads, and user-level interrupt handlers. The Marron kernel provides various implementations of thread synchronization mechanisms for system's research, but lacks lots of other useful features. The general architecture of the synchronization mechanisms follows the composition of Table 3.1 in Section 3.6. However, the kernel is designed for minimalism and has low overheads (instruction-wise), so an interaction with the kernel (e.g. a system call) only contains what a typical operating system kernel implementation needs to do in any case, but not more. Therefore, the results probably underreport the OS overhead compared to other operating systems.

Marron currently supports only 32-bit ARM platforms, so we evaluated the approaches on three system-on-a-chip platforms. The *BeagleBone Black* provides a single Cortex A8 core running at 550 MHz, the *Freescale i.MX6Q SABRE Lite* has four Cortex A9 cores at 792 MHz each, and the *BeagleBoard-X15* has two Cortex A15 cores at 1 GHz. Note that the Cortex A8, A9, and A15 cores have different microarchitectures, with the A8 being a 13-stage in-order pipeline design, while the A9 and the A15 are out-of-order designs, with a shorter 8-stage pipeline on the A8 and a longer 15-stage pipeline on the A15.

In a set of benchmarks, we evaluate microarchitectural overheads, like function call overheads, memory barriers, and atomic operations, and OS overheads, like system calls and context switching. The working set of the benchmark is small and fits into both instruction and data caches. Except for one test, the benchmarks do not need to access any external DRAM. Also, we only run our benchmarks on a single processor core to exclude contention and interference by the other processor cores. For better comparison, we have normalized the results to clock cycles of each CPU core. Measurements were taken with the internal cycle counter of the CPU cores. We first take the time, run each benchmark in a loop 1024 times, then take the time again and divide the result, therefore all results include the loop overhead. These results shall provide a rough reference point for later discussions and do not aim to be precise assessment of the architectures. Table 5.1 shows the results.

The first six tests of group "a" focus on ALU and load/store performance. Tests a1 and a2 evaluate the operations of the benchmark loop and show that the overhead is negligible. Tests a3 and a4 show the overhead of function calls. Test a3 benchmarks the costs of changes in the control flow. The registers in test a4 are often saved and restored in function prologues and epilogues. Also, the system call code comprises such a sequence. Each processor architecture can load

Table 5.1: Best-case overhead measurements of lowest-level building blocks on Marron for different 32-bit ARM platforms. Results in CPU cycles represent the average of 1024 runs.

Test		**Cortex A8**	**Cortex A9**	**Cortex A15**
		13 stages	8 stages	15 stages
		in-order	out-of-order	out-of-order
a1	empty loop	2	1	1
a2	read time stamp counter	4	4	8
a3	function call	7	9	8
a4	push/pop pair (registers r0 – r12)	15	14	16
a5	read thread ID in helper function	8	10	6
a6	cold-cache memory read	158	150	127
b1	acquire and release barriers	33	6	20
b2	CAS without barriers	8	22	30
b3	atomic increment/decrement pair	11	43	51
b4	uncontended spin_lock/unlock pair	74	38	108
b5	futex mutex fast path	75	54	102
c1	null system call	174	106	205
c2	`sys_preempt`	426	281	435
c3	`sys_yield`	507	323	521
c4	context switch A → B → A	1196	891	1212
c5	wake up thread on other processor core	—	449	674

or store two 32-bit registers in each cycle, according to the manuals and test a4. Test a5 reads the thread ID in a helper function. As the results of tests a3 and a5 on the Cortex A15 show, the tests are not always accurate and may include other side effects, such as interrupt handling or internal side effects of the pipeline. Test a6 shows the overhead of a cold-cache memory read to a non-dirty cache line.

The next five tests of group "b" evaluate the overhead of atomic operations. The ARM architecture uses a weakly-ordered memory model and requires explicit memory barriers to order memory accesses. We find these barriers in low-level synchronization primitives. Test b1 shows the overhead of such a barrier. The next two tests for CAS (b2) and atomic increment/decrement (b3) comprise atomic operations based on LL/SC primitives. CAS comprises one sequence. The increment/decrement pair comprises two. In test b4, the uncontended pair of spinlock operations uses a fair ticket spinlock implementation and comprises one atomic operation and two barriers. The last test b5 simulates the futex mutex fast path and comprises two CAS operations and two barriers.

The Cortex A8 is a single core design. Atomic operations are fast due to the lack of cache coherency overhead, but the memory ordering barriers are quite expensive. On the Cortex A9, which is a multicore design, barriers are inexpensive compared to CAS operations. However, on the faster Cortex A15 design, barriers are much more expensive than on the Cortex A9 due to the longer pipeline.

The last five tests of group "c" evaluate the kernel overhead. In test c1, a *null system call* measures the minimal overhead of a system call. In this benchmark, no internal lock in the kernel is taken. The other tests increase the interaction with the kernel. `sys_preempt` (c2) takes one lock in the kernel. `sys_yield` (c3) takes two locks. Test c4 shows a round trip of two system calls and two context switches. The last test c5 shows the overhead to wake up a thread on a second processor core.

All in all, we can see that atomic operations are roughly a magnitude more expensive than ALU operations. However, this is not true for system calls. The minimal overhead for system calls is only between two and two-and-a-half more than atomic operations. But we also see that the overheads for fine-grained locking in the kernel can quickly add up, as the Cortex A15 shows.

With this, the assumptions of Section 3.2.2 also hold for ARM, and it's worth to invest in a fast path that does not use system calls.

Also, the cold-cache memory accesses are two orders of magnitude more expensive than hot-cache accesses. This underlines the choice to focus on cold-cache memory accesses as metric in Section 3.2.4.

5.2 Performance Measurements

For the evaluation of the performance benefits of the designs of Chapter 4, we start with the specifics of deterministic futexes in Section 5.2.1. We then evaluate the efficient IPCP protocol in Section 5.2.2. Lastly we compare the benefits of futexes and monitors compared to a baseline design using system calls in a scenario with various degrees of contention on a mutex in Section 5.2.3.

5.2.1 Specifics of Deterministic Futexes

We now evaluate the specifics of the deterministic futex design presented in Sections 4.1. We focus on the specific design choices in the kernel, in particular the two nested BSTs, which are the shared global or private process-specific *address trees* and dedicated per-futex *wait queues*, and the resulting global lock for shared futexes.

We evaluate the overhead of using binary search trees on the average-case execution time (ACET) in a first experiment. As the global address tree is shared among processes to realize shared futexes, we also evaluate the interference of concurrent futex operations in another experiment.

We evaluate the design in the context of PikeOS. The implementation in PikeOS uses AVL trees as BSTs (see Section 4.1.2). The measurements were performed on a *Freescale i.MX6Q SABRE Lite* board with four Cortex A9 cores using PikeOS 5.0.2. The particular results are not comparable to results on Marron, but the processor-specific overheads presented in Section 5.1 still apply.

The presented results are taken from a conference paper by the author [ZK19].

Overhead of Binary Search Trees

To evaluate the overhead of using two nested BSTs in the presented design, we conduct the following experiment. We measure the execution times of `futex_wake` operations to wake one thread each time from a set with a variable number of blocked threads. We use two different tests:

- Test A: all threads wait on the same futex
- Test B: all threads wait on different futexes

Both tests measure the overhead of the BST for the wait queue and the address tree in isolation. All futexes are process-private to exclude interference by other processes.

Figure 5.1 shows the results of tests A and B. Both tests show logarithmically increasing execution times, as the number of nodes in the specific BSTs increases linearly during the test. The spikes at multiples of power-of-two numbers of threads are caused by rebalancing the AVL tree. Outliers are caused by other system activities, such as interrupt handling.

We can also observe that the timing quickly hits worst-case cache behavior, as tree nodes are kept at the same offset within page-aligned thread control blocks (TCBs), effectively causing trashing in the same cache set. Recall from Table 5.1

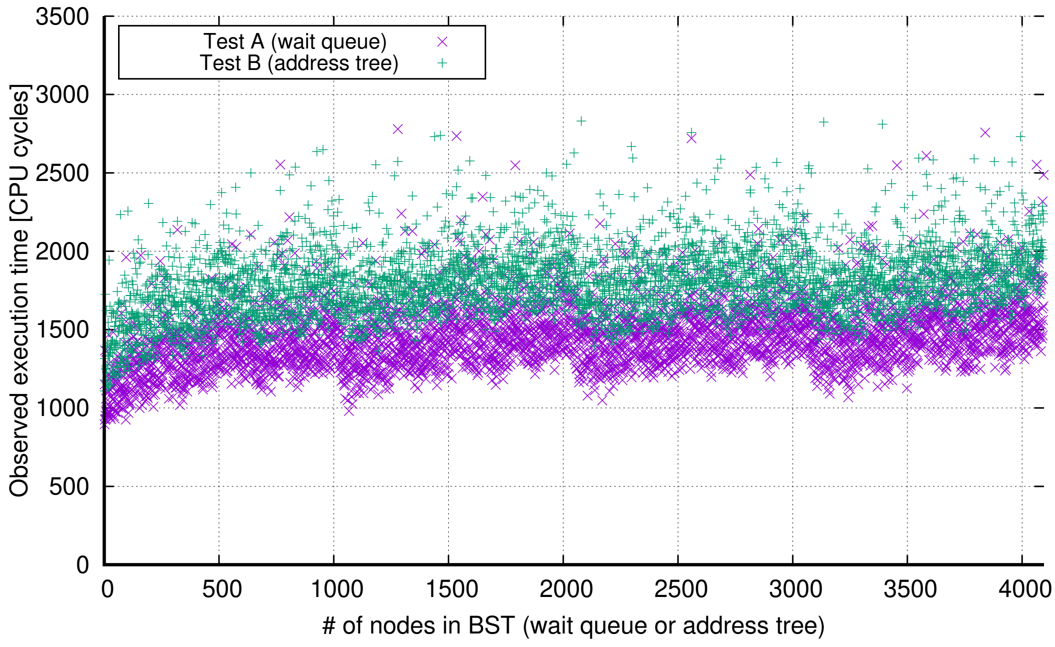

Figure 5.1: Observed execution times of `futex_wake` operations to wake one thread for a variable number of blocked threads on PikeOS on Cortex A9. All threads are either waiting on the same futex (test A) or on different futexes (test B). Test A shows the overhead of the BST to manage blocked threads in a wait queue, while test B shows the overhead of the BST during wait queue look-up.

---

that a cold-cache memory read takes 150 cycles on this platform. Note that this problem applies to linked list traversal of similar structures as well, and also affects other operating systems such as Linux, see Figure 3.6 in Section 3.5.3. The worst cases of the isolated BST operations need to be added to get the overall impact on the ACET when both BSTs are fully populated.

In comparison with Linux, where similar operations typically need constant time when there are no collisions in the hash table, the BST approach shows a quickly increasing overhead when the number of nodes is small, but the overall overhead is acceptable (min / max / avg in CPU cycles: test A: 896 / 2779 / 1461, test B: 1108 / 2831 / 1800).

Interference of Concurrent Futex Operations

To evaluate the interference of concurrent futex operations, we conduct another experiment, where a thread in process α performs and benchmarks a `futex_requeue` operation to requeue one thread 512 times, while a thread in process β requeues 512 threads in parallel on another processor each time. First, we let process α run in isolation (tests A and B), then execute both processes α and β concurrently

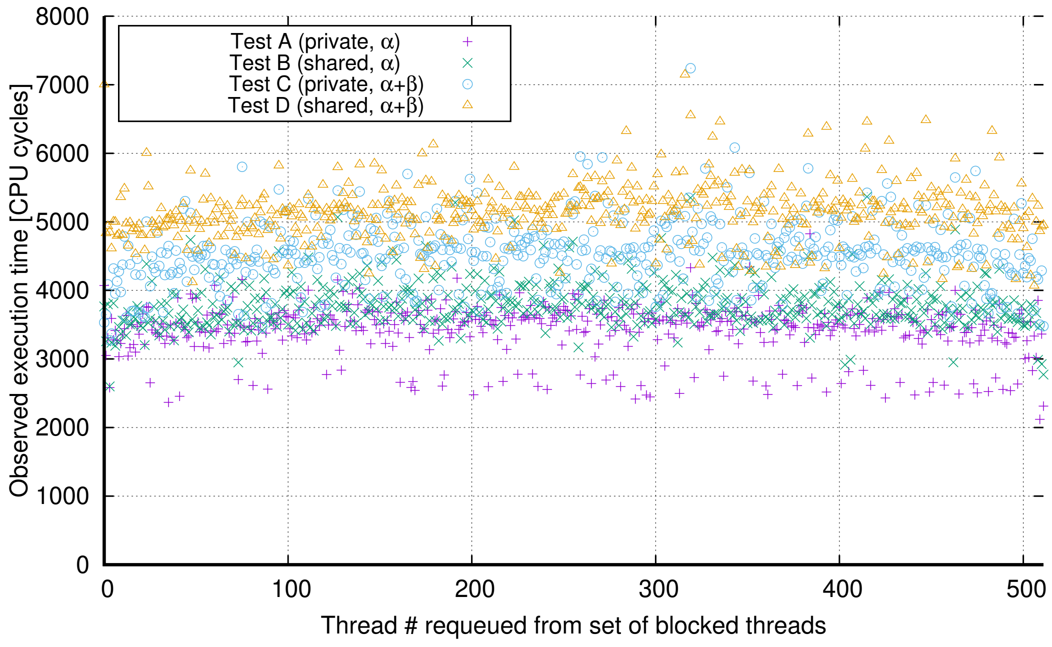

Figure 5.2: Observed execution times of 512 `futex_requeue` operations to requeue one thread on 512 blocked threads on PikeOS on Cortex A9. The four tests show different combinations of private (A and C) and shared (B and D) futexes. Additionally, in tests A and B, only process α runs on a single processor core. In tests C and D, a second process β requeues 512 threads in parallel on another processor core and causes interference.

(tests C and D). Also, we measure the difference of private (tests A and C) and shared (tests B and D) futexes.

Figure 5.2 shows the results. We see that shared futexes are more expensive than private futexes, as the kernel needs to retrieve the physical address of the underlying futex from the page tables. Also, the measured execution time is higher when both process α and β run in parallel. The preemptive design in the kernel serializes all internal operations on shared futexes with a fair ticket spinlock. The observed interference in test C with private futexes is additionally caused by *shared hardware* in the memory hierarchy, e.g. the L2 cache and the memory controller.

Lastly, Figure 5.2 shows arc-shaped execution times for requeue operations that reach a maximum at $2 \cdot \log \frac{n}{2}$ when both source and destination BSTs of a requeue operation are equally populated.

5.2.2 IPCP Performance

We briefly evaluate the performance benefits of using the UPRIO|NPRIO protocol for IPCP presented in Section 4.4.1. For the performance evaluation, we compare the UPRIO|NPRIO protocol to a baseline version using a pair of `sys_prio_set` system calls to implement IPCP. Both mechanisms are implemented in the Marron kernel introduced in Section 5.1. The benchmark comprises a best-case scenario where the working set fits into the caches and is not interrupted otherwise.

As IPCP is used frequently, the baseline implementation also uses *Benno scheduling* (Section 3.1.2) as optimization and operates on in-kernel equivalents of the protocol variables of the UPRIO|NPRIO protocol of Section 4.4.1.

Listing 5.1: System call to change the scheduling priority of the calling thread

```
1 prio_t sys_prio_set(prio_t new_prio)
2 {
3     prio_t prev_prio = SELF->prio;
4     SELF->prio = min(new_prio, max_prio);
5     if (SELF->prio < CURRENT_CPU->next_prio) {
6         sched_preempt(SELF);
7     }
8     return prev_prio;
9 }
```

The different steps in the implementation shown in Listing 5.1 comprise validating the given priority (line 4), updating an in-kernel priority of the current thread (also line 4), and comparing the priority to the priority of the next thread on the ready queue (line 5) to decide whether to preempt the current thread (line 6).

Table 5.2 shows the results of a temporary elevation of the scheduling priority of the current thread, comprising a `prio_raise` and a `prio_restore` operation. When no other thread is woken up in the mean time, the overhead of the UPRIO|NPRIO protocol is a little bit higher than a pair of function calls with TLS access (see Table 5.1), as the first test in the upper part of Table 5.2 shows. The second test enforces a pending preemption request. For this, the test overwrites the `nprio` variable with a high value. The `prio_restore` operation in turn invokes

Table 5.2: Comparision of the UPRIO|NPRIO protocol to a baseline version using system calls to change the scheduling priority on Marron for different 32-bit ARM platforms. Results in CPU cycles represent the average of 1024 runs.

Test	**Cortex A8**	**Cortex A9**	**Cortex A15**
UPRIO|NPRIO protocol			
- `sys_prio_raise`/`sys_prio_restore` pair	29	27	14
- `sys_prio_raise`/`sys_prio_restore` pair with `sys_preempt`	466	422	455
- wake-up of medium priority thread in IPCP	1242	932	1295
baseline (two system calls)			
- pair of two `sys_prio_set` system calls	419	276	468
- wake-up of medium priority thread in IPCP	1686	1213	1790

`sys_preempt` to let the kernel preempt the current thread. In this case, the execution time is significantly higher. The system call for preemption is expensive, as the kernel locks scheduling data internally. However, this is a penalty that is only to take when a now higher priority thread became ready.

The baseline using two full system calls to change the scheduling priority is much slower than the UPRIO|NPRIO protocol without preemption. However, we can observe that the overall execution time of the two system calls is *lower* than the fast implementation with preemption. This is a side effect of the test, as we explain in the following.

Note that in the baseline case, the `sys_prio_set` system call is quite fast and has no locking overhead, as the comparison to a null system call (see Table 5.1) shows. The Marron kernel implements a similar optimization as the UPRIO|NPRIO protocol to check `nprio` at kernel level without needing any locks before further preempting the thread (and then acquiring additional locks). In contrast, the `sys_preempt` system call expects only to be invoked when preemption is pending and acquires internal locks unconditionally. In so far, the comparison is a bit unfair, as the costs for preemption are missing in the baseline case. Unfortunately, the test cannot simply overwrite `nprio` to a higher value in the kernel to enforce a similar scheduling overhead.

For a fair comparison, we conducted another test that includes preemption. Here, a first thread temporarily raises its scheduling priority, wakes up a second thread at a medium priority, and lowers its priority again. In turn, the kernel preempts the first thread and switches to the second thread. The second thread immediately waits again, so we observe another context switch back to the test thread. But also in this scenario, the UPRIO|NPRIO protocol improves the performance over the baseline scenario. For reference, the overhead of the two context switches can be taken from Table 5.1.

5.2.3 Comparison for Different Mutex Implementations

We now present a performance evaluation of mutexes based on the different alternative design approaches for synchronization mechanisms as discussed in Section 3.6, i.e. a baseline using system calls, futexes, and monitors. For this, we implemented the synchronization mechanisms in the Marron kernel introduced in Section 5.1.

At first, we briefly discuss the implementation of the different synchronization mechanisms. We then compare the mutex performance with and without contention. Lastly, we compare the performance of the different designs under various degrees of contention to evaluate the impact for a real world application.

Parts of the evaluation discussed in this section were already presented in previous work of the author [Zue20].

Synchronization Mechanisms in Marron

We now discuss an implementation of the different alternative design approaches for synchronization mechanisms using similar building blocks in Marron. The implementation follows the structure of Table 3.1 in Section 3.6 and uses fine-grained locking.

The baseline approach provides mutexes and condition variables by dedicated system calls. Here, all wait queues are statically allocated at boot time and accessed via an index-based design, similar to the concept of static futexes of Section 4.2. However, the blocking API exposes no *compare* primitive, but implements mutex or condition variable semantics directly in the kernel. On notification, the condition variable implementation internally requeues threads from the condition variable wait queue to the mutex wait queue. Internal locking uses a per-process lock.

The futex implementation follows a hash-based wait queue design like in Linux (Section 3.5.1) with a specialized API for mutexes (`futex_lock`, `futex_unlock`) and a general-purpose API for the rest (`futex_wait`, `futex_wake`). The `futex_requeue` call supports only requeuing from condition variables to mutexes. Internally, the kernel implements a single shared wait queue for all futexes, i.e. a hashed wait queue of size 1, which is sufficient for testing the overhead of the kernel interface, but would not scale well in a real-world system. Futex wait queues are process-private and protected by a specific per-process lock.

The monitor approach implements wait queues in user space. The wait queues are protected by non-preemptive ticket spinlocks following the UPRIO|NPRIO protocol of Sections 4.4.1 and 5.2.2. The wait and wake-up primitives follow the description in Section 4.4.2.

Note that in all three cases, wait queues are implemented as priority-ordered linked lists. However, the focus here is not to stress the wait queue implementation, but to assess the overhead of the kernel interfaces, therefore the wait queue contains at most one thread. A more efficient wait queue implementation could use the *plist* approach of Linux described in Section 3.5.2, or use a BST. For the same reason, we also did not implement futexes using the deterministic futex design of Section 4.1, but used the simplified design presented above.

Comparison of Mutexes Performance

We perform an experiment to evaluate the performance of both the fast path and the slow path of a pair of `mutex_lock` and `mutex_unlock` operations in different contention scenarios and using different designs based on system calls, futexes, and monitors. Table 5.3 shows the results of uncontended mutex operations, contention on the same core, and contention by other cores for the three different ARM processors introduced in Section 5.1.

The overhead of the uncontended mutex runs show stable results, as these tests run in a single-threaded context. The working set of the tests is small and fits into instruction and data caches.

We determine the results for the contended case on the same processor core with the help of a second thread. The presented results show two contended and two uncontended lock/unlock operations, one *resume other thread* and one *suspend self* operation, and four context switches:

```
thread A:  sys_mutex_lock(&m); sys_thread_resume(B);     // A preempted
thread B:  sys_mutex_lock(&m);                           // B blocks
thread A:  sys_mutex_unlock(&m);                         // A preempted
thread B:  sys_mutex_unlock(&m); sys_thread_suspend(B);  // B suspends
thread A:  ...
```

The contended case on different processor cores determines the overhead of the operations in user space, e.g. the internal critical sections of the monitor and the atomic futex operations, on two processor cores in parallel. The test uses `mutex_trylock` instead of a blocking system call and effectively spins until it successfully acquires the mutex. The test releases two cores from a spinning barrier and measures the time until all cores acquire and release the mutex once, and reach the barrier again. The baseline variant implements `mutex_trylock` as system call, while futexes use atomic operations in user space, and the monitor approach uses an internal critical section. During testing, the results showed a great variation in timing between different runs and should be treated as a rough indicator of what to expect.

We can make the following observations:

Observation 1: In the uncontended case, the futex approach is faster than the monitor approach, and both are faster than the baseline approach using system calls. The costs of system calls dominates the overhead of the baseline variant.

Table 5.3: Performance comparison of uncontended and contended mutex scenarios for implementations based on explicit system calls, futexes, and monitors on Marron for three different 32-bit ARM platforms. Results in CPU cycles represent the average of 1024 runs.

Test		**Cortex A8**	**Cortex A9**	**Cortex A15**
uncontended mutex lock/unlock pair	system call	711	469	756
	futex	109	87	149
	monitor	218	150	260
contended mutex lock/unlock scenario (same core)	system call	3463	2461	3471
	futex	3449	2536	3551
	monitor	3205	3067	3195
contended mutex trylock/unlock scenario (2 cores)	system call	—	1275	1953
	futex	—	528	743
	monitor	—	947	1386
contended mutex trylock/unlock scenario (4 cores)	system call	—	3393	—
	futex	—	1186	—
	monitor	—	6312	—

Observation 2: The efficient implementation of IPCP does not contribute much overhead to the fast path. On all three platforms, changing priorities in user space does not cause much overhead. Compare this to the measurement of the priority raise/restore pair in Table 5.2.

Observation 3: Atomic operations cause significant overhead. Atomic operations and memory barriers provide more overhead than the efficient IPCP implementation (compare also to Tables 5.1 and 5.2). When using futexes, a mutex lock / unlock pair comprises a sequence of CAS $\rightarrow$ *acquire barrier* $\rightarrow$ *release barrier* $\rightarrow$ CAS, and the costs add up correspondingly. Similarly, the monitor comprises *two* pairs of spinlock lock / unlock operations of equal complexity than a futex pair. This explains the more than twice as high overhead of the monitor in the fast path. Note that the baseline version shows a similar locking overhead inside the kernel.

Observation 4: In the contended case with blocking, we see mixed results, depending on the architecture. On the Cortex A9, the baseline using system calls is faster. But on the other architectures, monitors are faster. Still, the results are in the same order of magnitude due to the complexity of the test. This underlines the point that the fast paths in the uncontended case come with extra costs in the contended case.

Observation 5: On high contention on the internal critical section of the monitor in the contended test with four cores, the monitor shows worse results than futexes or the baseline. Here, threads in `mutex_trylock` repeatedly spin to lock the internal critical section of the monitor to detect that the mutex is already taken. Futexes do well here, as they can directly probe the mutex due to the atomic operations. In the baseline variant, the system call delays the time between lock attempts and effectively relaxes the contention on the internal spinlock.

Comparison under Varying Degrees of Contention

We conduct another experiment to compare the three discussed approaches (monitor, futex, and baseline) in a scenario of varying degrees of contention.

For this, we distribute 2^{24} random values following a square distribution to a shared hash table comprising 64 hash buckets. We run this experiment using four parallel threads (one for each core) on the Cortex A9. Each thread atomically draws a unique value from the random pool and locks the resulting hash bucket for a constant time of 1 µs. Statistics counters in the different mutex implementations account contended and uncontended cases.

Figure 5.3 shows the average execution time per bucket operation in CPU cycles (including locking overhead) for the varying degrees of contention observed in the hash buckets. We include the results of a run without any locking and therefore without contention as reference shown as dotted horizontal lines. Note that 1 µs relates to 792 CPU cycles on the Freescale i.MX6Q platform.

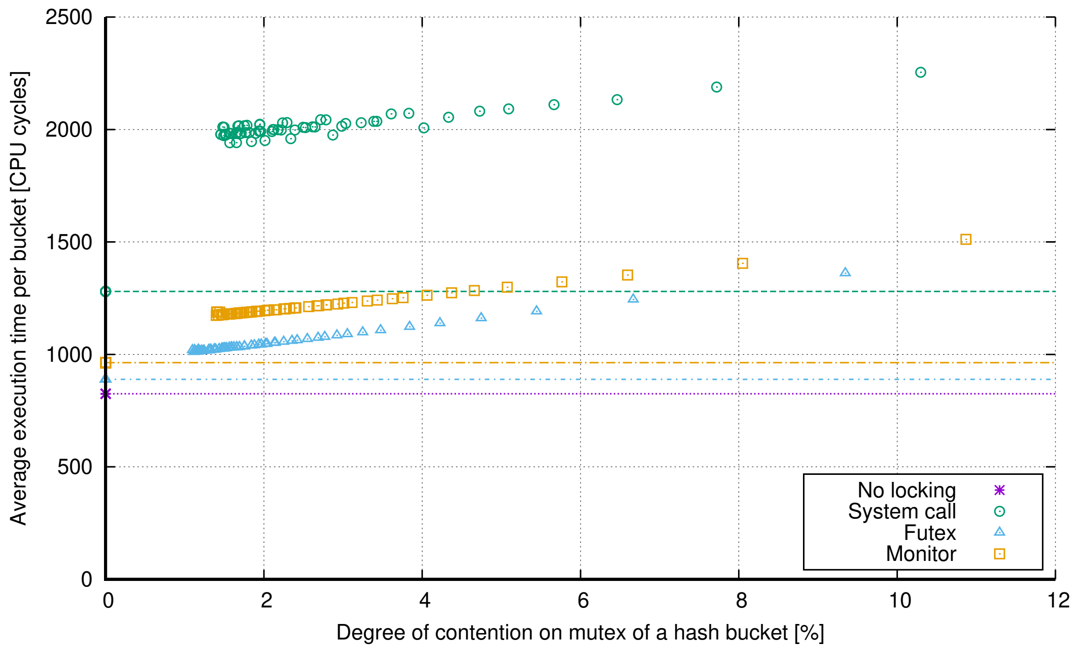

Figure 5.3: Comparison of different designs for mutexes at of varying degrees of contention following a square distribution on a shared hash table with 64 hash buckets, each protected by a mutex. Each thread keeps a hash bucket locked for a constant time of 1 µs. Tests run on Marron on Cortex A9.

The results show that both futexes and monitors result in less overhead in low contention scenarios compared to the system call approach. Also, futexes show less overhead compared to the monitor approach.

A second effect is that both futexes and monitors show *less* contention than the system call approach. Recall that both approaches comprise *two* semantic checks whether to suspend the current thread. The second semantic check in the kernel provides a *second chance* to acquire the mutex after a brief delay (the system call overhead). Again, this effect is stronger in futexes.

5.3 Analysis of Worst-Case Timing Behavior

The evaluation on the analytical level shows that the approaches are predictable. We first present a reference architecture that defines a worst-case timing model in Section 5.3.1. Then we analyze the kernel parts providing low-level blocking and fast IPCP in Section 5.3.2. Based on this, we analyze the higher-level blocking mechanisms in Section 5.3.3.

5.3.1 Reference Architecture for Analysis

We now define the reference architecture for the evaluation.

In Section 3.2.4, we discussed our assumption that the WCET is mainly driven by *cold-cache data memory accesses* and that our resulting approach is to use an abstract worst-case time model comprising polynomial terms of accessed cache lines for n threads and m processors. However, we have to provide specific numbers for the accessed cache lines to further compare the results. Therefore, we use a cold-cache memory access as 1 *unit of time*, i.e. we define that the worst-case time to access a cache line in the data cache, including the time to evict any content of a former dirty cache line, takes 1 unit of time. The typical cache line size of contemporary processors comprises 32 or 64 bytes and provides space for at least 8 pointer-sized variables. Therefore we assume that all data related to a synchronization mechanism fits into one cache line. Note that this "rule-of-thumb" model neglects instruction cache fetches and pipeline delays, as these are not dominating factors for our analysis.

We provide the worst cases for the kernel mechanisms and building blocks as presented in Section 2.3 and discussed in Section 3.4.2.

- In general, **dereferencing a pointer** takes 1 cold-cache memory access or 1 unit of time in the worst case.

- For **indirection/look-up of objects**, we consider the following simplified costs. Array- or table-based look-ups need $\mathcal{O}(1)$ time and take $1 + 1$ cache line accesses. Hash-based look-ups need $\mathcal{O}(n)$ time in the worst case and take $n + 1$ cache line accesses. Look-ups in a binary search tree (BST) need $\mathcal{O}(\log n)$ time and take $2 \cdot \log_2 n + 1$ cache line accesses[1]. In all cases, consider the constant 1 as overhead to dereference an additional pointer.

- For **queuing**, we consider the same bounds as for look-up.

- The time to **enable** or **disable preemption** or to test for pending preemption in a preemption point takes $\mathcal{O}(1)$ time or 2 cold-cache line accesses for the protocols discussed in Sections 3.7 and 4.4.1, i.e. to locate the TLS and access the protocol variables. Some cases of enabling preemption lead to scheduling, but we do not include this additional overhead here because it requires application-specific knowledge whether a thread would be preempted.

- We model the overhead for a successful **spinlock operation** as $\mathcal{O}(1)$ or 1 cache line access, as spinlocks are typically embedded in synchronization objects. This excludes any blocking time, which comprises unsuccessful busy-waiting.

[1]A search in a red-black tree with n nodes requires less than $2 \cdot \log_2(n) + 2$ comparisons [Sed98].

- We model **locking-related overhead** for a critical section as a small constant $t_{pp} = 2 + 1 + 1 + 2 = 6$ that comprises disabling preemption, two operations to first lock and then unlock a spinlock, and enabling preemption again. This is also the overhead of a preemption point (but in different order), hence the subscript *pp*.
- For a **critical section** of t_{CS} length, a thread may observe blocking by $m - 1$ other processors when using spinlocks. We denote the blocking as $t_{blocking} = (m - 1) \cdot t_{CS}$. But we also need to account the overhead for preemption control and spinlock operations. Therefore, the resulting overall worst-case time of a critical section including all overheads and blocking is $t_{blocking} + t_{CS} + t_{pp}$, which simplifies to $m \cdot t_{CS} + t_{pp}$.
- The overhead for a **system call** takes $\mathcal{O}(1)$ or t_{sys} time. This includes both the system call entry and exit code and any overheads of dispatching a system call function.
- For **scheduling operations**, we include any costs for timeout handling when suspending or waking up a thread. We keep these costs abstract as:
  - $t_{sched-preempt}$ to preempt the current thread,
  - $t_{sched-wait}$ to let the current thread wait,
  - $t_{sched-wake}$ for a wake-up operation, and
  - $t_{timeout-clear}$ to clear any pending timeouts.
- We define that a successful **atomic operation**, e.g. an operation on the futex value, takes 1 cache line access. This includes any memory barriers, but excludes any blocking time. Memory barriers are usually much faster compared to cold-cache memory accesses, as our assessment for ARM in Section 5.1 shows[2].
- We further assume that **loops with atomic operations** will *eventually* complete. If we can exclude malicious actors that actively try to interfere with an atomic operation, e.g. writing to the same cache line in a tight loop, we can then model that concurrent atomic operations by m processors will complete after at most m retries, i.e. in each attempt one of the processors succeeds until eventually all m processors complete. The worst-case time for an atomic operation therefore is m.

With this, we can model the operations of the blocking synchronization mechanisms of Table 3.1 in Section 3.6 from user space down to the scheduler level.

[2]Al Bahra shows comparable numbers for Power 7 (Table 2 in [Al-13]), which uses a weakly-ordered memory model similar as ARM.

5.3.2 Analysis of Kernel Primitives

We now evaluate the kernel operations of the different designs for their worst-case timing and compare the results. In each of the following analyses, we determine the impact of a design's internal data structures and locking on the blocking time. From this, we derive a resulting worst-case model for the common operations handling one or all threads. The resulting worst-case models comprise parts that are the same for comparable operations, e.g. system call overhead (t_{sys}), suspension ($t_{sched-wait}$), or wake-up of k threads ($k \cdot t_{sched-wake}$), and parts that comprise the *design-specific overhead*.

We start with a discussions on the worst-case costs of futexes in general.

General Worst-Case Costs of Futex Operations

We start with an analysis of the worst-cases of the particular futex operations presented in Section 2.2.11 at kernel level. For an implementation, we assume that the overall structure in the kernel follows our reference architecture in Section 5.3.1. The following discussion is the same for all futex designs.

To determine the worst-case time of futex operations, we decompose futex operations into their building blocks. An implementation comprises different wait queues, and wait queues are protected by locks. We first determine the worst-case time of an individual operation. We then derive worst-case blocking times, and so on. Therefore, we start the discussion from the inside out.

In the kernel, we can observe the following operations on wait queues:

- Insertion of a thread into a wait queue.
- Removal of a thread from a wait queue.
- Wake-up of k threads from a wait queue. The locking and preemption strategies of a design decide if the wait queue remains locked to wake up all k threads, or if threads are woken up one by one.
- Requeuing of k threads comprise both removal and insertion. The locking strategy of a design decides if both source and target wait queues are locked at once or consecutively, and the preemption strategy decides if threads are requeued one by one or all at once.

Due to their complexity, requeue operations are often the worst-case operations that dominate the worst-case time wait queues are locked. From this, we derive an individual wait-queue operation's worst-case time t_{wq}. Next we can derive the worst-case blocking time $t_{blocking}$ by $m - 1$ other processors a thread can observe while trying to lock a wait queue. For a specific operation on the locked wait queue, e.g. waiting, wake-up, etc., we must also include any locking-related overheads, so the operation usually takes $t_{blocking} + t_{operation} + t_{pp}$ time.

We now determine the worst cases of the futex system calls. Note that all futex operations in the kernel comprise the system call overhead t_{sys}.

- A `futex_wait` operation first locates and locks the wait queue, inserts the calling thread into the wait queue, and then suspends the thread in the scheduler. The worst-case behavior is when a spurious wake-up happens. In this case, the futex operation locks the wait queue again to remove a thread: $t_{futex-wait} = t_{sys} + t_{wq-insert} + t_{sched-wait} + t_{wq-remove} + \dots$

  Note that the parts $t_{sys} + t_{sched-wait}$ describe the same overhead in all futex designs, while the remaining parts depend on the particular futex design.

- A `futex_lock` operation needs to perform atomic operations on the futex value additionally to `futex_wait`. We follow our model of atomic operations in the reference architecture and assume this needs m additional steps: $t_{futex-lock} = t_{futex-wait} + m$.

- A `futex_wake` operation of one thread comprises a wait queue operation to remove a thread and an operation to wake up the thread: $t_{futex-wake-one} = t_{sys} + t_{wq-remove} + t_{sched-wake} + \dots$

- A `futex_wake` operation on all k relevant threads of a wait queue depends on the locking and preemption strategies of the particular design. An operation includes at least the following necessary steps: $t_{futex-wake-all} = t_{sys} + k \cdot t_{wq-remove} + k \cdot t_{sched-wake} + \dots$

- A `futex_unlock` operation does not require atomic operations and has similar characteristics than `futex_wake` of one thread: $t_{futex-unlock} = t_{futex-wake-one}$

- A `futex_requeue` operation on one thread comprises two wait queue operations. It first removes the thread from one wait queue and then inserts the thread into another wait queue: $t_{futex-requeue-one} = t_{sys} + t_{wq-remove} + t_{wq-insert} + \dots$

  Additionally, a design might optionally clear pending timeouts, as described in Section 4.3.2. This requires at least: $t'_{futex-requeue-one} = t_{sys} + t_{wq-remove} + t_{wq-insert} + t_{timeout-clear} + \dots$

- A `futex_requeue` operation on all k relevant threads on a wait queue requires: $t_{futex-requeue-all} = t_{sys} + k \cdot t_{wq-remove} + k \cdot t_{wq-insert} \cdots$

  In a design with clearing of timeouts, the operation takes: $t'_{futex-requeue-all} = t_{sys} + k \cdot t_{wq-remove} + k \cdot t_{wq-insert} + k \cdot t_{timeout-clear} + \dots$

This general overview provides a blueprint for the following analyses.

Futexes with Hashed Wait-Queues

We now discuss an analysis of a futex design using hashed wait queues similar to Linux as described in Section 3.5.2. Note that the actual WCET in Linux differs from the presented approach, as the assumptions of our simplified reference architecture do not necessarily hold for the Linux kernel, which has a higher implementation complexity. Especially, we omit the handling of Linux PI mutexes.

The kernel first hashes the futex address to derive a hash bucket, locks the hash bucket, and then handles waiting threads in a *plist* data structure of p priority levels. Recall from Section 3.5.2 that Linux has 140 internal priority levels. All operations on the wait queue are non-preemptive.

Therefore, the operations on the hash bucket dominate the execution time of the internal critical sections t_{hb}. We assume that a hashed wait queue is shared and contains n threads, with k threads specific to a futex of interest ($k \leq n$). We now determine the worst contender.

- Insertion takes at most p search steps to locate the right priority level. We assume this accesses p unique cache lines in the worst case:
  $t_{insert} = p$.
- Removal changes two adjacent nodes and two linked lists in the plist:
  $t_{remove} = 4$
- Wake-up of threads takes at most $t_{search} = n$ steps to locate a thread matching the futex value on a wait queue of n threads. Threads to wake up are first moved to a dedicated *wake-up queue* in $\mathcal{O}(1)$ time. We assume $t_{insert-wake-queue} = 1$ cache line access for this.

  In the worst case, a wake-up operation of one thread must search the full plist, remove the thread from the wait queue, and insert the thread on the wake-up queue:
  $t_{wake-one} = t_{search} + t_{remove} + t_{insert-wake-queue} = n + 5$.
- An operation that wakes up all k threads of the same futex on a shared wait queue must search the full wait queue of n threads to locate the k relevant threads, remove the threads from the wait queue, and insert them into the wake-up queue:
  $t_{wake-all} = n + 5k$.
- Requeuing of threads also iterates all n threads on the wait queue. If the requeue operation targets the same hashed wait queue, the threads are updated in place after at most n steps. But if the requeue operation targets a different hashed wait queue, the threads are first removed and then inserted into the new wait queue. As worst case, we assume that the target wait queue is fully populated, so requeuing of one thread observes at most:
  $t_{requeue-one} = t_{search} + t_{remove} + t_{insert} = n + 4 + p$.

- Requeuing k threads takes:
  $t_{requeue-all} = n + k \cdot (4 + p)$.

The worst-case operation on a wait queue is found in the requeue operation. The maximum number of blocked threads on a wait queue could be very large. For simplicity, we assume the system limit of N threads, and replace n and k by N to derive a worst-case hash bucket (hb) operation of:
$t_{hb} = N + N \cdot (4 + p) = N \cdot (p + 5)$.

Note that a requeue operation locks up to *two* wait queues. The wait queues are locked in increasing order to prevent deadlocks. But even in case of nested locks, there can be only $m - 1$ other processors performing operations on the first wait queue, and then $m - 1$ other processors performing operation on the second wait queue. Also, we can construct a worst-case scenario where each processor requeues the same set of threads to the next wait queue over and over. This results in a worst-case blocking time for a wait queue operation of:
$t_{blocking} = 2 \cdot (m - 1) \cdot t_{hb} = 2N \cdot (m - 1) \cdot (p + 5)$.

Note that the blocking time already depends on all variables. For this reason, a hash bucket lock is implemented as blocking mutex in a Linux real-time kernel.

We now determine the worst cases of the futex system calls. We put design-specific parts in parenthesis.

- A `futex_wait` operation comprises two wait queue operations due to a spurious wake-up:
  $t_{l-futex-wait}$
  $= t_{sys} + (t_{blocking} + t_{insert} + t_{pp}) + t_{sched-wait} + (t_{blocking} + t_{remove} + t_{pp})$
  $= t_{sys} + t_{sched-wait} + 4N \cdot (m - 1) \cdot (p + 5) + p + 16$.

- A `futex_lock` operation needs to perform atomic operations on the futex value additionally to $t_{l-futex-wait}$. In Section 3.5.3, we argued that there is no upper bound on the retries when we have to consider malicious actors:
  $t_{l-futex-lock-unbounded} = \infty$.

  But to provide a bound, we assume that the atomic operations take m additional steps:
  $t_{l-futex-lock} = t_{l-futex-wait} + m$.

- A `futex_wake` operation of one thread takes at most:
  $t_{l-futex-wake-one} = t_{sys} + (t_{blocking} + t_{wake-one} + t_{pp}) + t_{sched-wake}$
  $= t_{sys} + t_{sched-wake} + 2N \cdot (m - 1) \cdot (p + 5) + N + 11$.

  Here, we assume N unrelated threads on the wait queue.

- A `futex_wake` operation on all k relevant threads of a wait queue first moves the threads to the wake-up queue and then wakes up the threads one by one preemptively. We model additional overhead to process the wake-up queue as the cost of a preemption point:

$t_{l-futex-wake-all} = t_{sys} + (t_{blocking} + t_{wake-all} + t_{pp}) + k \cdot (t_{sched-wake} + k \cdot t_{pp})$
$= t_{sys} + k \cdot t_{sched-wake} + 2N \cdot (m-1) \cdot (p+5) + N + 11k + 6.$

Note that this includes only one locked wait queue operation.

- A `futex_unlock` is $t_{l-futex-unlock} = t_{l-futex-wake-one}$.
- A `futex_requeue` operation of one thread takes:
  $t_{l-futex-requeue-one} = t_{sys} + (t_{blocking} + t_{requeue-one} + t_{pp})$
  $= t_{sys} + 2N \cdot (m-1) \cdot (p+5) + N + p + 10.$

  Note that requeue operations do not clear timeouts in Linux.
- A `futex_requeue` operation of all k threads takes:
  $t_{l-futex-requeue-all} = t_{sys} + (t_{blocking} + t_{requeue-all} + t_{pp})$
  $= t_{sys} + 2N \cdot (m-1) \cdot (p+5) + N + k \cdot (p+4) + 6.$

In all cases, we can observe that the design-specific overhead in a hash-based futex design is dominated by the blocking time and that this overhead depends on $m \cdot N \cdot p$.

Note that in the overhead, the dependency on N relates to the interference problems due to blocked threads of *other* partitions described in Section 3.5.3.

Deterministic Futexes

We now determine the worst case of the deterministic futex design of Section 4.1.

Recall that deterministic futexes use two nested BSTs, but individual operations are preemptive after processing each thread. One lock protects both BSTs, so we determine the worst case of an operation.

- Locating, inserting, and removing a wait queue in the address tree requires at most $2 \cdot \log_2 n + 1$ cache line accesses for n wait queues.
- Locating, inserting, and removing a waiting thread in a wait queue requires at most $2 \cdot \log_2 n + 1$ cache line accesses for n waiting threads.
- A number of n threads can wait on the same wait queue or on n different wait queues, maxing out either the wait queue *or* the address tree. For wait and wake operations, the worst case is when both address tree and wait queue are equally filled with $\frac{n}{2}$ nodes. In this case, an operation takes at most $4 \cdot \log_2 \frac{n}{2} + 2$ cache line accesses.
- For requeue operations, the worst case is when the address tree and both source and target wait queue are equally filled with $\frac{n}{3}$ nodes. In this case, an operation takes at most $6 \cdot \log_2 \frac{n}{3} + 3$ cache line accesses.

Like in the hash-based futex design, the requeue operation dominates the worst case. For shared futexes, we have to assume worst-case interference by other threads and replace n by N. We then end up with a worst case of:
$t_{bst} = 6 \cdot \log_2 \frac{N}{3} + 3$.
We use this for *all* BST operations in the following.

If we consider the blocking during preemptible operations, we know that one lock protects both source and target wait queues in a requeue operation and we get interference by at most $m - 1$ other processors, so the blocking time is:
$t_{blocking} = (m - 1) \cdot t_{bst}$.
The resulting time for a locked BST operation is:
$t_{bst-locked} = t_{blocking} + t_{bst} + t_{pp} = m \cdot t_{bst} + t_{pp}$.

We now determine the worst cases of the futex system calls.

- A `futex_wait` operation with a spurious wake-up takes:
  $t_{d-futex-wait}$
  $= t_{sys} + (t_{blocking} + t_{bst} + t_{pp}) + t_{sched-wait} + (t_{blocking} + t_{bst} + t_{pp})$
  $= t_{sys} + t_{sched-wait} + 2m \cdot (6 \cdot \log_2 \frac{N}{3} + 3) + 12$.
- A `futex_lock` operation performs atomic operations on the futex value from kernel space. Section 4.3.1 describes the trade-off to bound the number of atomic retries in the kernel at the cost of retries in user space. Like in the hash-based futexes, we consider a fixed number of m retries before failing:
  $t_{d-futex-lock} = t_{d-futex-wait} + m$.
- A `futex_wake` operation of one thread takes:
  $t_{d-futex-wake-one} = t_{sys} + (t_{blocking} + t_{bst} + t_{pp}) + t_{sched-wake}$
  $= t_{sys} + t_{sched-wake} + m \cdot (6 \cdot \log_2 \frac{N}{3} + 3) + 6$.
- A `futex_wake` operation of k threads is preemptive and takes at most:
  $t_{d-futex-wake-all} = t_{sys} + k \cdot ((t_{blocking} + t_{bst} + t_{pp}) + t_{sched-wake} + t_{pp})$
  $= t_{sys} + k \cdot t_{sched-wake} + k \cdot m \cdot (6 \cdot \log_2 \frac{N}{3} + 3) + 12k$.
- A `futex_unlock` operation $t_{d-futex-unlock}$ has similar characteristics as a `futex_wake` operation for one thread.
- A `futex_requeue` operation also clears any pending timeouts. Requeuing one thread takes:
  $t_{d-futex-requeue-one} = t_{sys} + (t_{blocking} + t_{bst} + t_{pp}) + t_{timeout-clear}$
  $= t_{sys} + t_{timeout-clear} + m \cdot (6 \cdot \log_2 \frac{N}{3} + 3) + 6$.
- A `futex_requeue` operation of all k thread takes:
  $t_{d-futex-requeue-all} = t_{sys} + k \cdot ((t_{blocking} + t_{bst} + t_{pp}) + t_{timeout-clear} + t_{pp})$
  $= t_{sys} + k \cdot t_{timeout-clear} + k \cdot m \cdot (6 \cdot \log_2 \frac{N}{3} + 3) + 12k$.

Here, the blocking time again dominates the design-specific overheads of the worst-case times. The worst-case overhead depends on $m \cdot \log N$ for preemptible operations. This reduces the dependency on threads of other partitions to a logarithmic bound compared to a hash-based design like in Linux with a linear bound.

Static Futexes

We determine the worst case for a design based on static futexes as presented in Section 4.2. We use the following considerations for the system. Wait queues are allocated in advance and not shared among partitions or processes. A single lock protects all wait queues of a partition or a process. Operations on wait queues are non-preemptible. We further assume that linked lists are used to manage wait queues. Also, the non-preemptive design wakes up waiting threads or clears the timeout for requeued threads *with* the wait queue lock held.

These design considerations follow our design decisions for AUTOBEST in Section 4.2.2 and result in a minimal kernel implementation. In contrast to hash-based futexes and deterministic futexes, we assume that only a small number of n threads wait on a wait queue.

We first look at the worst-case operations on wait queues.

- Insertion into a priority-ordered wait queue needs at most:
  $t_{insert} = n + 1$.

  For removal of one waiting thread, we consider:
  $t_{remove} = 3$.

- Due to the non-preemptive design, operations processing more than one thread are critical. Waking up k waiting threads takes $k \cdot t_{remove}$ cache line accesses to handle the wait queue. However, we must also consider k wake-up operations at scheduler level:
  $t_{wake-all} = k \cdot t_{sched-wake} + 3k$.

- Requeuing of one thread takes $t_{remove} + t_{insert} + t_{timeout-clear}$ time. For requeuing of k threads, we need at most:
  $t_{requeue-all} = k \cdot t_{timeout-clear} + k^2 + 4k$.

We see that the operations that process more than one thread are the most expensive ones, but we do not know which of the kernel operations is costlier. If we consider the worst case of $k = n$ threads on a wait queue, we get:
$t_{s-wq} = \max(t_{wake-all}, t_{requeue-all})$
$= \max(n \cdot t_{sched-wake} + 3n, n \cdot t_{timeout-clear} + n^2 + 4n)$.

Considering $m - 1$ other processors, the worst case blocking time is:
$t_{blocking} = (m - 1) \cdot t_{s-wq}$.

We now analyze the worst cases of the futex system calls.

- A `futex_wait` operation with a spurious wake-up takes:
  $t_{s-futex-wait}$

$= t_{sys} + (t_{blocking} + t_{insert} + t_{pp}) + t_{sched-wait} + (t_{blocking} + t_{remove} + t_{pp})$
$= t_{sys} + t_{sched-wait} + 2 \cdot (m-1) \cdot \max(\ldots) + n + 16.$

- For `futex_lock`, the same argument as for deterministic futexes applies, so the operation adds an overhead m to the worst-case results of `futex_wait` to bound the number of atomic operations before it fails with an error: $t_{s-futex-lock} = t_{s-futex-wait} + m$.
- A `futex_wake` operation of one thread takes:
  $t_{s-futex-wake-one} = t_{sys} + (t_{blocking} + t_{remove} + t_{sched-wake} + t_{pp})$
  $= t_{sys} + t_{sched-wake} + (m-1) \cdot \max(\ldots) + 9.$
- A `futex_wake` operation of $k = n$ threads takes:
  $t_{s-futex-wake-all} = t_{sys} + (t_{blocking} + k \cdot (t_{remove} + t_{sched-wake}) + t_{pp})$
  $= t_{sys} + n \cdot t_{sched-wake} + (m-1) \cdot \max(\ldots) + 3n + 6.$
- A `futex_unlock` operation $t_{s-futex-unlock}$ has similar characteristics as the `futex_wake` operation for one thread.
- A `futex_requeue` operation clears pending timeouts. Requeuing one thread takes:
  $t_{s-futex-requeue-one} = t_{sys} + (t_{blocking} + t_{remove} + t_{insert} + t_{timeout-clear} + t_{pp})$
  $= t_{sys} + t_{timeout-clear} + (m-1) \cdot \max(\ldots) + n + 10.$
- A `futex_requeue` operation on all $k = n$ thread takes:
  $t_{s-futex-requeue-all} = t_{sys} + (t_{blocking} + k \cdot (t_{remove} + t_{insert} + t_{timeout-clear}) + t_{pp})$
  $= t_{sys} + n \cdot t_{timeout-clear} + (m-1) \cdot \max(\ldots) + n^2 + 4n + 6.$

Here, the blocking term $(m-1) \cdot \max(\ldots)$ includes kernel operations, in particular the time it takes to wake up or clear the timeout of all threads, and dominates the worst-case time for $m \geq 2$. Also, there are terms of n^2 affecting the worst-case time. This is only acceptable if both m and n are small, therefore static futexes focus on implementations for single processors. However, there is no interference by threads of other processes or partitions.

Note that we deliberately opted for such *worse* design decisions compared to the other designs to show the impact of tight coupling of wait queue and scheduler operations without preemption points. Still, alternative designs that reduce the blocking time are possible. When processing *all* threads, a design can for example *detach* a complete wait queue by swapping the wait queue head with an empty wait queue in $\mathcal{O}(1)$ and then process the blocked threads preemptively. This would allow to exclude scheduler operations from the blocking time.

Light-Weight Blocking for IPCP

We now evaluate the light-weight blocking mechanisms for IPCP of Section 4.4.2.

The kernel interface comprises just a wait and a wake-up operation. Both operations compare a futex-like value in user space. The wake-up operation references the target thread by a thread ID. For this, we assume a lock-free array-based look-up mechanism in $\mathcal{O}(1)$ time and that the look-up takes $t_{look-up} = 2$ time. We follow the suggested architecture of Table 3.1 in Section 3.6 and compare the value in user space under the scheduler lock in constant time of $t_{compare} = 2$ cache line accesses. Also, we assume that an update of the `uprio` and the `nprio` protocol variables takes $t_{update-vars} = 2$ cache line accesses.

The worst case considerations of the wait and wake-up system calls are:

- The `wait_at_prio` operation comprises:
  $t_{wait-at-prio} = t_{sys} + t_{compare} + t_{sched-wait}$
  $= t_{sys} + t_{sched-wait} + 2.$
- The `wake_set_prio` operation comprises:
  $t_{wake-set-prio} = t_{sys} + t_{look-up} + t_{compare} + t_{sched-wake} + t_{update-vars} + t_{pp}$
  $= t_{sys} + t_{sched-wake} + 8.$

  We assume that the preemption point is not taken, similar to the other wake-up operations. Preemption adds $t_{sched-preempt}$ to the term.

Compared to the wait and wake-up one operations of futex-based approaches, we observe only constant overhead here.

System Calls for Priority Changes and Preemption

When implementing IPCP using system calls in a baseline design, the implementation would probably provide a `prio_set` system call, similar to the one presented in Section 5.2.2, and would also use *Benno scheduling* (Section 3.1.2) to speed up internal operations. The implementation updates the in-kernel representation of the protocol variables and compares the given priority to the priority of the next thread on the ready queue in $\mathcal{O}(1)$ time. We assume two cache line accesses for each dereferenced pointer, so the resulting worst-case is:
$t_{prio-set} = t_{sys} + t_{update-vars} + t_{check-for-preemption} = t_{sys} + 4.$

Note that we again excluded the costs of preemption here. With preemption, the costs are: $t_{prio-set-with-preemption} = t_{sys} + 4 + t_{sched-preempt}.$

The fast IPCP protocols of Section 4.4.1 use a system call to preempt the thread when necessary. We can now argue that such a preemption system call will have similar complexity and similar costs:
$t_{sys-preempt} = t_{prio-set}$ and $t_{sys-preempt-with-preemption} = t_{prio-set-with-preemption}.$

Comparison of Kernel Primitives

We now compare the different approaches and start with the futex-based designs.

We first discussed the worst case of futexes with hashed wait-queues like in Linux. The design decouples operations on wait queues from low-level scheduler operations. Operations on the wait queues are non-preemptive. The worst-case overhead depends on $m \cdot N$. This is also the worst-case interference other threads of unrelated processes or partitions might observe. The analysis is not directly applicable to a real Linux implementation, as our model is reduced in complexity and omits many corner cases, but it should show the trends correctly.

Next, we analyzed deterministic futexes. Deterministic futexes also decouple operations on wait queues from low-level scheduler operations, however, the design is preemptive after processing each thread. The worst-case interference other threads might observe is reduced to $m \cdot \log N$ compared to a hash-based design. But the caller might observe this overhead for each processed thread, which affects the worst-case time of individual operations. This aspect is worse compared to the overhead of a hash-based design.

We then discussed static futexes. Static futexes are non-preemptive and include the low-level scheduler operations into the wait queue operations. Therefore, the worst-case overhead also depends on scheduler operations: $m \cdot n \cdot \max(t_{sched-wake}, t_{timeout-clear})$. The design choice to include low-level scheduler operations into non-preemptive wait queue operations only makes sense for both a small number of threads n and a small number of processors m.

In all considered designs, a `futex_lock` operation must bound the number of retries of the atomic operations to be analyzable, but the operations still fail. Also, the `futex_requeue` operation just requeued threads, as we have omitted wake-up operations for "naked notifies" in the analyses.

In so far, the analyses match the expectations of the selected designs. Especially the design of deterministic futexes addresses the issues of the Linux design at kernel level. The static futex design as presented is only usable when all parameters are known and small, e.g. in a statically configured system.

For the light-weight blocking in IPCP, we can observe that the implementation provides a thin wrapper with constant overhead around the scheduler services for suspending and wake-up of threads. The same observation holds for system calls to change the scheduling priority of a thread and to preempt the current thread.

5.3.3 Analysis of Mutexes and Condition Variables

We now evaluate both mutexes and condition variables of the different design approaches. We start with a baseline design using system calls, evaluate a futex-based design, and then consider a monitor-based design. For the futex- and monitor-based designs, we include the kernel operations of Section 5.3.2 in our worst-case considerations. Finally, we compare the results and the design-specific overheads.

System-Call-Based Synchronization Mechanisms

We first determine the worst case of a baseline implementation of mutexes and condition variables using system calls. Similar to the design of static futexes in Section 4.2, we assume that wait queues are allocated upfront. But unlike the analysis of static futexes in Section 5.3.2, we use design choices focusing on scalability. Like deterministic futexes in Section 5.3.2, the baseline design uses a BST as wait queue, and operations on the wait queue are preemptible after processing each thread. However, each wait queue has its own lock, and mutex and condition variable operations lock at most one wait queue at a time. Note that a *wake all* operation is not needed for mutex or condition variables.

As approach to determine the worst case, we follow the general approach for the kernel primitives of Section 5.3.2.

We first determine the worst case of operations on a wait queue. Locating, inserting, and removing of a thread in a wait queue requires at most $2 \cdot \log_2 n + 1$ cache line accesses for n waiting threads. This is also our internal worst case t_{b-wq}, therefore $t_{blocking} = (m - 1) \cdot t_{b-wq}$ when considering m processors.

We now determine the worst cases of the system calls.

- A `mutex_lock` operation with a spurious wake-up locks the BST twice to first insert and then remove a thread again. This takes:
  $t_{b-mutex-lock}$
  $= t_{sys} + (t_{blocking} + t_{insert} + t_{pp}) + t_{sched-wait} + (t_{blocking} + t_{remove} + t_{pp})$
  $= t_{sys} + t_{sched-wait} + 4m \cdot \log_2 n + 2m + 12.$

- A `mutex_unlock` operation takes:
  $t_{b-mutex-unlock} = t_{sys} + (t_{blocking} + t_{remove} + t_{pp}) + t_{sched-wake}$
  $= t_{sys} + t_{sched-wake} + 2m \cdot \log_2 n + m + 6.$

- A `cond_wait` operation first locks the condition variable wait queue and adds the thread to the wait queue. Then it locks the mutex wait queue, unlocks the mutex, and removes the next mutex owner. It wakes up the next mutex owner, and then suspends the calling thread. After wake-up, it must handle a spurious wake-up. For this, the operation removes the thread from the condition variable wait queue, adds it to the mutex wait queue, and waits again. Afterwards, it locks the wait queue and checks for consistency (see Section 4.4.2 for a related reason in the monitor case). This takes:
  $t_{b-cond-wait} = t_{sys} + (t_{blocking} + t_{insert} + t_{pp}) + (t_{blocking} + t_{remove} + t_{pp}) + t_{sched-wake} + t_{sched-wait} + (t_{blocking} + t_{remove} + t_{pp}) + (t_{blocking} + t_{insert} + t_{pp}) + t_{sched-wait} + (t_{blocking} + t_{pp})$
  $= t_{sys} + 2 \cdot t_{sched-wait} + t_{sched-wake} + (10m - 2) \cdot \log_2 n + 5m + 29.$

- A `cond_signal` operation removes a thread from the condition variable wait queue and adds the thread to the mutex wait queue. We assume the mutex is properly locked during this operation. This also clears any pending timeouts:

$t_{b-cond-signal}$
$= t_{sys} + (t_{blocking} + t_{remove} + t_{pp}) + t_{timeout-clear} + (t_{blocking} + t_{insert} + t_{pp})$
$= t_{sys} + t_{timeout-clear} + 4m \cdot \log_2 n + 2m + 12.$

Support for "naked notifies" additionally needs $t_{sched-wake}$.

- A `cond_broadcast` operation follows the blueprint of `cond_signal` in a loop. For k threads, it takes:
  $t_{b-cond-broadcast}$
  $= t_{sys} + k \cdot ((t_{blocking} + t_{remove} + t_{pp}) + t_{timeout-clear} + (t_{blocking} + t_{insert} + t_{pp}))$
  $= t_{sys} + k \cdot t_{timeout-clear} + 4k \cdot m \cdot \log_2 n + 2k \cdot m + 12k.$

Note that the blocking time always depends on $m \cdot \log n$ per internal wait queue operation. This also dominates the design-specific overheads.

From a worst-case analysis point of view, this presented design is a hybrid of the other designs. Compared to the futex-based designs analyzed in Section 5.3.2, the kernel handles all corner cases internally.

Futex-Based Synchronization Mechanisms

We now present an analysis of mutexes and condition variables based on futexes and following the designs in Sections 4.3.1 and 4.3.2. We combine the user space results with the results of the analysis of deterministic futexes of Section 5.3.2.

Note that in all cases, an implementation of a synchronization mechanism in user space is a relatively thin wrapper upon the futex API in the kernel and mostly comprises constant operations, but we have to consider the time for extra retries if the comparison of the futex value in the kernel fails. However, unlike in the analysis of the kernel operations, we can now exclude malicious behavior and assume that loops with atomic operations in user space and in the kernel will complete after at most m retries.

- A `mutex_lock` operation tries to lock the mutex in the fast path at most m times before calling the `futex_lock` in the kernel. We assume the kernel operation succeeds after m steps:
  $t_{f-mutex-lock} = m + t_{d-futex-lock} = 2m + t_{d-futex-wait}$
  $= t_{sys} + t_{sched-wait} + 2m \cdot (6 \cdot \log_2 \frac{N}{3} + 3) + 2m + 12.$

- A `mutex_unlock` operation fails the fast path after m times and then calls `futex_unlock`:
  $t_{f-mutex-unlock} = m + t_{d-futex-unlock}$
  $= t_{sys} + t_{sched-wake} + m \cdot (6 \cdot \log_2 \frac{N}{3} + 3) + m + 6.$

- A `cond_wait` operation comprises the following steps in the worst case: (i) read the condition variable's sequence counter (constant), (ii) unlock the support mutex (`mutex_unlock`), (iii) wait on the sequence variable (`futex_wait`),

(iv) observe a spurious wake-up, and (v) lock the mutex again (`mutex_lock`). Thus, the worst case is:
$t_{f-cond-wait} = 1 + t_{f-mutex-unlock} + t_{d-futex-wait} + t_{f-mutex-lock}$
$= 3 \cdot t_{sys} + 2 \cdot t_{sched-wait} + t_{sched-wake} + 5m \cdot (6 \cdot \log_2 \frac{N}{3} + 3) + 3m + 30.$

- A `cond_signal` operation increments the sequence counter and then calls `futex_requeue` to requeue one thread. As usual, we assume the mutex is properly locked during this operation. The worst case takes:
$t_{f-cond-signal} = m + t_{d-futex-requeue-one}$
$= t_{sys} + t_{timeout-clear} + m \cdot (6 \cdot \log_2 \frac{N}{3} + 3) + m + 6.$
- A `cond_broadcast` operation increments the sequence counter and calls `futex_requeue` to requeue all threads. The worst case for k threads takes:
$t_{f-cond-broadcast} = m + t_{d-futex-requeue-all}$
$= t_{sys} + k \cdot t_{timeout-clear} + k \cdot m \cdot (6 \cdot \log_2 \frac{N}{3} + 3) + 12k + m.$

The overheads added by the failed fast paths in user space comprise the extra operations to handle the atomic operations. The worse-case overhead in the kernel of $m \cdot logN$ still dominates.

Recall from Section 5.3.2 that for shared futexes, a wait queue operation includes interference from threads of other partitions/processes, but if we replace N by n, we get the worst-case for private, non-shared futexes with n threads in a partition, and then $m \cdot logn$ dominates the overhead.

Monitor-Based Synchronization Mechanisms

We now analyze the monitor-based mutex and condition variables presented in Section 4.5.2 and Section 4.5.3.

We start with an analysis of the low-level monitor primitives of Section 4.4.3. We assume that a monitor protects some internal data, and that the related critical section takes t_{CS} time, so the blocking time comprises $t_{blocking} = (m-1) \cdot t_{CS}$. Note that the following primitives do not contain operations on wait queues.

- An *enter* operation modifies the state of user space variables in a constant time to disable preemption, then waits the full blocking time of the internal spinlock before the caller acquires the internal lock. Recall from the reference architecture in Section 5.3.1 that t_{pp} accounts the overhead for this, i.e. disabling preemption, two operations to lock and unlock a spinlock, and enabling preemption again, without taking a system call for preemption. We use half of the overhead of the enter operation and half of the overhead for a leave operation:
$t_{m-enter} = \frac{1}{2} t_{pp} + t_{blocking} = (m-1) \cdot t_{CS} + 3.$
- Similarly, a *leave* operation modifies user space variables to unlock the spinlock and enable preemption in a constant time and then issues a system call

for preemption. We leave out the actual time for preemption $t_{sched-preempt}$, like in the other considerations:
$t_{m-leave} = \frac{1}{2}t_{pp} + t_{sys-preempt} = 3 + (t_{sys} + 4) = t_{sys} + 7$.

- A *notify* operation defers the actual wake-up until the caller has unlocked the internal spinlock and calls the fused system call to both wake up and preempt. We account this as a *leave with wake-up* operation:
  $t_{m-leave-with-wake-up} = \frac{1}{2}t_{pp} + t_{wake-set-prio} = 3 + (t_{sys} + t_{sched-wake} + 8)$
  $= t_{sys} + t_{sched-wake} + 11$.

  Note that the additional overhead to wake up a thread is $t_{sched-wake} + 4$.

- A *wait* operation first unlocks the internal spinlock, calls the kernel to wait, and then locks the internal spinlock again.
  $t_{m-wait} = \frac{1}{2}t_{pp} + t_{wait-at-prio} + \frac{1}{2}t_{pp} + t_{blocking}$
  $= 3 + (t_{sys} + t_{sched-wait} + 2) + 3 + (m-1) \cdot t_{CS}$
  $= t_{sys} + t_{sched-wait} + (m-1) \cdot t_{CS} + 8$.

Besides the blocking time in the *enter* and *wait* operations, the monitor directives map to the low-level kernel primitives with a constant overhead.

We now refine the internal data protected by the spinlock and introduce the wait queue. Like in the system-call-based baseline design, we use a BST for the wait queue, therefore locating, inserting, and removing of a thread in a wait queue requires at most $t_{m-wq} = 1 + 2 \cdot \log_2 n$ cache line accesses for n threads. However, this is yet not sufficient to define t_{CS}. We must analyze each operation first. Note that the parts comprise t_{CS} are put in parenthesis.

- A contended `mutex_lock` operation comprises *enter*, insertion into the wait queue, *wait*, a spurious wake-up, the resulting removal from the wait queue, and *leave*. This takes:
  $t_{m-mutex-lock} = t_{m-enter} + (t_{m-wq}) + t_{m-wait} + (t_{m-wq}) + t_{m-leave}$
  $= 2 \cdot t_{sys} + t_{sched-wait} + 2 \cdot t_{m-wq} + 2(m-1) \cdot t_{CS} + 18$.

- A contended `mutex_unlock` operation comprises *enter*, removal from the wait queue, and *leave with wake-up*:
  $t_{m-mutex-unlock} = t_{m-enter} + (t_{m-wq}) + t_{m-leave-with-wake-up}$
  $= t_{sys} + t_{sched-wake} + t_{m-wq} + (m-1) \cdot t_{CS} + 14$.

- A `cond_wait` operation comprises *enter*, insertion into the condition variable wait queue, removal of the next mutex owner from the mutex wait queue, a low-level wake operation to wake the next mutex owner, a *wait* operation on the condition variable, a spurious wake-up, removal of the caller from the condition variable wait queue, insertion into the mutex wait queue *wait* on the mutex, and a *leave* operation. This takes:
  $t_{m-cond-wait}$

$= t_{m-enter} + (2 \cdot t_{m-wq}) + t_{wake-set-prio} + t_{m-wait} + (2 \cdot t_{m-wq}) + t_{m-wait} + t_{m-leave}$
$= 4 \cdot t_{sys} + 2 \cdot t_{sched-wait} + t_{sched-wake} + +4 \cdot t_{m-wq} + 3(m-1) \cdot t_{CS} + 34.$

- A `cond_signal` operation comprises *enter*, removal from the condition variable wait queue, insertion into the mutex wait queue, and *leave*. As usual, we assume the mutex is properly locked during this operation. The worst case takes:
  $t_{m-cond-signal} = t_{m-enter} + (2 \cdot t_{m-wq}) + t_{m-leave}$
  $= t_{sys} + (m-1) \cdot t_{CS} + 2 \cdot t_{m-wq} + 10.$

  Support for "naked notifies" would change $t_{m-leave}$ into $t_{m-leave-with-wake-up}$ and add another $t_{sched-wake} + 4$.

  Another alternative is to clear the timeout, like in the other implementations of condition variables. For this, we introduce a *leave and clear timeout* operation based on a `clear_timeout_set_prio` system call, with similar costs as $t_{m-leave-with-wake-up}$ and $t_{wake-set-prio}$. This would add another $t_{timeout-clear} + 4$ to the baseline version.

- A *non-preemptive* `cond_broadcast` operation is similar to `cond_signal`, but handles all k threads on the condition variable wait queue. The worse case takes:
  $t_{m-cond-broadcast} = t_{m-enter} + (2k \cdot t_{m-wq}) + t_{m-leave}$
  $= t_{sys} + (m-1) \cdot t_{CS} + 2k \cdot t_{m-wq} + 10.$

  As alternative, we can also consider a *preemptive* variant of `cond_broadcast` that unlocks the spinlock, calls the kernel for a preemption point, and then locks the spinlock again. But then we should make it similar to the other variants and also include an operation to clear the timeout. We model this as k times $t_{m-cond-signal}$ plus the overhead to clear the timeout:
  $t_{m-cond-broadcast-tc} = k \cdot (t_{m-cond-signal} + t_{timeout-clear} + 4)$
  $= k \cdot t_{sys} + k \cdot t_{timeout-clear} + k \cdot (m-1) \cdot t_{CS} + 2k \cdot t_{m-wq} + 14k.$

We can now see that the non-preemptive version of `cond_broadcast` dominates the worst case of the spinlock-protected critical section. For n threads, this takes $t_{CS} = 2n \cdot t_{m-wq}$. When we now insert t_{CS} in each term, we see that the overhead depends on $m \cdot n \cdot \log n$ because of the missing preemptiveness.

But if we consider the preemptive version of `cond_broadcast`, then waiting and signaling dominate the worst case with *two* BST operations, and we see the overall overhead only depends on $m \cdot \log n$.

Comparison of Worst Cases

We now compare the different approaches for mutexes and condition variables.

For the comparison, we use the same conditions for all three approaches. This helps us to see the individual overheads. For deterministic futexes, we consider private, non-shared futexes as use case. This removes the interference of other partitions, as the baseline variant is also non-shared. For the monitor-based design, we consider the preemptive variant. With this, all three approaches use a BST for the wait queue and handle threads preemptively. The only remaining difference, besides the general approach, is the locking strategy for the wait queues. The baseline variant uses a dedicated lock for each wait queue. Deterministic futexes use a shared lock for all private wait queues of a process. The monitor-based approach uses a dedicated lock for each mutex and shares the lock for associated condition variables.

In all three approaches, we see the following general patterns. The number of internal wait and wake-up operations is the same in all variants. Mutexes always operate on one wait queue, while condition variables must handle two wait queues consistently. In general, all operations show the same complexity in all variants when considering the preemptive variants of `cond_broadcast`.

We start the discussion with the baseline design using system calls. We can observe that all operations (obviously) need exactly one system call. This version defines the minimum overhead of all three approaches. The overhead, i.e. the additional operations in each operation's term, comprises just $m \cdot t_{b-wq}$ for each wait queue operation including blocking overhead, with t_{b-wq} being of $\log n$ complexity.

In the operations of the futex-based synchronization mechanisms, we see as first speciality that `cond_wait` takes three system calls. This is because the kernel does neither provide automatic requeuing on timeouts as discussion in Section 4.3.2 nor a combined *wake-up-and-wait* primitive. But this is the only unusual overhead we can observe. The remaining overheads are for wait queue operations including any blocking with the same complexity of $m \cdot \log n$ as in the base line case and for a constant number of additional atomic operations. Also, the `cond_wait` operations in both the baseline and futexes see blocking of the wait queues five times. In the baseline, this is caused by the distinct locks per wait queue, in futexes due to the disadvantageous split between the system calls.

Overall, this shows that the deterministic futex design when using private and non-shared futexes has similar polynomial terms in the worst-case timing model with the same degree as the baseline variant. However, the actual logarithmic term is different due to the two nested BSTs. Also, in the case of shared futexes, the polynomial degree stays the same, but the logarithmic term now has to consider *all* N threads in a system.

Note that the discussion has evaded the topic about what happens if a semantic check in the kernel fails. For mutexes, the semantic checks in the kernel acquires a mutex for the calling thread *or* blocks the calling thread, therefore a *failure* does not cause retries. Likewise, a failed *compare-equal* check in the condition variable protocol means that the condition variable has been signaled, as discussed in Section 4.3.2. But this is not true for the other futex-based synchronization mechanisms of Section 4.3 like semaphores. We leave open how a detailed worst-case analysis would look like.

The last variant for discussion is the monitor-based synchronization design. Here we see a noticeable difference: `mutex_lock` needs two, `cond_wait` needs four, and `cond_broadcast` needs k system calls in the preemptive variant due to the much lower level of the system calls. Also, we see blocking in `cond_wait` only three times and just half of the blocking terms in `cond_signal` and `cond_broadcast`, as the internal lock protects both queues and the operations are not split into different system calls (a baseline implementation with a single lock protecting all queues would show a similar behavior). However, the complexity of the wait queue operations follows the same pattern as in the baseline and in futexes. Any remaining worst-case overhead comprises constant steps. Therefore, the *preemptive* monitor design has comparable worst-case terms with similar polynomial degrees as the baseline variant.

However, the monitor design is intended to be *non-preemptive* in the first place and the clearing of timeouts is also not included. This is on purpose, because the number of threads to requeue is usually small and a spurious wake-up of a requeued thread already waiting on the mutex has no impact on the correctness (the thread's position on the mutex wait queue remains unchanged) and the problem that timeouts trigger can be considered rare (a meaningful condition for a timeout in `cond_wait` is that the timeout should be much greater than the blocking time, i.e. $timeout \gg m \cdot t_{CS}$). Summarized, this design moves both pessimism (wait queue handling is non-preemptive) and optimism (the number of threads n is small and only depends on the current application) into user space.

5.4 Evaluation Summary

We now summarize and discuss the evaluation results.

General Approach

Our results show that both general design approaches for blocking synchronization mechanisms, namely futexes and monitors, are both efficient and predictable.

Our assessment of the different ARM processors in Section 5.1 has shown that our design approach to reduce or even get rid of system calls in the fast path is desirable not only on the x86 processor architecture. Reducing the number of system calls is a main driver for the performance improvements in the best, uncontended case, as the benchmarks for the efficient IPCP implementation in Section 5.2.2 and for futexes and monitors in Sections 5.2.3 and 5.2.3 show.

To evaluate the impact of the designs on the WCET, we presented a worst-case timing model based on cold-cache data accesses in Section 5.3.1 that shows differences in the design when an implementation inadvertently triggers worst-case allocation patterns in the cache. The benchmark results in Section 5.2.1 show trashing in the same sets in the caches due to these worst-case memory access patterns and a second problem that the memory accesses cause contention on shared resources further down the memory hierarchy.

Main Differences of Futex and Monitor Designs

We will now summarize the differences of the futex and monitor designs from both a best-case and a worst-case point of view.

In the best case, the overhead in the fast path of both the futex and the monitor variants is less than a system call, but the monitor shows more overhead than a futex. A futex fast path typically comprises one atomic operation with either acquire or release semantics or equivalently a memory barrier. The monitor fast path requires a non-preemptive critical section (load and store instructions on the local processor), and one or two atomic operations with both acquire and release semantics or equivalent memory barriers in the spinlock operations.

For the worst case, we assume that the fast paths are not taken. Then we see mostly similar costs compared to a baseline using system calls. But the actual place where these costs and overheads need to be accounted to differs and depends on the design. For futexes, the kernel needs to look-up a wait queue and then suspend the calling thread or wake up a waiting thread. This is similar to the baseline design. In contrast, the monitor design maintains the wait queue in user space and just suspends or wakes up threads in the kernel. Here, most of the in-kernel costs are moved to user space. However, the operations at the lowest layer to suspend the current thread or wake up a waiting thread are the same, regardless of the futex, monitor, or baseline design.

Deterministic Futexes

Deterministic futexes use two nested BSTs to firstly locate particular wait queues, and to secondly maintain the individual threads blocked on the wait queues. In both cases, the number of nodes in the BST grows dynamically depending on the number of blocked threads. From a performance point of view, the BSTs do not comprise many nodes on average and traversal is quick. Also, as threads can only wait at one wait queue at once, we can observe the interesting effect that maxing out the number of nodes on either BST does not lead to worst-case behavior. The worst cases is found when BSTs are equally filled.

In comparison to the Linux implementation analyzed by the author in [ZK19] and shown in Figure 3.6 in Section 3.5.3, the benchmarks of deterministic futexes in Section 5.2.1 show that using BSTs effectively bound the execution time and the interference by and to other partitions to logarithmic complexity, as expected.

The worst-case analyses in Section 5.3.2 have shown that deterministic futexes improve the worst-case interference of m processors and N overall threads in the system from $m \cdot N$ (hash-based wait queues, like in Linux) to $m \cdot \log N$. But the analyses have also shown that the overall execution time of operations that process more than one thread takes more time when using deterministic futexes due to the preemptible design.

The analyses of the higher-level synchronization mechanisms built on top of deterministic futexes of Section 5.3.3 also show a similar complexity as a baseline implementation using system calls, but a higher worst case and more system calls due to limitations of the futex API in case of spurious wake-ups compared to in-kernel implementations of blocking mechanisms.

Also, deterministic futexes address the problem of unbounded loops for atomic operations on futex values found in the Linux implementation. The presented design moves the overall problem from the kernel to user space, where we have argued that bounding the loops is only a problem of the application not behaving correctly when using mutexes and condition variables. We have left open how to handle this for other synchronization mechanisms, like the semaphores of Section 4.3.

Static Futexes

The analysis of static futexes in Section 5.3.2 with similar design decisions as in AUTOBEST shows that using a non-preemptible futex design due to its simplicity only pays off if all parameters are small bounded, as the worst-case overheads depend on $m \cdot n \cdot t_{scheduler-operation}$ and also include scheduler operations.

As the number of threads in statically configured systems is usually small and the scope of futexes in AUTOBEST is limited to a single processor core, the resulting worst-case is acceptable if these preconditions are met.

Monitor-Based Synchronization Mechanisms

The second general design approach using monitors moves any wait queue management into user space and provides minimal kernel mechanisms with constant overheads to the kernel's scheduling primitives, as the analyses in Section 5.3.2 show.

The general design approach to keep the number of system calls small when using monitors is to use a non-preemptive design for wait queue operations, but this can cause some hiccups, as for example spurious wake-ups after requeuing are not proactively prevented. However, this does not impact the correctness of the approach, as the position of a thread in a wait queue is independent of the waiting state in the kernel.

But due to the use of minimal primitives, we see a huge difference of the number of system calls between the best case and the worst case, especially if we compare the results to a baseline implementation of mutexes and condition variables based on system calls.

Also, a critical point in a WCET analysis is the non-preemptive critical section in user space. An in-kernel implementation can simply disable interrupts to achieve non-preemptiveness, however, this is not possible in user space. Therefore, in the WCET analysis, extra delays due to interrupt handling have to be accounted for.

The Uprio|Nprio Protocol

The Uprio|Nprio protocol for efficient IPCP shows the best results from both performance gains and worst-case point of view. The protocol never needs a system call to raise the priority, and only one when preemption is really needed. In the best case, the costs just comprise manipulation of protocol variables in TLS. In this regard, the protocol is better than any system-call-based implementation of IPCP or other mechanisms to change the scheduling priority. Also, the analysis of the underlying mechanisms in Section 5.3.2 shows that only a thin wrapper with constant overhead around the kernel's scheduler services is needed.

Chapter 6

Discussion

In this chapter, we will discuss the designs presented in Chapter 4 and the evaluation results of Chapter 5. We highlight the basic techniques of each design, and discuss practicality, limitations, challenges, and safety and security concerns. We discuss the designs in comparison with other state-of-the-art approaches.

Section 6.1 starts with a discussion on the exploration of the design space as framework for further discussions. This covers the aspects both common to monitors and futexes. Then we discuss the different aspects of futex-based designs in Section 6.2. Section 6.3 discusses the monitor-based approach for synchronization mechanism and the efficient IPCP protocols. Section 6.4 compares both design approaches. We also discuss the applicability of real-time protocols to the designs in Section 6.5. Finally, we discuss the contributions of this thesis in Section 6.6.

6.1 Design Space Exploration

We briefly summarize the design space of fast paths for blocking synchronization mechanisms in user space.

Fast paths and semantic state: The analysis in Section 3.4 of the building blocks of blocking synchronization mechanisms in a *baseline* implementation using system calls has shown that most thread synchronization mechanisms follow a similar blueprint in their implementation. At the lowest level, all mechanisms suspend the calling thread or wake up one or more waiting threads. To generalize this, we have defined a synchronization-mechanism-specific *semantic state* that needs to be checked before the decision to suspend or wake-up is taken. This semantic state also comprises one or more wait queues. The semantic state and the wait queue are usually protected by a specific lock.

Fine-grained locking in an operating system kernel often uses a different lock to protect ready queues. In Section 3.8, we have discussed techniques for coupling of consecutive critical sections without nesting them by using wait indicators. We also discussed different protocols for wait indicators. Alternatives to consecutive locking comprise using just a single lock for both critical section (global lock, scalability issues), or hand-over-hand locking (this inflates the WCET of the first (outer) critical section with the pessimism of the second (inner) critical section).

In this layered stack of operations, we have explored different ways to enable fast paths in user space. The analysis in Section 3.6 has shown one general approach to move the semantic state partly or fully into user space. We then explored different designs to exploit the fast paths in Chapter 4.

Consistency: However, the semantic state needs to be protected against concurrent modifications in user space as well. This requires an additional critical section in user space and a proper handover of the waiting state to the next consecutive critical section in the kernel using a wait indicator mechanism.

Futexes actually "compress" the semantic state into single variables and use atomic operations for consistency (Section 3.5.1). Due to the atomic operation, only a limited amount of data can be modified. In the futex designs we discussed, this is just a 32-bit variable, but it could be a larger state. However, futexes use lock-free algorithms to change the atomic variables. These require loops if the atomic operation does not succeed, as Section 4.3 shows.

As alternative to atomic operations, in Section 3.6 we discussed to use a "full" critical section to protect the semantic state in user space. For this, we intended to use spinlocks to serialize access from multiple processors. And for real-time applications, we especially wanted fair spinlocks. However, spinlocks are susceptible to the lock holder preemption problem, and fair spinlocks additionally to the lock waiter preemption problem. Since we target real-time systems and use P-FP scheduling, we can use IPCP to effectively disable preemption.

Using system calls to disable preemption would defeat the futex approach of moving the fast path into user space, therefore we discussed techniques to change the scheduling priority efficiently without system calls in Section 4.4.1. With this, we get non-preemptible critical sections in user space. We explored this approach as an alternative to using atomic operations in the monitor design presented in Section 4.5.

Wait queues: With the semantic state moved to user space, we then looked at the next logical layer, wait queue management. In Section 3.6, we discussed two general approaches to keep the wait queue in the kernel, or move the wait queue to user space as well. Wait queues in the kernel follow the baseline design. They require a mechanism to allocate and address related kernel objects. In contrast, wait queues in user space requires more sophisticated synchronization in user space to keep the wait queue consistent. They also require a mechanism to address particular (blocked) threads.

Wait queues in the kernel have the benefit that the kernel has implicit knowledge of the relation of synchronization objects and blocked threads. This information is not available when wait queues are kept in user space. A benefit of wait queues in user space is that the fast path code gets the implicit information whether a wait queue has waiting threads or not. Providing this information when the wait queue is in the kernel requires rigorous accounting in user space or a back-channel by the kernel.

The next logical layer in the design space below wait queue management is already the interaction with the scheduler to suspend and wake up threads. We decided to keep this layer in the kernel and not try to move it into user space, as this would require to depart from a traditional kernel architecture. It would be interesting to see if this part of the design space could be further exploited, e.g. by using user space schedulers like [ABLL92].

Futexes with wait queues in the kernel: Wait queues in the kernel raise two related questions: how is the related kernel object allocated, and how does user space address this kernel object.

For futexes in Linux, we see that a kernel object comprises pre-allocated hash buckets with shared wait queue heads (Section 3.5). In the deterministic futex design presented in Section 4.1, the wait queue heads are created on demand. Both designs create wait queues as logical kernel objects on demand without prior registration. This helps when the fast path in user space observes contention and requires kernel support for blocking. There is no a priori relation of user space objects to kernel objects. In contrast, in the static futex design of Section 4.2, the wait queues heads are created before use, e.g. at compile time. Also, each synchronization object in user space requires a related kernel object.

To address the wait queues in the kernel, Linux and the deterministic futex design address the in-kernel wait queue by the futex address in user space. This address is unique for each futex object in user space and is the key technique to create logical wait queues on demand. In the static futex design, user space code addresses the pre-allocated wait queue heads by their index, e.g. the position in an array. But there are more alternatives possible: our original design for deterministic futexes [Zue13] uses the thread ID of the first waiting thread to locate the in-kernel wait queue, assuming the kernel supports an array-based look-up of threads by IDs.

Monitors with wait queues in user space: We named the approach that keeps both the semantic state *and* the wait queue in user space a monitor. It does not require a scheme to directly or indirectly address a wait queue in the kernel, but addresses blocked threads by their thread ID directly. Also, the kernel operations are reduced to a minimum as described in Section 4.4.2 for the light-weight waiting and wake-up mechanisms. However, we added some extensions to the mechanisms to cooperate with the fast IPCP implementation described in Section 4.4.1.

Compared to futexes, the monitor design moves the semantics of the blocking mechanism completely into user space and avoids complexity and related determinism problems in the kernel. This comes at the costs that the kernel does not need to know anything about the protocols or the semantics of the synchronization mechanisms. But this also means that the kernel cannot know or do anything. User space must now handle everything.

Interference due to shared namespaces: Another important aspect that all designs must handle are interference problems when using shared namespaces. Both futexes in Linux and deterministic futexes distinguish between process-*private* and global-*shared* futexes. In Linux, both private and global namespaces share the same hashed wait queues, so one can observe interference by futex operations of unrelated processes (see Section 3.5.3).

In contrast, the deterministic futex design keeps private futexes specific to their process. Only global futexes can observe interference. Due to the global interference, we also opted for a design that is preemptive for operations processing more than one thread. This effectively bounds the interference in a predictable way. In the static futex design, any global interference is explicitly configured by the system configurator when connecting the wait and wake-up ends of the wait queues (see Section 4.2).

The monitor design similarly requires to have access to the thread IDs of all participating threads. This is the case for threads in the same process, but usually not given for threads in other processes. Giving applications global access to all threads in the system causes interference. Note that the presented design has a safeguard to prevent unwanted wake-ups by requiring the addresses of the local state variables in user space to match.

6.2 Futexes

We now discuss the two presented futex designs in detail. We discuss deterministic futexes in Section 6.2.1 and static futexes in Section 6.2.2. Section 6.2.3 discusses the practicality of futexes for implementing blocking synchronization mechanisms.

6.2.1 Deterministic Futexes

We discuss the deterministic futex design presented in Section 4.1 and evaluated in Section 5.2.1.

The discussion is partly taken from the author's previous work [ZK19].

Comparison to Futexes in Linux: The deterministic futex design follows the general design principles and the APIs of futexes in Linux as reference. In both cases, the kernel provides a general *compare-and-block* mechanism and a specialized mechanism for mutexes where the kernel specifies the protocol. The according higher-level synchronization mechanisms of Section 4.3 also follow the design of futex-based synchronization mechanisms in Linux.

The analysis of the Linux implementation in Section 3.5 clearly shows that it was designed for best case scenarios, e.g. only a small number of threads need to block, and collisions in the futex hash table are rare. This is *usually* the case during normal operation of a system. However, if one needs to determine upper bounds of the WCET, the corner cases in the Linux implementation lead to potentially unbounded execution time.

The deterministic futex design presented in Section 4.1 improves on the Linux design w.r.t. predictability, while maintaining a similar feature set. Due to the use of nested BSTs instead of hashes and linked lists, the deterministic futex design shows additional overhead with logarithmic complexity in all futex operations, as the evaluation in Section 5.2.1 shows. Also, the locking approach is restricted to a single shared lock, which is worse in the average case compared to the Linux implementation, as Linux uses a dedicated lock for each hash bucket, but at the same time, the deterministic futex design bounds the worst-case timing by preemptively processing multiple threads.

The deterministic futex design does not support all futex use cases available in Linux, as it handles either just *one* or *all* threads, and not an arbitrary number of threads. However, we do not consider this to be a problem, since typical implementations of POSIX synchronization mechanisms do not require operations on an arbitrary number of threads. Even if we consider that *some* mechanisms like the barriers discussed in Section 4.3.4 could benefit from a wake-up operation with a flexible number of waiters, preemption at any time and other race conditions require us to wake up *all* waiting threads to prevent missed wake-ups. With these limitations, it is unlikely that the deterministic futex design would be acceptable for inclusion in Linux.

Lack of support for the priority inheritance protocol (PIP): The deterministic futex design presented in this thesis does not support PIP for mutexes. Still, it would be possible to provide support for PIP and include any data to build a resource allocation graph in the wait queue anchor and in the thread's TCB. For priority inheritance in non-nested locks, tracking the highest priority blocked thread as source of the inherited priority is sufficient.

However, the *nested* and *transitive* case, where a set of threads block on a mutex, but the current lock holder is itself waiting on another mutex, is more complex. The nesting creates a *chain* of dependencies in the resource allocation graph, which can become very long (potentially unbounded) and must not contain cyclic dependencies (deadlocks). We assume that implementing support for PIP with nested locks with an arbitrary level of nesting and with proper deadlock detection would increase the temporal behavior of the `futex_lock` and `futex_unlock` operations beyond a level acceptable for predictability. But an implementation with a strictly bounded number of nesting levels would be possible. We consider this for future work.

From the predictability point of view, using the immediate priority ceiling protocol (IPCP) instead of PIP prevents this complexity, as then any nesting of critical sections must be handled by user space code. Also using IPCP enables further optimization, as the techniques discussed in Sections 3.7 and 4.4.1 to change a thread's scheduling priority in user space without the need for system calls show. This allows to prevent any system call overhead for critical sections without contention, similar to uncontended futex operations. However, this now burdens the user space programmer or system integrator to set up ceiling priorities of the mutexes correctly.

Preemptible operation: Implementing operations on multiple threads in a preemptible way also needs further discussion. In Linux, all operations handling multiple threads execute uninterruptibly w.r.t. other futex operations targeting the same futex, but the presented implementation does not. Preemption in these operations introduces a *sneak-in* problem, where threads can re-enter a wait queue while another thread operates on them. This may facilitate denial-of-service attacks on the kernel, as operations may never terminate. The presented approach with an explicit open/closed state for wait queues solves this, but it introduces the additional problem of multiple wait queues in closed state, which is solved with the drain ticket concept, as described in Section 4.1.5.

The question arises if it is in general acceptable to help out older, but still unfinished operations, i.e. wait queues with a lower drain ticket number. We can answer this question by considering the following usage constraint of condition variables: the caller of `cond_signal` and `cond_broadcast` shall have the support mutex locked as well, so none of the requeued threads will run before the caller unlocks the support mutex. Therefore, handling threads of a previous waiting round can only happen when `cond_signal` and `cond_broadcast` do not have the support mutex locked, and in this case, POSIX does no longer guarantee "predictable scheduling" [IEE17]. This means the answer to this question is yes, in accordance to *RQ9 (hidden transience)* of the requirements of Section 3.2.3.

A different use case is a barrier implementation like the one discussed in Section 4.3.4 where a given number of threads block until all threads have reached the barrier. Here, the implementation of `barrier_wait` uses `futex_wake` to wake all blocked threads. A preemptive `futex_wake` operation could get immediately preempted by a higher priority thread that is woken up as first thread and then the other threads are kept blocked until the original thread continues draining the wait queue. Note that this would not happen in a non-preemptible implementation. However, POSIX also notes that applications using barriers "may be subject to priority inversion" [IEE17], so this allows some freedom in the interpretation of the standard. Alternatively, an implementation can mitigate this issue by temporarily raising the caller's scheduling priority to a priority higher than the priorities of all blocked threads during wake-up. This could be implemented at user space level in the barrier implementation, or at kernel level in `futex_wake`.

6.2.2 Static Futexes

We now discuss *static futexes*, the futex design for statically configured systems presented in Section 4.2 and evaluated in Section 5.3.2.

Index-based futex design: A first idea of the static futex design is that the kernel knows all wait queues in advance. The wait queues could be pre-allocated at compile time, or created at initialization time of the user space synchronization. In all cases, this leads to an 1:1 relation of user space synchronization objects to kernel wait queues. The key advantage of this idea is that the kernel can now look up wait queues in $\mathcal{O}(1)$ time, rather than using a wait queue look-up based on futex user space addresses. This also solves problems of internal locking and of related interference. An alternative name for this approach is *index-based futex designs*. Such an approach is also used by Spliet et al. [SVBD14] in their work on futexes in LITMUS$^{\mathrm{RT}}$.

However, the downside of using an index-based design is a different API that uses an *index* rather the futex address. For the usual robustness, scalability, and interference reasons, futex arrays and according indices should be kept in a local per-process namespace. However, index numbers are problematic for process-shared synchronization objects placed in a shared memory, as the indexes can differ between the processes and indices kept in synchronization objects in user space might be ambiguous about the owning process. This restricts the flexibility of the design compared to the address-based futex designs that always have a unique key to identify the wait queue, namely the physical address of the futex object.

Split wait and wake-up ends: A second idea explored in the static futex design used in AUTOBEST is to split the waiting and the wake-up end of the wait queues and provide the ends to different processes or partitions. The original intention for this split was to provide a mechanism to configure shared futexes. Another intention for this was robustness: The wake-up side API requires just the process-local ID of the wait queue to perform a wake-up operation, so the futex value to manage threads on the waiting side can be kept outside the shared memory.

Non-preemptible design and static configuration: The rest of the static futex design follows the standard blueprint of futexes. The kernel provides a generic *compare-and-block* primitive for waiting. The design uses a non-preemptible design, as only *wake all* and *requeue all* operations need to process more than one thread. This design decision is only acceptable for a small number of blocked threads and if the overall number of threads is bound and known at compile or analysis time. The impact from the timing analysis also suggests to use such a non-preemptible design only for single processor systems, as the analysis in Section 5.3.2 shows.

6.2.3 Practicality of Futexes

We now discuss the *practicality* of the presented futex designs for the blocking synchronization mechanisms described in Section 2.2.

Futexes offer two APIs, a generic API that provides *compare-equal* semantics for the blocking condition, and a mutex API that implements a specific mutex protocol in both the blocking and wake-up operations. However, the pure existence of the mutex API already hints to a problem that the generic API is not well suited for mutexes. Also, the futex-based designs of the blocking synchronization mechanisms in Section 4.3 have also shown that the general API does not apply well in all cases, e.g. when handling spurious wake-ups.

Mutexes: Futexes provide a dedicated API and a specialized protocol only for mutexes, as Section 4.3.1 shows. The mutex protocol fits well to mutexes, as it encodes the different states of a mutex efficiently, namely the current lock owner by the thread ID and contention on the mutex by the `WAITERS` bit.

Condition variables: The protocol for condition variables presented in Section 4.3.2 shows spurious wake-ups when a condition variable is notified before a thread suspends itself in the kernel. The protocol for condition variables introduced in glibc 2.25 in Linux shows a solution for this particular problem.

Counting semaphores: The design of the counting semaphores of Section 4.3.3 shows lots of spurious wake-ups. The root cause of this is that the futex value encodes two independent counters atomically, and only one counter is relevant for the blocking condition in the kernel. The problems of spurious wake-ups could be avoided if the kernel would ignore parts of the futex value when evaluating the blocking condition.

Also, threads observing spurious wake-ups can *steal* resources from legitimate threads. This cannot be solved by the kernel, but by a different protocol that orders blocking threads differently.

Barriers: The barriers of Section 4.3.4 show similar problems with spurious wake-ups as the semaphores. This could be solved by a different API where the kernel only compares a part of the futex value.

One-time initializers: The design presented in Section 4.3.5 fits well to the generic futex API. One-time initializers are the only user space synchronization mechanism without any draw-backs.

ARINC 653 queuing ports and buffers: ARINC 653 queuing ports and buffers show the problem of spurious wake-ups, i.e. when a second waiter arrives and modifies the futex value while a first waiter is on its way to block in the kernel, and the problem of stealing as described for the counting semaphores. The protocol presented in Section 4.3.6 could benefit at least from the idea that the kernel evaluates only part of the futex value as blocking condition.

ARINC 653 blackboards and events: Both blackboards and events provide a kind of condition variable with *wake-up all* semantics if the condition is signaled. The same problems as for barriers appear here.

AUTOSAR eventmasks: We tried to implement a futex design for eventmasks in the context of AUTOBEST, however, the generic *compare-equal* condition often leads to unwanted spurious wake-ups. Also, the eventmasks require a condition check in the wake-up operation. For eventmasks, the kernel-based baseline implementation was the best option.

General Discussion: Summarized, we can observe the following problems when constructing blocking synchronization mechanisms besides mutexes from futexes:

- The *compare-equal* semantic check in the kernel is too unspecific for many futex protocols and causes most of the spurious wake-up problems.

  The best option to solve this would be if the user could specify a function to evaluate the futex value. This would allow the user to define an arbitrary blocking condition [BFC95,Piz16]. However, calling a user-provided function in user space requires to cross the user space $\leftrightarrow$ kernel space boundary again and stands in stark contrast to the idea of futexes to avoid the costs of crossing this boundary in the first place. Instead, this requires the kernel to run user defined code, like in Massalin's work on *Synthesis* [Mas92], or the user-provided *filter* programs in the *Berkeley Packet Filter (BPF)* [MJ93]. An alternative for the presented protocols could be if the compare operation is restricted to *parts* of the futex value. This would help the counting semaphores and barrier implementation to prevent spurious wake-ups.

- The semantic operation before blocking is a read-only comparison. However, a feedback mechanism (e.g. modification of the futex value) after successful evaluation of the blocking condition could be helpful in some protocols, similar to setting the `WAITERS` bit in the mutex protocol to indicate waiting threads. A generic write primitive is hard to specify, and a user-provided function would be the best option again. But a simple fallback mechanism like setting a `WAITERS` bit in the futex value could be already helpful. This would allow the counting semaphore implementation to detect if threads are currently blocked in the kernel, for example.
- After waking up a thread, only the kernel knows if the wait queue still contains other waiting threads. Again, a user-specific function that modifies the futex value accordingly would be the best option. Alternatively, the futex operation could clear the `WAITERS` bit in this case.
- Next to the semantic check for waiting, a semantic check for wake-up could be handy. This would help in the design of a futex protocol for eventmasks. A user-provided function to evaluate if a thread really needs to be woken up would solve this problem. Note this effectively moves the condition check from the waiting side to wake-up side.
- The problem that any changes to the futex value requires atomic operations in the kernel, and loops with atomic operation must eventually terminate.

Note that the previously discussed ideas are not new. The missing information whether there are still threads blocked on a futex is a serious problem for the design of user space protocols and a source of superfluous system calls in glibc in Linux. Buhr et al. discuss similar problems for monitors [BFC95]. Pizlo solves some of these problems in the implementation of fine-grained locking in the WebKit browser [Piz16] by providing user-specific callbacks to evaluate the blocking condition and during wake-up.

Also, we have observed the problem of “stealing” an according event in some protocols like the counting semaphores. A thread frees a resource and wakes up the next waiting thread, but a third thread acquires the resource in the mean time before the woken up thread is scheduled. These are side effects similar to the TOWTTOS problem discussed in Section 3.4.2.

In general, futex protocols should be designed in such a way that TOWTTOS problems are minimized. The mutex protocol shows a possible solution. A thread waiting for a mutex could time out, but may become the mutex owner *before* eventually being scheduled. This requires a releasing thread to hand over the resource to the next waiting thread instead of letting acquiring threads compete for a resource. This increases the robustness of the protocols, but at the same time limits its scalability. This is a classical trade-off between fairness and scalability a system implementer must consider.

6.3 Monitors

We discuss the monitor design presented in this thesis. Section 6.3.1 discusses efficient IPCP protocols and Section 6.3.2 discusses the monitor approach. In Section 6.3.3, we discuss the practicality of the presented monitor design for implementing blocking synchronization mechanisms.

6.3.1 Efficient IPCP Protocols

We discuss the Uprio|Nprio protocol for fast IPCP presented in Section 4.4.1. The discussion is adapted from previous work of the author [ZBK14, Zue20].

General discussion: The Uprio|Nprio protocol shows the following behavior: no system call is needed to raise the priority, but a restore operation might require a system call. The system call is only necessary when preemption is really needed, i.e. when a new thread with a priority above the base priority and below or equal the elevated priority of the thread in using the IPCP protocol became ready. In practice, this makes the protocol close to optimal in the number of required system calls for the uses cases IPCP, especially when using nested critical sections or in our monitor use case when waking up threads.

Note that the protocol requires a kernel that is optimized for frequent priority changes and does not keep the current thread on ready queues, so `nprio` is naturally available from the highest priority thread on the ready queue and also used internally by the kernel to decide whether to preempt the current thread.

The protocol requires just two protocol variables shared between user space and the kernel. However, due to this simplicity, the protocol exposes a short race condition. Recall the implementation of `prio_restore` in Section 4.4.1:

Listing 6.1: Restore previous scheduling priority

```
1 void prio_restore(prio_t prev)
2 {
3     SELF->uprio = prev;
4     if (SELF->uprio < SELF->nprio) {
5         sys_preempt();
6     }
7 }
```

The system call for preemption in line 5 would be superfluous if the thread is preempted after updating `uprio` in line 3 but before calling `sys_preempt` in line 5. Almatary et al. solve this corner case at the expense of additional protocol variables [AAB15]. However, the Uprio|Nprio protocol shares this behavior with the preemption control mechanisms in the Linux kernel discussed in Section 3.7.2 and the Symunix II protocol [ELS88]. In their work on futexes in LITMUS^{RT}, Spliet et al. also observed a similar race condition in the `unlock` operation of their

PCP-DU-PF protocol, but provide no solution [SVBD14]. Still, the race condition does not impact the worst case; one system call is always needed.

The benchmark results in Section 5.2.2 shows that the Uprio|Nprio protocol behaves as expected. The overall performance gains look very promising, and the protocol needs at most one system call when lowering the priority. However, the benchmark results of a critical section followed by a *sys_preempt* system call in Table 5.2 show that the worst-case timing is still in a similar range of the pure system-call-based approach. We assume this effect has prevented the adoption of such protocols for general purpose operating system.

At this point, it becomes clear that further benefits of the protocol are difficult to analyze by using microbenchmarks. We need to run real-world workloads or require statistical information on the distribution of nested and non-nested locking and typical preemption patterns to see the overall effect on a long-running system.

Safety and security considerations: From a safety point of view, the following aspects are relevant. A thread T_i can try to exceed its maximum controlled priority π_i^{max} by placing a higher priority value into `uprio`. The kernel must check this whenever it reads `uprio` and must bound the value to π_i^{max}. Additionally, it is possible to enforce a lower priority bound π_i^{min} in an implementation, should that be a requirement. Lastly, a thread can act as a foul player and not issue a system call on lowering the priority. This behavior has the same effect as a thread not leaving the critical section, because it also delays the scheduling of higher priority threads. This problem is not introduced by the fast priority switching approach: it would also happen with the baseline approach using system calls. Threads accessing the same resource must mutually trust each other anyway. We consider this to be a programming error and not a side effect of the protocol.

From a security point of view, the Uprio|Nprio protocol can leak scheduling information of unrelated processes if processes are not temporally isolated, e.g. by a TDMA scheme like in ARINC 653. In the used protocol, `nprio` exposes the priority of the next eligible thread for scheduling on the ready queue to other processes. This may hinder its use in security sensitive operation environments. The other IPCP implementations discussed in Section 3.7 are also problematic, as threads can observe that they were interrupted and preemption becomes pending.

6.3.2 Light-Weight Monitors

We now discuss the monitor approach presented in Section 4.4.2. The discussions is taken from the author's previous work [Zue20].

General discussion: The presented light-weight monitor design of Section 4.4.3 comprises the Uprio|Nprio protocol, fair spinlocks, and the low-level wait and wake-up mechanisms presented in Section 3.8. The waiting primitive interacts with the UPRIO|NPRIO protocol by letting a thread suspend on a lower priority, but do not touch the protocol variables, so the thread is immediately set back to its previous elevated scheduling priority after wake-up. The key technique for the wake-up operation is to fuse an additional priority change operation into the same system call.

In general, the evaluation in Section 5.2.3 shows that the monitor approach works and saves CPU cycles by avoiding system calls in the uncontended case. The monitor has similar properties as futexes. Synchronization mechanisms built on top of monitors do not need initial registration in the kernel, and therefore also no resources or memory allocations in the kernel. The monitors are better than futexes w.r.t. predictability, as they often lack the loops futex-based solutions show (see Section 4.3), but they also come with more overhead due to the IPCP implementation and spinlocks to protect internal critical sections. But unlike futexes (including the deterministic futex design), the presented monitor approach requires a real-time scheduler. In our presented case, we need P-FP scheduling for the IPCP implementation to work. Alternatively, one could also use DFP instead of IPCP when using P-EDF scheduling [AAB15], however we have not yet explored according optimized wait and wake-up mechanisms for this.

Due to direct addressing of threads, the monitor approach is typically limited to synchronization of threads in the same process, as mentioned in Section 6.1. However, when a system allows access to threads in other processes, then the monitor approach can also be used for shared memory communication, like futexes. In this case, a thread's waiting state variable (`ustate`) must be placed in the shared memory as well. Note that the robustness considerations here are the same as when using futexes or other synchronization mechanisms, as synchronization over shared memory requires that applications must trust each other. But typically, access to threads in other processes is a source of unwanted interference and therefore not allowed.

We also expect that threads behave correctly and use the protocols appropriately, but the impact of misbehavior on other processes is bounded by the maximum priority of a thread. The safety and security considerations for the monitor are the same as for the efficient implementation of IPCP.

WCET considerations: From a WCET point of view, the monitor approach reduces predictability issues compared to futexes. As the monitor building blocks are similar to a baseline version but just shifted in place, the WCET considerations are similar for both. The monitor adds additional constant overheads for the extra system calls for preemption. Also, only the wake-up of one thread is optimized and interacts nicely with IPCP. Waking up an additional thread needs one additional system call each. As mitigation, batching techniques could be used for example to wake up multiple threads. However, we assume that the wake-up of more than one thread is rare, but this again requires deeper knowledge of the application.

6.3.3 Practicality of Monitors

We discuss the practicality of the monitor-based designs of Section 4.5 to implement higher-level synchronization mechanisms of Section 2.2.

Compared to the practicality of futexes as discussed in Section 6.2.3, the monitor design already solves most of the problems by moving the effective blocking and wake-up condition into the critical section in user space. For the kernel, only a simple eventcount-like protocol to prevent both missed wake-ups and spurious wake-ups remains, as the evaluation in Section 5.3.2 shows.

Note that we have only implemented blocking mutexes and condition variables using the light-weight monitor design, however, we can still discuss potential problems for the other synchronization mechanisms.

Mutexes: The presented mutex design of Section 4.5.2 work well with the monitor. However, the extra critical section after waiting is costly. As optimization, a fast path could be introduced to check the mutex owner after waiting and then skip the critical section. However, this would also require a change in the unlock operation with an according memory barrier to *first* increment the sequence in `ustate`, and *then* hand over the mutex ownership, as the wake-up side must complete the increment the sequence for the protocol to work.

Using an API without support for a timeout (just infinite timeouts, no other spurious wake-ups by the kernel) could also help to simplify the protocol and remove the second critical section after waiting, as the example in Section 4.4.2 shows.

Condition variables: The condition variables of Section 4.5.3 also require the extra critical section after waiting. As `cond_wait` comprises both a `mutex_lock` and `mutex_unlock` operation, the same fast path as discussed for the mutexes could be implemented here as well.

One specific problem becomes visible: when a thread waits on the condition variable with a timeout, and the condition variable is notified and moved to the mutex wait queue, the kernel will still expire the timeout and will cause a spurious wake-up while waiting on the mutex.

Another problem is that a `cond_broadcast` requires one system call for each thread.

Low-level monitor API: The design presented in Section 4.5.4 shows a specific downside: notification of more than one thread requires a wake-up system call *inside* the spinlock-protected critical section. The general problem that one system call is required to wake up each thread is shared with the other condition variable design.

Counting semaphores: We do not expect any problems in an implementation of counting semaphores. The complexity will be similar to the mutex implementation.

Barriers: We do not expect any problems here either, however, barriers must wake up multiple threads at once, so an implementation would be inefficient compared to a futex-based approach, which can use a single `futex_wake` system call.

One-time initializers: The argument is similar to barriers, however, contention on one-time initializers is expected to be rare. But another problem becomes visible: higher memory usage as futexes. An internal lock and a wait queue comprise more memory than the 32-bit futex value.

ARINC 653 queuing ports and buffers: The design should work well for the produce-consumer-patterns in queuing ports and buffers.

ARINC 653 blackboards and events: Like in barriers, we require multiple system calls to wake up all waiting threads.

AUTOSAR eventmasks: An implementation using a monitor would be similar to a baseline implementation inside an operating system kernel.

General Discussion: We observe the following problems in the monitor design:

- A wake-up operation only processes one thread at a time. This is a natural problem of keeping the wait queue in user space. As already mentioned in Section 6.3.2, batching techniques could help here.
- Any previously set timeout remains active in the kernel, and a thread may observe a spurious wake-up when the timeout expires while already waiting on a mutex in `cond_wait`. This is also a problem of keeping the wait queue in user space. This problem could be solved by providing a specific system call to clear all pending timeouts, i.e. let an already waiting thread wait infinitely. Another option to solve this problem is that, when the timeout expires, the kernel checks if the timeout is still applicable. This, for example, could be realized by an additional flag in user space that is cleared during requeue operations, or by checking if `ustate` was modified since the thread started waiting. However, this requires a probably costly access to user space in the kernel's timer handling code. We leave this to be evaluated in future work.

- Another problem is the extra critical section after waiting to clean up the wait queue in case of spurious wake-ups. This is again a problem of keeping the wait queue in user space. As mitigation, we could implement some fast path, however these shortcuts require additional memory barriers to ensure the correctness of the protocols.
- Deletion of threads is a problem. Consider that a thread is deleted while waiting on the wait queue. The thread's memory must not be freed until the thread was removed from the wait queue, as the wait queue nodes are kept in memory allocated to the thread. A potential mitigation is to use *cancellation handlers* as described in POSIX, i.e. `pthread_cleanup_push` and `pthread_cleanup_pop`, and remove the thread from the wait queue before deleting the thread.

The presented problems mostly relate to the fact that the wait queue is managed in user space. This is both advantage and disadvantage of the presented monitor approach.

6.4 Comparison of Futexes and Monitors

In this thesis, we have explored and discussed two general designs for predictable and efficient synchronization, namely futexes and light-weight spinning monitors. However, there are still open problems with the presented designs.

Futex-based blocking synchronization mechanisms use atomic operations on the futex word to decide whether the fast path operation succeeds or if a thread needs to take the slow path and must block in the kernel. However, due to the compare-equal mechanism in the kernel, blocking is prevented if the futex value changed in the mean time. In this case, the futex operation must be repeated. With this, futexes behave like lock-free algorithms that must also repeat an operation if the atomic transaction cannot be completed. But as the loops to fulfill the blocking condition may be potentially unbounded, futexes are not wait-free. This problem is mainly caused by the interaction of the compare-equal condition for blocking (originally designed to prevent missed wake-ups) and the atomic protocols on the futex variables. Futexes only work well for eventcount-like protocols that do not encode any other information in the futex word.

Another problem we can observe with futexes is that even when using FIFO-ordered wait queues, the exact temporal order of waiting threads can get lost due to spurious wake-ups, as the time of the *decision to wait*, i.e. when reading the futex value, does not necessarily correlate with the time a thread eventually blocks in the kernel and is added to the wait queue.

Related to the previous problem, we have the problem of "stealing" of events or resources in futex-based protocols due to the delay between wake-up and scheduling due to the TOWTTOS problem discussed in Section 3.4.2.

Both effects are caused by the split of a former single semantic operation into two in the fast path design and the resulting delays between a first semantic operation in user space and a second semantic operation and wait queue operation in the kernel.

Another problem is accounting of resources. For example, the ARINC 653 standard requires that a conforming implementation must be able to report the number of currently waiting threads. However, this is hard to achieve with futexes. When user space code accounts the resources, e.g. a thread increments a waiters counter while blocking, the number will no longer be correct when the thread was woken up by a timeout, but is not yet scheduled again to decrease the counter. The TOWTTOS problem leads to overreporting in this case. Similarly, the number of waiting threads in the kernel does not reflect the number of threads that *intended* to wait (the threads could be still in user space) and will experience a spurious wake-up when the futex value changes. This leads to underreporting. For scalability reasons, system designers should not include APIs with unnecessary preciseness into the specification [CKZ+13].

Related to this is the missing feedback mechanism on the state of the wait queue (empty or not empty) in the kernel. The missing information whether there are still threads blocked on a futex is a serious problem for the design of user space protocols.

In contrast to the futex-based designs, the light-weight monitor comprises a different design approach that tries to prevent most of the predictability issues of the futex protocols. By using a spinlock-protected critical section in user space, the monitor prevents most of the subtle race conditions in the atomic protocols of the futex design and at the same time enables a richer semantics. Also, the building blocks are simpler, i.e. no complex look-up mechanism for wait queues is needed in the kernel. However, moving the wait queue into user space is not a panacea, as Section 6.3.3 shows. But it solves the previously mentioned problems of exact temporal ordering, stealing of events, and underreporting of blocked threads, as the wait queue is kept consistent in the first semantic operation.

The biggest weakness of the monitor approach is that for resource sharing between processes, the thread IDs of all participating threads must be accessible by all the participating processes. Compared to futexes, it seems that we have simply moved the source of interference to the thread look-up mechanisms now.

But this is not unexpected: any kind of resource sharing that uses a shared global namespace (for futexes, this is the physical address of a futex value), will expose this problem. Only the use of local namespaces, e.g. capabilities, can prevent this problem. However, this also requires a careful application design and the clear identification of the points of sharing, leading to an *anticipatory* design, if we stick to the categorization of Spliet et al. [SVBD14] But there are often use cases where this is not possible, like in best-effort software that just “has to work”. In these cases, the deterministic futex approach seems to be a better fit, as it restricts the shared global namespace just to wait queues and not to all threads.

6.5 Fast Paths and Real-Time Protocols

We discuss the applicability of the real-time protocols analyzed in Section 3.3 to the presented designs.

So far, we have only shown how to use NPCS and IPCP in the designs. These two protocols are necessary building blocks for the monitor design to achieve non-preemptible critical sections in user space and prevent lock waiter and lock holder preemption problems. For futexes, the fast IPCP implementation of Section 4.4.1 is a good match because both mechanisms avoid system calls in their fast paths.

We briefly discussed PIP in the context of futexes, as it is a requirement for POSIX and supported by Linux. PIP as a *reactive* protocol works well with futexes, as the Linux design shows. This observation also applies to any form of PIP on a multiprocessor systems that includes migration of preempted threads. However, we have not yet developed a PIP design for deterministic futexes. We leave this to future work.

In the monitor design, PIP is impossible to implement at kernel level, because the kernel has not enough information to build up a proper resource allocation graph due to the lack of access to the wait queues and because the kernel does not know the specific semantics of the blocking mechanism in user space. Supporting PIP would require a mechanism where user space code tells the kernel about changes in the resource allocation graph. As any increase of a lock holder's inherited priority naturally requires that other threads must block in the kernel, this information could be provided in the system call to suspend a thread. But the other way around, e.g. decreasing the inherited priority of the lock holder after a spurious wake-up of a high-priority blocked thread, does not expose implicit system calls to hook. We also leave it to future work to investigate this further.

MrsP combines FIFO-spinlocks with IPCP. When a spinning thread detects that the lock holder is currently preempted, it migrates the lock holder to its CPU. From an implementation point of view, support for MrsP would require three mechanisms: (i) detection that a lock holder is preempted, (ii) migration of a preempted lock holder to the current processor, and (iii) detection that a lock holder was migrated. The first two mechanisms are needed by the spinning side to detect and migrate a lock holder to the spinning processor. The third mechanism is needed by the migrated thread to detect migration and migrate back to its original processor. The necessary information of the first and the third mechanism must be available without doing system calls to be exploitable in a fast path. See also the related discussion on *preemption recovery* in Section 2.1.3.

DPCP requires upfront migration to a resource's bound processor. This protocol does not play well with fast paths, as migration requires a system call.

Lastly, MPCP and FMLP$^+$ are remaining candidates that could work well with a futex fast path, similar to IPCP. We have not further evaluated these protocols, however, Spliet et al. evaluated a fast path implementation of these protocols [SVBD14].

6.6 Discussion of Contributions

For this thesis, our goal was to achieve both efficiency and predictability for thread synchronization mechanism in the context of mixed-criticality systems.

State-of-the-art mechanisms to construct efficient synchronization mechanisms in non-real-time systems are futexes for blocking synchronization in Linux, and techniques for preemption control in user space to prevent lock holder and lock waiter preemption. These mechanisms are *building blocks* for efficient higher-level synchronization mechanisms in best-effort systems, and in both cases the key technique for good average performance is to avoid expensive system calls.

Our general approach was to (i) *analyze* existing efficient design approaches for their key techniques for efficiency and also to identify predictability issues, (ii) *improve* existing approaches w.r.t. predictability, (iii) *create* new approaches that address both efficiency and predictability, and (iv) *evaluate* the approaches by performance measurements and analyzes of their worst-case timing behavior.

Based on these efficient and predictable building blocks, this thesis provides both efficient and predictable user-facing thread synchronization mechanisms in the context of POSIX, ARINC 653, and AUTOSAR that are the main programming interfaces in industrial, avionics, and automotive domains.

We now discuss the five main contributions of this thesis claimed in Section 1.3 and also the practicality of the approaches for the synthesis of higher-level synchronization mechanisms. We show what we have achieved to extend the state-of-the-art for both efficient and predictable thread synchronization mechanism in the context of mixed-criticality systems.

Analysis of synchronization mechanisms: The key technique for good average performance in state-of-the-art synchronization mechanisms in non-real-time systems is to avoid expensive system calls (Section 3.1), as system calls comprise an expensive overhead (Section 3.2.2).

As a novel approach towards the construction of efficient synchronization mechanisms, we first generalized (Section 3.4.1) and then decomposed (Section 3.4.2) existing synchronization mechanisms into their low-level building blocks with the goal to identify potential fast paths.

We also identified that the fast paths in user space have a relationship to techniques and problems when using fine-grained locking. In particular, we discussed *wait indicator* mechanisms for loosely-coupled consecutive critical sections to transport state information from one critical section to the next one without nesting of locks. We have also identified a novel effect we named the TOWTTOS problem (Section 3.4.2) and that it is inherent to all systems using fine-grained locking and coupling of critical sections in the described way. We are not aware about any further research on these effects in systems using fine-grained locking, but the problem itself must been known for a longer time by practitioners, as the Linux kernel deals with this problem properly.

We then analyzed the benefits of placing these low-level building blocks at different places above or below the boundary between user space and kernel space (Section 3.6). Moving the boundary between user space and kernel space and redefining the responsibility of user space and the kernel is a usual approach in systems design, as the vast literature on microkernels, exokernels, virtual machine monitors, etc. shows. However, this is a novel approach in the context of real-time systems.

Deterministic futexes: Futexes were conceived to improve the average-case performance of blocking synchronization mechanisms, but not with real-time systems and requirements for predictability in mind. Deterministic futexes address the predictability issues found in futexes in Linux (Section 3.5.3) with an improved and novel design.

The design of deterministic futexes (Section 4.1) keeps a similar API and feature-set with only minor modifications that exclude use cases we deemed not necessary for the synchronization mechanisms we considered. This is a challenging task due to the flexibility of futexes and the interference channels when futexes are used in shared memory segments.

The key technique for deterministic futexes to increase the predictability is to use two self-balancing binary search trees (BSTs), which bound internal operations to logarithmic complexity, and to use a preemptible design for operations handling more than one thread.

Regarding efficiency, our deterministic futexes and Linux futexes are comparable, as they use the same fast path mechanisms (Section 5.2.1). From a worst-case point of view, both logarithmic complexity and the preemptible design especially bounds any interference when sharing synchronization objects between different processes or partitions (Section 5.3.2). As down-side, the overall execution time of long-running operations increases. We deem this to be acceptable for a mechanism that can be shared between different processes or partitions.

The design shows a huge improvement over Linux w.r.t. predictability. Related work comprises proposals by Gleixner to pre-allocate futex wait queues in Linux [Bro16] (this would require changes to the futex API), and the author's own previous work [Zue13].

As open points for future research, we have identified the missing support for the priority inheritance protocol (PIP) and the impact of changes to the *compare-equal* semantic check of futexes (Section 6.2.1)

Static futexes: Our second futex design, static futexes, relax the property that the kernel must create wait queues of futexes on demand (Section 4.2). This allows us to allocate all in-kernel data structures upfront and use a simpler *index-based scheme* to address wait queues in the kernel. Still, static futexes keep the overall interface characteristics of futexes, allowing to reuse existing user space futex protocols with minor adaptions.

We originally intended static futexes as a novel way to manage blocking synchronization in resource-constrained embedded systems where all resources are known at compile time. Our design restrictions for AUTOBEST [ZK19] focus on tailoring futexes for ARINC 653 use cases only (Section 4.2.2). We also proposed a novel technique to split access to wait and wake-up operations as mechanism for sharing resources between temporally decoupled threads like in systems using ARINC 653 time partitioning.

Our evaluation of the static futex design uses the design limitations in AUTOBEST (Section 5.3.2). This mainly shows the impact of the choice to use a *non-preemptive design* in comparison to deterministic futexes. This limitation re-introduces scalability and predictability issues, but also shows that the approach is suitable for single-processor environments and systems with few waiting threads (Section 6.2.2).

Therefore, we consider the static futex design in AUTOBEST just to be a representative of a stripped-down futex design where the discussed trade-offs between performance and predictability are acceptable. But static futexes can provide the full feature-set as Linux futexes or deterministic futexes, and can then be used in capability-based kernel designs, such as L4-based microkernels [EH13].

Index-based futex designs were also discussed by Spliet et al. [SVBD14] in their work on futexes in LITMUS$^{\text{RT}}$, but without a focus on resource-constrained systems or ARINC 653 in mind.

For the future, we expect more research on index-based futex designs in the context of virtualization of legacy RTOS APIs, as futexes allow to keep similar average performance characteristics as when using library operating systems, and the index-based designs pose the least risks compared to other futex approaches.

Fast priority switching: The automotive standards OSEK OS and AUTOSAR OS use the immediate priority ceiling protocol (IPCP) for mutual exclusion in single processor environments. In other contexts, IPCP helps to bound lock waiter and lock holder preemption. As key technique, IPCP temporarily raises the scheduling priority of the currently executing thread during a critical section.

We presented two novel efficient mechanisms to indicate priority changes in variables shared between kernel and user space (Section 4.4.1). Both mechanisms avoid system calls to raise the scheduling priority, and need system calls to restore the original scheduling priority only in the worst case. The UPRIO|KPRIO protocol needs a system call only when the kernel has *observed* an elevated priority. The UPRIO|NPRIO protocol improves on this and needs a system call only when the current thread really needs to be preempted.

The performance evaluation shows efficiency gains (Section 5.2.2), and the mechanisms show minimal constant overhead to manage protocol variables and only require one system call in the worst case (Section 5.3.2). Of all the presented designs, the IPCP mechanism is probably the least disputable one, as it achieves both the greatest performance gains and has the least impact on worst-case timing and predictability.

The presented mechanism extends the related work on preemption control mechanisms (Section 3.7.1), and in particular Edler et al.'s preemption control mechanism for *Symunix II* [ELS88] for fixed-priority scheduling. To the best of our knowledge, the author's workshop paper [ZBK14] was the first to address the problem of efficient priority changes and to present a near optimal solution with the UPRIO|NPRIO protocol. The related approach by Almatary et al. [AAB15] uses a different protocol and does not handle nested locking scenarios as well as the UPRIO|NPRIO protocol.

For future research, an open question remains if such approaches also work for schedulers using more than one level, as the key technique here is to export information like in *Benno scheduling* [BSC+11, BSH12] to user space, which is tied to priority scheduling.

Non-preemptive busy-waiting monitors: Next to futexes, our second approach to blocking synchronization mechanism is the presented monitor design.

A novel synthesis of the building blocks takes the ideas of futexes to the extreme and also moves the wait queue abstraction into user space (Section 3.6). The resulting monitor design combines different techniques that avoid system calls for efficiency and effectively moves software layer previously kept at kernel level into user space. As far as we know, such an approach was not previously discussed by the research community, neither in the context of best-effort systems nor real-time systems.

This requires to use critical sections with predictable timing behavior in user space, which we achieve by our efficient IPCP mechanism for preemption control, fair spinlocks for mutual exclusion, and minimal wait and wake-up primitives in the kernel. The necessary system calls for suspension and wake-up interact with the IPCP mechanism for both efficiency and to emulate the behavior of traditional blocking synchronization mechanism (Section 4.4.2).

The resulting monitor primitives (Section 4.4.3) are a completely new design approach that allows to safely compose arbitrary blocking synchronization mechanisms in user space (Section 4.5).

Our evaluation shows that the monitors, when implementing blocking mutexes, are twice as slow than futexes in the average case due to two internal critical sections (Section 5.2.3), but more predictable at both kernel and user space level (Section 5.3.3). The primitives that are needed in the kernel show constant overhead on the general scheduler primitives of the kernel (Section 5.3.2).

Compared to futexes, the monitor approach helps to get rid of the complexity of wait queue handling at kernel level (Section 3.5.3). The monitors also address the limitations of futexes that are caused by the use of protocols based on atomic operations on 32-bit integers in user space and the limited flexibility of the *compare-equal* semantic check for blocking in the kernel (Section 6.2.3). With this, the monitor design effectively solves the consistency problems found in futexes by moving both the fast-path decision and the wait queue handling into the same critical section into user space and provides a more robust mechanism to suspend and wake up threads.

In the context of other monitor designs and implementations [BFC95], the presented design is comparable to the canon on efficient monitor implementations in language runtimes (see e.g. [BKMS98, KP98, Kaw05, Piz16]), but with a strong focus on real-time use cases.

Overall, the complexity of a synchronization mechanism based on monitors is similar to the complexity of a baseline mechanism based on system calls. Our analysis shows this, as the monitor design just moves software layers that exist in the kernel into user space. However, two differences remain: in the worst case, the monitor requires multiple system calls where a baseline only needs one system call; and an in-kernel implementation can simply disable interrupts to achieve non-preemptiveness and bound the execution time at the expense of interrupt latency. This is not possible in user space and needs to be accounted in a WCET analysis.

The monitor design shows open points for further research, especially when considering the limitations that the low-level kernel mechanisms only process one waiting thread and the open research points how to extend the monitor approach with real-time locking protocols for priority inheritance instead of priority ceiling.

Higher-level synchronization mechanisms: Lastly, we must discuss how the presented approaches and building blocks help to improve the efficiency of higher-level synchronization mechanisms in the context of POSIX, ARINC 653, and AUTOSAR.

Based on futexes, we presented blocking mutexes, condition variables, semaphores, barriers, and ARINC 653-specific communication port mechanisms (Section 4.3). This covers the common cases of the specific user-facing synchronization mechanisms for both industrial (POSIX) and avionic (ARINC 653) applications. Futexes are well known and used in the context of Linux as the *only* mechanism for user space synchronization mechanisms. But using futexes is a novel approach to construct synchronization mechanisms in avionic systems.

Based on monitors, we presented only blocking mutexes, condition variables, and the low-level monitor primitives (Section 4.5). We argue here that the higher-level synchronization mechanisms we discussed for futexes can also be implemented based on monitors, but with much less effort, as an implementation can use simpler low-level monitor primitives instead of defining protocols on atomic variables. With this, the monitors cover the use cases of both industrial and avionic applications as well.

Both futexes and monitors now allow to improve the average execution time of *uncontended* synchronization, as our evaluation on mutex performance shows (Section 5.2.3). At the same time, we pay for the performance gains in the uncontended case with overheads in the contended case. Like other optimizations, e.g. write-back caches, the presented designs pose a trade-off between performance gains in typical average use case scenarios and performance losses in pathological and non-typical corner cases. Unfortunately, the designers of real-time systems must account for the worst case when they cannot exclude the bad cases by other means.

Note that the thesis leaves the point open to show how much the real performance gains for real applications would be, as the actual benefits depend on the degree of contention (Section 5.2.3). Nevertheless, the performance gains in the uncontended case underline the practicality and usefulness of the approaches.

For blocking synchronization mechanisms, futexes are the fastest, but not the most predictable choice. Our presented deterministic futex design improves the determinism issues at kernel level compared to the Linux implementation, but does not address the issues of futexes at user space level for conceptual compatibility to Linux (Section 6.2.1). All futex-based designs show inherent problems and limitations caused by the temporal decoupling of the atomic operation in user space and the kernel's blocking mechanism using a *compare-and-block* mechanism (Section 6.2.3). However, the atomicity of the futex protocols becomes handy for shared memory synchronization when one cannot make assumptions on the progress of others.

Monitors are a better choice for synchronization mechanisms w.r.t. predictability at both user space and kernel level (Section 6.3.2), but effectively limited to synchronization jobs in the same process (Section 6.3.3). The monitor approach also shows limitations to process multiple threads and that the kernel cannot track the blocking dependencies between threads. Time will tell how the monitor approach will evolve.

Note that the futex-based and the monitor-based designs ultimately focus on blocking synchronization mechanisms in industrial and avionics use cases, but not automotive systems. Both OSEK OS and AUTOSAR OS use eventmasks for blocking synchronization, and the presented futex and monitor designs seem over-engineered, as they provide more features than actually needed. Therefore, we did not provide any higher-level blocking synchronization mechanism. However, the automotive APIs also require the use of IPCP for resource management on a single processor. And with our efficient IPCP protocol, we also provide an approach which meets our thesis' goals.

Chapter 7

Conclusion

The personal motivation for this thesis is the author's involvement in the design and implementation of real-time operating systems in the last two decades as kernel architect of PikeOS at SYSGO GmbH. As use cases evolve and domain-specific APIs grow, RTOS products must adapt to changes to remain competitive. Supporting a wide set of domain-specific APIs and requirements in a system such as PikeOS requires a solution that allows to unify the common parts and to keep the subtle differences out of the kernel, if possible. The futex concept used in Linux showed an interesting approach to solve this problem, but also exhibited predictability concerns that needed be addressed. This sparked the author's general interest in the research of fundamentals of blocking synchronization mechanisms. And so a journey started ...

In this thesis, we have systematically explored the design space of fast paths in blocking synchronization mechanisms w.r.t. increased *efficiency* and the necessary *predictability* for WCET analysis. Our system model targets mixed-criticality systems and related operating systems that support *both* real-time and best-effort applications well. From the operating system point of view, we focused on partitioned fixed-priority scheduling, which is the most used scheduling approach for real-time operating systems in practical industrial, avionics, and automotive scenarios.

Our key technique is to *avoid unnecessary system calls*, as system calls have high costs compared to other processor operations, such as atomic synchronization primitives. For this, we have to move *some* operations of synchronization mechanisms from the operating system kernel into user space.

We have presented two general design approaches for blocking synchronization mechanisms, namely futexes and monitors. Futexes implement fast path synchronization based on atomic operations in user space, and the kernel provides a generic *compare-and-block* mechanism as fallback. Futexes are the state-of-the-art mechanism for blocking synchronization in Linux, but with known issues w.r.t. predictability. With deterministic futexes, we have presented a futex design that addresses the predictability issues found in the Linux implementation. We also presented a second futex design, static futex, which is suitable for resource-constrained embedded systems. The novel monitor design provides non-preemptive busy-waiting critical sections in user space and blocking primitives where the wait queue abstraction is kept in user space as well. A key component of monitors is a novel mechanism to change the scheduling priority in user space. This also allows to implement IPCP efficiently.

In our experimental and analytical evaluation, we have shown that the presented design approaches improve the average performance without sacrificing WCET. For example, both futexes and monitors provide blocking mutexes that do not require system calls in the uncontended case. At the same time, all mechanisms show constant or bounded overheads in a worst-case timing analysis.

Our specific contributions in this thesis claimed in Section 1.3 are a generalization of blocking synchronization mechanisms and decomposition into low-level building blocks to identify fast paths, the two futex designs that improve the predictability compared to Linux, the efficient mechanism to change the scheduling priority for IPCP, and the synthesis of the monitor design. We have demonstrated the claims in the course of this thesis and summarized our contributions to the state-of-the-art in Section 6.6.

We proved that exploiting fast paths is a worthwhile approach when designing systems that not only have to fulfill real-time requirements, but also have to hosts best-effort workloads and have secondary requirements like for example decreased energy consumption. The trade-off of the performance gain in the average case versus the increased complexity at the implementation level must be considered in comparison to these secondary requirements.

The practicality and usefulness to build efficient synchronization mechanisms from these building blocks is given, as our evaluation and the use of futexes in Linux show. But this is not only true for industrial APIs like POSIX, but also for avionics and automotive use cases.

The research results presented in this thesis are ready to be transferred into real products in the field, as our examples of deterministic futexes in PikeOS and of static futexes in AUTOBEST show. The monitor approach is currently used in a research operating system only, but we hope that it will find use in real systems as well.

With this thesis, we have presented *some* of the key techniques to achieve both efficient and predictable blocking synchronization, laid a foundation to properly discuss the design trade-offs, and gave criteria for evaluation. However, a lot of open questions and room for improvements in future research remain, as our discussion in Section 6.6 shows.

Bibliography

[AAB15] Hesham Almatary, Neil C. Audsley, and Alan Burns. Reducing the implementation overheads of IPCP and DFP. In *2015 IEEE Real-Time Systems Symposium (RTSS 2015), San Antonio, TX, USA*, pages 295–304, 2015.

[ABLL92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Trans. Comput. Syst.*, 10(1):53–79, 1992.

[AEE15] AEEC. ARINC Specification 653: Avionics Application Software Standard Interface, August 2015.

[AEF$^+$14] Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastian, Reinhard von Hanxleden, Reinhard Wilhelm, and Wang Yi. Building timing predictable embedded systems. *ACM Trans. Embedded Comput. Syst.*, 13(4):82:1–82:37, 2014.

[Al-13] Samy Al-Bahra. Nonblocking algorithms and scalable multicore programming. *Commun. ACM*, 56(7):50–61, 2013.

[AM95] James H. Anderson and Mark Moir. Universal constructions for multi-object operations. In *14th ACM Symposium on Principles of Distributed Computing (PODC '95), Ottawa, ON, Canada*, pages 184–193. ACM, 1995.

[And90] Thomas E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[And93] Arne Andersson. Balanced search trees made simple. In *3rd Workshop on Algorithms and Data Structures (WADS '93), Montréal, QC, Canada*, volume 709 of *Lecture Notes in Computer Science*, pages 60–71. Springer, 1993.

[Bak91] Theodore P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, 1991.

[BAKR19] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. A Fork() in the Road. In *Workshop on Hot Topics in Operating Systems (HotOS '19), Bertinoro, Italy*, pages 14–22. ACM, 2019.

[BB12] Björn B. Brandenburg and Andrea Bastoni. The case for migratory priority inheritance in linux: Bounded priority inversions on multiprocessors. In *14th Real-Time Linux Workshop (RTLWS '12)*, 2012.

[BBB$^+$09] James Barhorst, Todd Belote, Pam Binns, Jon Hoffman, James Paunicka, Prakash Sarathy, John Scoredos, Peter Stanfill, Douglas Stuart, and Russell Urzi. A research agenda for mixed-criticality systems. *Cyber-Physical Systems Week*, 12, 2009.

[BD96] Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Comput. Syst.*, 9(2):131–152, 1996.

[BD19] Alan Burns and Robert I. Davis. Mixed Criticality Systems - A Review. *Department of Computer Science, University of York, Tech. Rep*, March 2019.

[BFC95] Peter A. Buhr, Michel Fortier, and Michael H. Coffin. Monitor classification. *ACM Comput. Surv.*, 27(1):63–107, 1995.

[BGHL87] A. Birrell, J. Guttag, J. Horning, and R. Levin. Synchronization primitives for a multiprocessor: A formal specification. In *11th ACM Symposium on Operating Systems Principles (SOSP '87), Austin, TX, USA*, page 94–102. ACM, 1987.

[BKMS98] David F. Bacon, Ravi B. Konuru, Chet Murthy, and Mauricio J. Serrano. Thin Locks: Featherweight Synchronization for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '98), Montréal, QC, Canada*, pages 258–268. ACM, 1998.

[Bla90] David L. Black. Scheduling support for concurrency and parallelism in the mach operating system. *IEEE Computer*, 23(5):35–43, 1990.

[BLBA07] Aaron Block, Hennadiy Leontyev, Björn B. Brandenburg, and James H. Anderson. A flexible real-time locking protocol for multiprocessors. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007), Daegu, Korea*, pages 47–56. IEEE, 2007.

[BN14] Davidlohr Bueso and Scott Norton. An Overview of Kernel Lock Improvements. *LinuxCon North America, Chicago, IL*, August 2014. `http://events17.linuxfoundation.org/sites/events/files/slides/linuxcon-2014-locking-final.pdf`.

[Bra11] Björn B. Brandenburg. *Scheduling and locking in multiprocessor real-time operating systems.* PhD thesis, The University of North Carolina at Chapel Hill, 2011.

[Bra14] Björn B. Brandenburg. The FMLP+: an asymptotically optimal real-time locking protocol for suspension-aware analysis. In *26th Euromicro Conference on Real-Time Systems (ECRTS 2014), Madrid, Spain*, pages 61–71. IEEE, 2014.

[Bra19] Björn B. Brandenburg. Multiprocessor real-time locking protocols: A systematic review. *CoRR*, abs/1909.09600, 2019. `http://arxiv.org/abs/1909.09600`.

[BRE92] Brian N. Bershad, David D. Redell, and John R. Ellis. Fast mutual exclusion for uniprocessors. In *5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '92), Boston, MA, USA*, pages 223–233. ACM, 1992.

[Bro16] Neil Brown. In pursuit of faster futexes. *LWN.net*, May 2016. `https://lwn.net/Articles/685769/`.

[BSC$^+$11] Bernard Blackham, Yao Shi, Sudipta Chattopadhyay, Abhik Roychoudhury, and Gernot Heiser. Timing analysis of a protected operating system kernel. In *32nd IEEE Real-Time Systems Symposium (RTSS 2011), Vienna, Austria*, pages 339–348. IEEE, 2011.

[BSH12] Bernard Blackham, Yao Shi, and Gernot Heiser. Improving interrupt response time in a verifiable protected microkernel. In *7th European Conference on Computer Systems (EuroSys '12), Bern, Switzerland*, pages 323–336. ACM, 2012.

[Bur12] Alan Burns. A deadline-floor inheritance protocol for EDF scheduled real-time systems with resource sharing. *Technical Report YCS-2012–476, Department of Computer Science, University of York, UK*, 2012.

[BW09] Alan Burns and Andy J. Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX.* Addison-Wesley Educational Publishers Inc, USA, 4th edition, 2009.

[BW13] Alan Burns and Andy J. Wellings. A schedulability compatible multiprocessor resource sharing protocol - mrsp. In *25th Euromicro Conference on Real-Time Systems (ECRTS 2013), Paris, France*, pages 282–291. IEEE, 2013.

[BWKMZ12] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Non-scalable locks are dangerous. In *Linux Symposium*, pages 119–130, July 2012.

[CAB07] John M. Calandrino, James H. Anderson, and Dan P. Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *19th Euromicro Conference on Real-Time Systems (ECRTS '07), Pisa, Italy*, pages 247–258. IEEE, 2007.

[CKZ$^+$13] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *24th ACM Symposium on Operating Systems Principles (SOSP '13), Farmington, PA, USA*, pages 1–17. ACM, 2013.

[Cor14] Jonathan Corbet. MCS locks and qspinlocks. *LWN.net*, May 2014. `https://lwn.net/Articles/590243/`.

[Cra93a] Travis S. Craig. Building FIFO and Priority-Queuing Spin Locks from Atomic Swap. Technical report, University of Washington, February 1993. TR 93-02-02.

[Cra93b] Travis S. Craig. Queuing spin lock algorithms to support timing predictability. In *Real-Time Systems Symposium (RTSS '93), Raleigh-Durham, NC, USA*, pages 148–157. IEEE, 1993.

[DB11] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, 2011.

[DGT13] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *24th ACM Symposium on Operating Systems Principles (SOSP '13), Farmington, PA, USA*, pages 33–48. ACM, 2013.

[Dij68] Edsger W. Dijkstra. The structure of the "THE" multiprogramming system. *Commun. ACM*, 11(5):341–346, 1968.

[DM03] Ulrich Drepper and Ingo Molnar. The Native POSIX Thread Library for Linux. Technical report, Red Hat, Inc., February 2003.

[Dre11] Ulrich Drepper. Futexes Are Tricky. White Paper, November 2011. `https://www.akkadia.org/drepper/futex.pdf`.

[EH13] Kevin Elphinstone and Gernot Heiser. From L3 to sel4 what have we learnt in 20 years of L4 microkernels? In *24th ACM Symposium on Operating Systems Principles (SOSP '13), Farmington, PA, USA*, pages 133–150, 2013.

[ELS88] Jan Edler, Jim Lipkis, and Edith Schonberg. Process management for highly parallel UNIX systems. In *USENIX Workshop on Unix and Supercomputers, Pittsburgh, PA, USA*, pages 1–17, 1988.

[FAA16] FAA. Position Paper CAST-32A: Multi-core Processors, November 2016.

[FRK02] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *Ottawa Linux Symposium, OLS '02*, pages 479–495, 2002.

[GH91] Andy Glew and Wen-mei Hwu. A feature taxonomy and survey of synchronization primitive implementations. Technical Report UILU-ENG-91-2211, University of Illinois at Urbana-Campaign, February 1991.

[GLN01] Paolo Gai, Giuseppe Lipari, and Marco Di Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *22nd IEEE Real-Time Systems Symposium (RTSS 2001), London, UK*, pages 73–83. IEEE, 2001.

[GLQ16] Hugo Guiroux, Renaud Lachaize, and Vivien Quéma. Multicore locks: The case is not closed yet. In *2016 USENIX Annual Technical Conference (USENIX ATC 2016), Denver, CO, USA*, pages 649–662. USENIX Association, 2016.

[GT90] Gary Graunke and Shreekant S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23(6):60–69, 1990.

[HG09] Darren Hart and Dinakar Guniguntalay. Requeue-PI: Making Glibc Condvars PI-Aware. In *11th Real-Time Linux Workshop (RTLWS '09)*, pages 215–227, 2009.

[HLSS09] Wanja Hofer, Daniel Lohmann, Fabian Scheler, and Wolfgang Schröder-Preikschat. Sloth: Threads as interrupts. In *30th IEEE Real-Time Systems Symposium (RTSS '09), Washington, DC, USA*, pages 204–213. IEEE, 2009.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *20th Annual International Symposium on Computer Architecture (ISCA '93), San Diego, CA, USA*, page 289–300. ACM, 1993.

[HP01] Michael Hohmuth and Michael Peter. Helping in a multiprocessor environment. In *2nd Workshop on Common Microkernel System Platforms*, 2001.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., 2008.

[HSS05] Bijun He, William N. Scherer III, and Michael L. Scott. Preemption adaptivity in time-published queue-based spin locks. In *12th International Conference on High Performance Computing (HiPC 2005), Goa, India*, volume 3769 of *Lecture Notes in Computer Science*, pages 7–18. Springer, 2005.

[IEC98] IEC 61508. Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems, 1998.

[IEE17] IEEE. POSIX.1-2008 / IEEE Std 1003.1-2017 Real-Time API, 2017.

[ISO11a] ISO 26262: Road vehicles – Functional safety, 2011.

[ISO11b] ISO/IEC 9899:2011: Information technology – Programming languages – C, 2011.

[JH95] Theodore Johnson and Krishna Harathi. The performance of holding versus releasing locks in a multiprogrammed multiprocessor. Technical Report TR95-013, University of Florida, 1995.

[Jos04] Andrew Josey. API Standards for Open Systems. *The Open Group*, July 2004.

[Kaw05] Kiyokuni Kawachiya. *Java Locks: Analysis and Acceleration.* PhD thesis, Graduate School of Media and Governance, Keio University, Tokyo, Japan, 2005.

[KE95] Steven Kleiman and Joseph R. Eykholt. Interrupts as threads. *Operating Systems Review*, 29(2):21–26, 1995.

[Kee77] J. Leslie Keedy. An Outline of the ICL 2900 Series System Architecture. *Australian Computer Journal*, 9(2):53–62, July 1977.

[KEH$^+$09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an OS kernel. In *22nd ACM Symposium on Operating Systems Principles (SOSP 2009), Big Sky, Montana, USA*, pages 207–220. ACM, 2009.

[KHF$^+$19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP 2019), San Francisco, CA, USA*, pages 1–19. IEEE, 2019.

[KKO02] Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. Lock reservation: Java locks can mostly do without atomic operations. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2002), Seattle, WA, USA*, pages 130–141. ACM, 2002.

[KLMO91] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. Empirical studies of competitve spinning for a shared-memory multiprocessor. In *13th ACM Symposium on Operating Systems Principles (SOSP '91), Pacific Grove, CA, USA*, pages 41–55. ACM, 1991.

[Kop91] Hermann Kopetz. Event-triggered versus time-triggered real-time systems. In *International Workshop on Operating Systems of the 90s and Beyond, Dagstuhl, Germany*, volume 563 of *Lecture Notes in Computer Science*, pages 87–101. Springer, 1991.

[KP98] Andreas Krall and Mark Probst. Monitors and exceptions: How to implement java efficiently. *Concurrency and Computation: Practice and Experience*, 10(11-13):837–850, 1998.

[KWC$^+$16] Namhoon Kim, Bryan C. Ward, Micaiah Chisholm, Cheng-Yang Fu, James H. Anderson, and F. Donelson Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2016), Vienna, Austria*, pages 149–160. IEEE, 2016.

[KWS97] Leonidas I. Kontothanassis, Robert W. Wisniewski, and Michael L. Scott. Scheduler-conscious synchronization. *ACM Trans. Comput. Syst.*, 15(1):3–40, February 1997.

[Kyl14] Kaz Kylheku. What is PTHREAD_MUTEX_ADAPTIVE_-NP? Online article, August 2014. `http://stackoverflow.com/a/25168942`.

[LA93] Beng-Hong Lim and Anant Agarwal. Waiting algorithms for synchronization in large-scale multiprocessors. *ACM Trans. Comput. Syst.*, 11(3):253–294, August 1993.

[LA94] Beng-Hong Lim and Anant Agarwal. Reactive synchronization algorithms for multiprocessors. In *6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '94), San Jose, CA, USA*, pages 25–35. ACM Press, 1994.

[Lie93] Jochen Liedtke. Improving IPC by kernel design. In *14th ACM Symposium on Operating System Principles (SOSP '93), Asheville, NC, USA*, pages 175–188. ACM, 1993.

[Liu00] Jane W. S. Liu. *Real-Time systems.* Prentice Hall, 2000.

[LL73] Chung Laung Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, 20(1):46–61, 1973.

[LR80] Butler W. Lampson and David D. Redell. Experience with Processes and Monitors in Mesa. *Commun. ACM*, 23(2):105–117, 1980.

[LSG$^+$18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 2018), Baltimore, MD, USA*, pages 973–990. USENIX Association, 2018.

[LW01] Jochen Liedtke and Horst Wenske. Lazy process switching. In *8th Workshop on Hot Topics in Operating Systems (HotOS 2001), Elmau/Oberbayern, Germany*, pages 15–18, 2001.

[Mas92] Henry Massalin. *Synthesis: An Effcient Implementation of Fundamental Operating System Services.* PhD thesis, Columbia University, 1992.

[MBM93] Peter J. Moylan, Robert E. Betz, and Richard H. Middleton. The priority disinheritance problem. Technical report, University of Newcastle, Australia, October 1993. EE9345.

[Mic04] Maged M. Michael. ABA prevention using single-word instructions. Technical Report RC23089, IBM Thomas J. Watson Research Center, January 2004.

[MJ93] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter 1993 Technical Conference, San Diego, CA, USA*, pages 259–270. USENIX Association, 1993.

[MLH94] Peter S. Magnusson, Anders Landin, and Erik Hagersten. Queue locks on cache coherent multiprocessors. In *8th International Symposium on Parallel Processing, Cancún, Mexico*, pages 165–171. IEEE, 1994.

[MNNW14] Marshall Kirk McKusick, George V. Neville-Neil, and Robert N. M. Watson. *The design and implementation of the FreeBSD operating system*. Pearson Education, 2014.

[MS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[MS98] Maged M. Michael and Michael L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of parallel and distributed computing*, 51(1):1–26, 1998.

[MSLM91] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. In *13th ACM Symposium on Operating Systems Principles (SOSP '91), Pacific Grove, CA, USA*, page 110–121. ACM, 1991.

[MST$^+$19] Ross McIlroy, Jaroslav Sevcík, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *CoRR*, abs/1902.05178, 2019. `http://arxiv.org/abs/1902.05178`.

[NP12] Jan Nowotsch and Michael Paulitsch. Leveraging multi-core computing architectures in avionics. In *9th European Dependable Computing Conference (EDCC 2012), Sibiu, Romania*, pages 132–143. IEEE, 2012.

[OL13] Jiannan Ouyang and John R. Lange. Preemptable ticket spinlocks: improving consolidated performance in the cloud. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution*

Environments (VEE '13, co-located with ASPLOS 2013), Houston, TX, USA, pages 191–200. ACM, 2013.

[OSE05] OSEK/VDX. Operating System, Version 2.2.3, February 2005.

[Ous82] John K. Ousterhout. Scheduling techniques for concurrent systems. In *3rd International Conference on Distributed Computing Systems, Miami/Ft. Lauderdale, Florida, USA*, pages 22–30. IEEE, 1982.

[Ous90] John K. Ousterhout. Why aren't operating systems getting faster as fast as hardware? In *USENIX Summer 1990 Technical Conference, Anaheim, CA, USA*, pages 247–256, 1990.

[PDEH15] Sean Peters, Adrian Danis, Kevin Elphinstone, and Gernot Heiser. For a microkernel, a big lock is fine. In *6th Asia-Pacific Workshop on Systems (APSys '15), Tokyo, Japan*, pages 3:1–3:7. ACM, 2015.

[Piz16] Filip Pizlo. Locking in webkit. Online article, May 2016. `https://webkit.org/blog/6161/locking-in-webkit/`.

[PMN$^+$09] Rodolfo Pellizzoni, Patrick O'Neil Meredith, Min-Young Nam, Mu Sun, Marco Caccamo, and Lui Sha. Handling mixed-criticality in soc-based real-time embedded systems. In *9th ACM & IEEE International conference on Embedded software (EMSOFT 2009), Grenoble, France*, pages 235–244. ACM, 2009.

[Raj90] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *10th International Conference on Distributed Computing Systems (ICDCS 1990), Paris, France*, pages 116–123. IEEE, 1990.

[Raj91] Ragunathan Rajkumar. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*, volume 151. Kluwer Academic Publishers, 1991.

[Ree76] David P. Reed. Processor multiplexing in a layered operating system. Master's thesis, Massachusetts institute of Technology, MIT/LCS/TR-164, June 1976.

[Rhe96] Injong Rhee. Optimizing a fifo, scalable spin lock using consistent memory. In *17th IEEE Real-Time Systems Symposium (RTSS '96), Washington, DC, USA*, pages 106–114. IEEE, 1996.

[RK79] David P. Reed and Rajendra K. Kanodia. Synchronization with eventcounts and sequencers. *Commun. ACM*, 22(2):115–123, 1979.

[RSI12] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals, Part 1: Covering Windows Server 2008 R2 and Windows 7.* Microsoft Press, Redmond, WA, USA, 6th edition, 2012.

[RSL88] Ragunathan Rajkumar, Lui Sha, and John P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *9th IEEE Real-Time Systems Symposium (RTSS '88), Huntsville, AL, USA*, pages 259–269. IEEE, 1988.

[RTC11] RTCA. DO-178C: Software Considerations in Airborne Systems and Equipment Certification, 2011.

[Sal66] Jerome Howard Saltzer. *Traffic control in a multiplexed computer system.* PhD thesis, MIT, July 1966.

[Sch96] Benoit Schillings. Be Engineering Insights: Benaphores. *Be Newsletters*, 1(26), May 1996.

[SCM$^+$14] Lui Sha, Marco Caccamo, Renato Mancuso, Jung-Eun Kim, Man-Ki Yoon, Rodolfo Pellizzoni, Heechul Yun, Russell Kegley, Dennis Perlman, Greg Arundale, and Richard Bradford. Single Core Equivalent Virtual Machines for Hard Real-Time Computing on Multicore Processors. Technical report, University of Illinois at Urbana-Champaign, November 2014. `http://hdl.handle.net/2142/55672`.

[Sed98] Robert Sedgewick. *Algorithms in C - parts 1-4: fundamentals, data structures, sorting, searching (3. ed.).* Addison-Wesley-Longman, 1998.

[SRL90] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, 1990.

[SS10] Livio Soares and Michael Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2010), Vancouver, BC, Canada*, pages 33–46. USENIX Association, 2010.

[Sun02] Multithreading in the Solaris Operating Environment. White Paper. Sun Microsystems, Inc., 2002.

[SVBD14] Roy Spliet, Manohar Vanga, Björn B. Brandenburg, and Sven Dziadek. Fast on average, predictable in the worst case: Exploring real-time futexes in LITMUSRT. In *35th IEEE Real-Time Systems Symposium (RTSS 2014), Rome, Italy*, pages 96–105. IEEE, 2014.

[Tan09] Andrew S. Tanenbaum. *Modern operating systems.* Pearson, 2009.

[TS94] Hiroaki Takada and Ken Sakamura. Predictable spin lock algorithms with preemption. In *11th IEEE Workshop on Real-Time Operating Systems and Software (RTOSS '94)*, pages 2–6. IEEE, 1994.

[TS97] Hiroaki Takada and Ken Sakamura. A novel approach to multiprogrammed multiprocessor synchronization for real-time kernels. In *18th Real-Time Systems Symposium (RTSS '97)*, pages 134–143. IEEE, 1997.

[ULSD04] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. Towards scalable multiprocessor virtual machines. In *3rd Virtual Machine Research and Technology Symposium (VM '04), San Jose, CA, USA*, pages 43–56. USENIX, 2004.

[Ves07] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE Real-Time Systems Symposium (RTSS 2007), Tucson, AZ, USA*, pages 239–243. IEEE, 2007.

[VL87] George Varghese and Anthony Lauck. Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility. In *11th ACM Symposium on Operating System Principles (SOSP 1987), Austin, TX, USA*, pages 25–38. ACM, 1987.

[WEE$^+$08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008.

[WKS95] Robert W. Wisniewski, Leonidas I. Kontothanassis, and Michael L. Scott. High performance synchronization algorithms for multiprogrammed multiprocessors. In *5th ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP '95), Santa Barbara, CA, USA*, pages 199–206. ACM, 1995.

[WP09] Alex Wilson and Thierry Preyssler. Incremental certification and integrated modular avionics. *IEEE Aerospace and Electronic Systems Magazine*, 24(11):10–15, 2009.

[WRSP15] Qi Wang, Yuxin Ren, Matt Scaperoth, and Gabriel Parmer. Speck: a kernel for scalable predictability. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '15), Seattle, WA, USA*, pages 121–132. IEEE, 2015.

[YYP+13] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '13), Philadelphia, PA, USA*, pages 55–64. IEEE, 2013.

[ZBK14] Alexander Zuepke, Marc Bommert, and Robert Kaiser. Fast User Space Priority Switching. In *10th annual workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT 2014), Madrid, Spain*, July 2014.

[ZBL15] Alexander Zuepke, Marc Bommert, and Daniel Lohmann. AUTOBEST: a united AUTOSAR-OS and ARINC 653 kernel. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '15), Seattle, WA, USA*, pages 133–144. IEEE, 2015.

[ZK18] Alexander Zuepke and Robert Kaiser. Deterministic Futexes Revisited. In *14th annual workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT 2018), Barcelona, Spain*, July 2018.

[ZK19] Alexander Zuepke and Robert Kaiser. Deterministic futexes: Addressing WCET and bounded interference concerns. In *25th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '19), Montréal, QC, Canada*, pages 65–76. IEEE, 2019.

[Zue13] Alexander Zuepke. Deterministic Fast User Space Synchronisation. In *9th annual workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT 2013), Paris, France*, July 2013.

[Zue20] Alexander Zuepke. Turning futexes inside-out: Efficient and deterministic user space synchronization primitives for real-time systems with IPCP. In *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, pages 22:1–22:23, July 2020.

[ZUW20] Xingyuan Zhang, Christian Urban, and Chunhan Wu. Priority inheritance protocol proved correct. *J. Autom. Reasoning*, 64(1):73–95, 2020.

Lebenslauf

Persönliche Daten

Name	Alexander Züpke
Geburtsdatum und -ort	15. Oktober 1978 in Wesel am Niederrhein

Schulischer und beruflicher Werdegang

1985 – 1987	Otto-Willmann Grundschule, Voerde
1987 – 1989	St. Vincentius Grundschule Haffen-Mehr, Rees
1989 – Juni 1998	Gymnasium Aspel der Stadt Rees, Rees Abschluss: Abitur
Juli 1998 – April 1999	Bundeswehr Grundwehrdienst, Kalkar
Mai 1999 – August 1999	Feinmechanisches Grundpraktikum ANDI Maschinenbau, Isselburg
September 1999 – Sept. 2003	Studium der Informationstechnik FH Gelsenkirchen, Abteilung Bocholt Abschluss: Dipl.-Ing. (FH)
September 2003 – heute (seit Oktober 2012 in Teilzeit)	Softwareentwickler und Softwarearchitekt SYSGO GmbH, Klein-Winternheim Entwicklung und Softwarezertifizierung *PikeOS*
Oktober 2012 – März 2016 August 2017 – März 2020 (jeweils in Teilzeit)	Wissenschaftlicher Mitarbeiter Hochschule RheinMain, Wiesbaden Forschungsprojekte *AUTOBEST* und *AQUAS*

Lehre

WS 2012/13 – WS 2018/19	Praktikum *Betriebssysteme u. Rechnerarchitektur* Hochschule RheinMain, Wiesbaden
WS 2019/20	Vorlesung und Praktikum *Betriebssysteme* Hochschule RheinMain, Wiesbaden